

High-Performance Regular Expression Processing for Cross-Domain Systems with High Assurance Requirements

David S. Hardin Konrad L. Slind

Trusted Systems Group
Advanced Technology Center
Rockwell Collins



Collaborators

- Rockwell Collins ATC: David Hardin, Doug Hiratzka, Konrad Slind
- Rockwell Collins Government Systems: Ed Tubbs

Cross-Domain Guards

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



Many guards have been developed over time by different vendors, each with their own particular method of programming and operation.

Turnstile

- Turnstile is a high-assurance guard accredited to DCID 6/3 PL 5 (the highest level), and is UCDMO listed.
- Turnstile was developed by Rockwell Collins under contract to NSA R2, and is currently being used in several applications.
- A variant, called MicroTurnstile, was designed as a “bump-in-the-wire” guard for USB data, and is being productized by the Tactical Army Cross Domain Information Solution (TACDIS) program.
- The basic guard engine software for Turnstile was ported to the Rockwell Collins SecureOne hardware, which is serving as the ground guard for the USAF CRIIS program.

Guardol

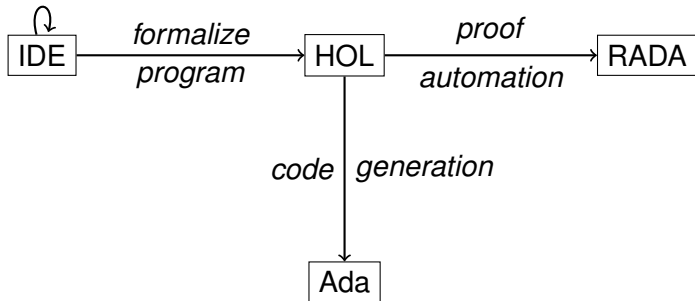
Guardol is a Domain-Specific Language for guards.

Guardol program goals include:

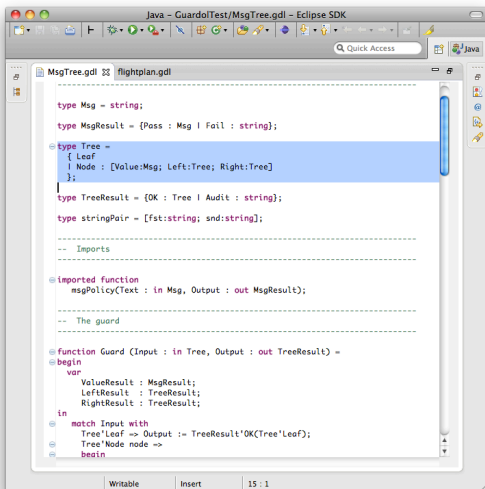
- Provide a single language that can be used to program many guards
 - Guardol has been used to program the Rockwell Collins Turnstile, MicroTurnstile, and the Raytheon High-Speed Guard
 - The Guardol toolchain also produces executables for host-based development and testing
- Integrate highly automated formal verification with development
- Support high-assurance code generation

Guardol Toolchain Architecture

parse; edit



Guardol Eclipse Environment



```
Java - GuardolTest/MsgTree.gdl - Eclipse SDK
MsgTree.gdl flightplan.gdl

type Msg = string;
type MsgResult = {Pass : Msg | Fail : string};
type Tree =
{ Leaf
  | Node : [Value:Msg; Left:Tree; Right:Tree]
};
type TreeResult = {OK : Tree | Audit : string};
type stringPair = [fst:string; snd:string];

-- Imports

-- Imported function
msgPolicy(Text : in Msg, Output : out MsgResult);

-- The guard

function Guard (Input : in Tree, Output : out TreeResult) =
begin
  var
    ValueResult : MsgResult;
    LeftResult : TreeResult;
    RightResult : TreeResult;
  in
    match Input with
    Tree'Leaf => Output := TreeResult'OK(Tree'Leaf);
    Tree'Node node =>
    begin
```

ANTLR grammar example

The Eclipse/Xtext-based Guardol Interactive Development Environment uses ANTLR4 to generate parsers.

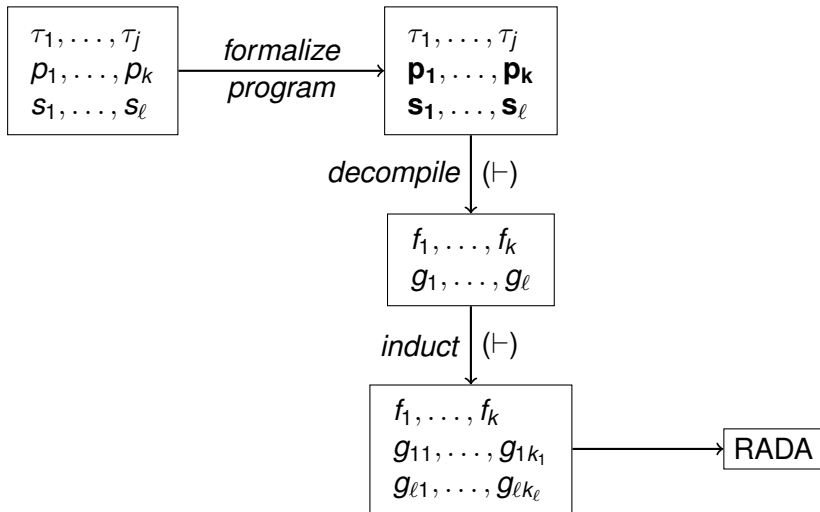
```
expr: ID
     | INT
     | REAL
     | BOOL
     | ID '(' (expr (',' expr)*)? ')'
     | 'not' expr
     | '-' expr
     | expr op=('*' | '/' | 'div') expr
     | expr op=('+' | '-') expr
     | expr op=('<' | '<=' | '>' | '>=' | '=' | '<>') expr
     | expr op='and' expr
     | expr op=('or' | 'xor') expr
     | expr op='=>'<assoc=right> expr
     | expr op='->'<assoc=right> expr
     | 'if' expr 'then' expr 'else' expr
     | '(' expr ')'
```


Verification

HOL4 is used as a **semantical conduit** to RADA

- **RADA** is a SMT-based system for reasoning about catamorphisms
- **HOL4** is an implementation of higher order logic.
- We use it to give a semantics to Guardol evaluation
- Decompilation into logic transforms specs about Guardol program evaluation into properties of HOL functions
- The translation is verified by HOL proof
- Induction schemes from the definition of the functions are used to drive the skeleton of the inductive proof

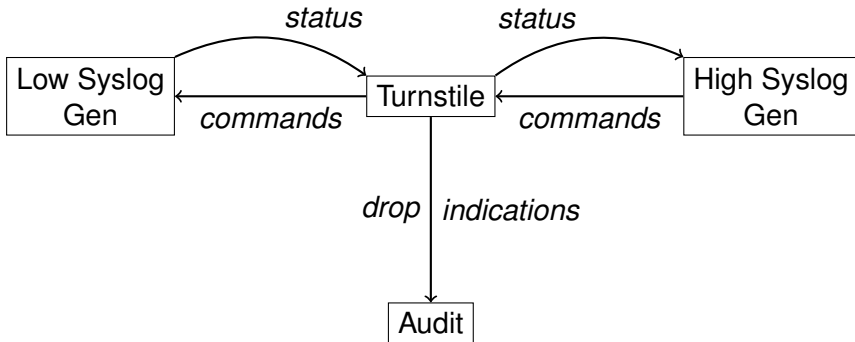
Verification path



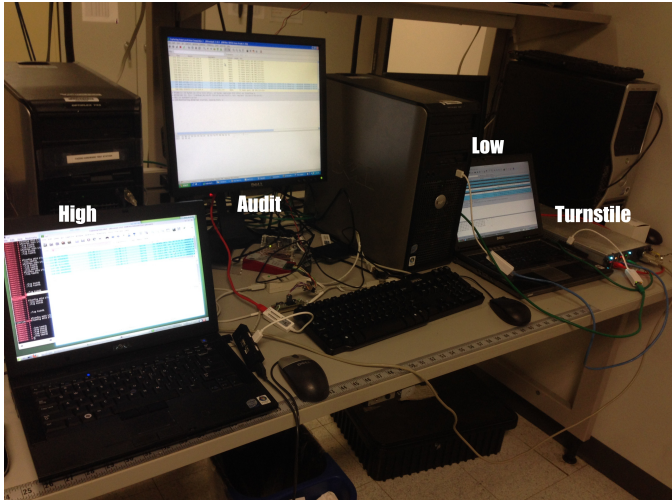
Turnstile/Guardol Demonstration Requirements

- Assume low-to-high and high-to-low traffic employs the syslog protocol, over UDP
- Employ a Turnstile in a two-way guard configuration
- Filter traffic from high to low based on destination IP address, UDP protocol, syslog port number, and value of PRI header field (e.g., <9>).
- Filter traffic from low to high based on destination IP address, UDP protocol, syslog port number, and a regular expression provided by the customer.
- Audit message (in syslog format) is generated when a “bad” packet is detected, and sent over Turnstile’s audit network.

Demonstration Architecture



Demonstration Lab Setup



Issues

Issue: It is impractical to encode the customer's regular expression logic as Turnstile rules.

Solution: Use Guardol on low-to-high transfers, and traditional Turnstile rules for high-to-low transfers.

Issue: Guardol does not support regular expressions.

Solution: Add regular expression support to the Guardol language.

Regular Languages Background

- **Reg = NFA = DFA**
- Two choices for matching a regular expression:
 - interpretation: run a regex interpreter function
 - compilation: translate regex to a DFA that does the match
- Usual story: **Reg** \rightarrow **NFA** \rightarrow **DFA**
 - subset construction on automata
 - awkward handling of negation
- Brzowski story: **Reg** \rightarrow **DFA**
 - algebraic and beautiful
 - handles negation and intersection
 - often generates minimal DFAs

Strange historical fact: Brzowski's work published much earlier, but Ken Thompson's regex matcher for Unix used automata and was very influential.

Extended Regular Expressions: Syntax

The abstract syntax of regular expressions can be given by the following ML-style datatype declaration.

```
regex ::= Epsilon           ; empty string
        | Syms of charset   ; set of characters
        | Not of regex      ; complement      (non-standard)
        | Or of regex regex ; disjunction
        | And of regex regex ; intersection  (non-standard)
        | Cat of regex regex ; concatenation
        | Star of regex     ; Kleene star
```


Extended Regular Expressions: Semantics

The semantics of regular expressions is given in terms of definitions made in Formal Language Theory.

$\mathcal{L}(\epsilon)$	$=$	$\{\epsilon\}$	set of just the empty string
$\mathcal{L}(\mathbf{Symb} P)$	$=$	$\{w \mid \exists x. P x \wedge (w = [x])\}$	character set
$\mathcal{L}(\mathbf{Star} r)$	$=$	$(\mathcal{L}(r))^*$	Kleene star
$\mathcal{L}(\mathbf{Not} r)$	$=$	$\overline{\mathcal{L}(r)}$	complement
$\mathcal{L}(\mathbf{Or} r_1 r_2)$	$=$	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$	union
$\mathcal{L}(\mathbf{And} r_1 r_2)$	$=$	$\mathcal{L}(r_1) \cap \mathcal{L}(r_2)$	intersection
$\mathcal{L}(\mathbf{Cat} r_1 r_2)$	$=$	$\mathcal{L}(r_1) \cdot \mathcal{L}(r_2)$	concatenation

Interpreting Regular Expressions

- Brzozowski (1964) presented an algorithm which can be thought of as an interpreter for membership of string **s** in the language denoted by regular expression **r**.
- The algorithm operates by proceeding, left-to-right, through **s**. At each step, it takes the “derivative” of the regular expression (an innovation due to Brzozowski) with respect to the current symbol in the string.
- Once the string has been completely traversed, the algorithm checks to see if the resulting regular expression has the empty string in its language. If so, **s** is accepted.
- A paper by Owens, Reppy, and Turon (JFP, 2009) extended Brzozowski’s technique to handle character classes, and also to handle larger character sets, such as Unicode.

Brzowski's Derivative

Loosely speaking, Brzowski's Derivative, (**Deriv** x r), computes a regular expression that generates the language obtained by removing symbol x from the front of all strings in **Lang**(r). The basic correctness property of **Deriv**, proved in HOL4, states

$$|- w \text{ IN Lang (Deriv } x \text{ } r) = (x::w) \text{ IN Lang } r$$

The string matching procedure (**Matches**) iterates **Deriv** through a string and checks if the final result has **Epsilon** in its language.

$$\begin{aligned} \mathbf{Matches} \ r \ w &= \mathbf{hasEpsilon}(\mathbf{DerivString} \ w \ r) \\ \mathbf{DerivString} \ (a \cdot w) \ r &= \mathbf{DerivString} \ w \ (\mathbf{Deriv} \ a \ r) \\ \mathbf{Deriv} \ \epsilon \ r &= r \end{aligned}$$

Example

Consider the regular expression

$$[a - t]^*$$

and the strings “combat” and “wombat”. Then, for “combat”,

$$\begin{aligned} & (\text{Deriv } 't' (\text{Deriv } 'a' (\text{Deriv } 'b' (\text{Deriv } 'm' (\text{Deriv } 'o' (\text{Deriv } 'c' [a - t]^*)))))) \\ &= (\text{Deriv } 't' \dots (\text{Deriv } 'o' (\text{norm } (\varepsilon \cdot [a - t]^*))) \\ &= (\text{Deriv } 't' \dots (\text{Deriv } 'o' [a - t]^*)) \\ &\dots \\ &= [a - t]^* \end{aligned}$$

The empty string is in the language

$$\mathcal{L}([a - t]^*)$$

Thus, the string “combat” matches the regular expression.

Example (cont'd.)

On the other hand, for the string “wombat”,

$$\begin{aligned}
 & (\text{Deriv } 't' (\text{Deriv } 'a' (\text{Deriv } 'b' (\text{Deriv } 'm' (\text{Deriv } 'o' (\text{Deriv } 'w' [a - t]^*)))))) \\
 &= (\text{Deriv } 't' \dots (\text{Deriv } 'o' (\text{norm } (\emptyset \cdot [a - t]^*)))) \\
 &= (\text{Deriv } 't' \dots (\text{Deriv } 'o' \emptyset)) \\
 &\dots \\
 &= \emptyset
 \end{aligned}$$

Thus, the string “wombat” does not match the regular expression.

Interpreter Correctness

The correctness of **Matches** follows from the correctness of **Deriv** by induction on the length of the string.

$$\vdash \mathbf{Matches} \ r \ s \iff s \in \mathbf{Lang} \ r$$

This theorem is a simple result to prove in HOL4. Using it, reasoning about the match interpreter can be replaced by reasoning about membership in the language of r . This is actually simpler in general.

Matches is an executable function, but it is quite slow, especially when compared to automata. For execution it is better to use a DFA (deterministic finite state automaton).

Compiling Extended Regular Expressions to DFAs

One can think of Brzozowski's regex compilation algorithm (**Brz**) as compiling a regular expression to a DFA, which is subsequently run on strings.

The essential insight behind **Brz** is that regexes are identified with DFA states:

- The given regex r_0 is the start state
- For each symbol a_i in the alphabet, compute $r_{a_i} = \mathbf{Deriv} a_i r$
- The r_{a_i} are the successor states to r
- Stop when no new states are created
- Final states are those that match the empty string

Compiling Regular Expressions to DFA's

Let r be a regular expression. Then

$$\mathbf{Brz}(r) = \{init, trans, final\}$$

where the components of the automaton have the following signature:

```
init   : state                ; the start state
trans  : state * symbol -> state ; the transition function
final  : state -> bool         ; the set of final states
```


Compiling Regular Expressions to DFA's (cont'd.)

The following pseudo-code executes DFA **d** on input **s**:

```
Exec_DFA (d:DFA, s:string) returns verdict:bool = {  
  var  
    q, len : int;  
  in  
    len := s.Length;  
    q := d.init;  
    for (i=0; i<len; i++) { q := d.trans[q,s[i]]; }  
    verdict := member(q,d.final);  
}
```

Brzowski proved that, for all extended regular expressions **r**, and strings **s**:

$$\vdash (\mathbf{Exec_DFA}(\mathbf{Brz}(r))\ s = \mathbf{true}) \iff s \in \mathcal{L}(r)$$

Regular Expressions in Guardol

We have extended the Guardol language with a new primitive expression

regex_match(*rlit*, *s*)

which takes a regular expression literal *rlit*, and a string expression *s*, and returns a boolean result indicating whether $s \in \mathcal{L}(r)$, where *r* is the **regexp** corresponding to *rlit*.

Guardol Regular Expression Literals

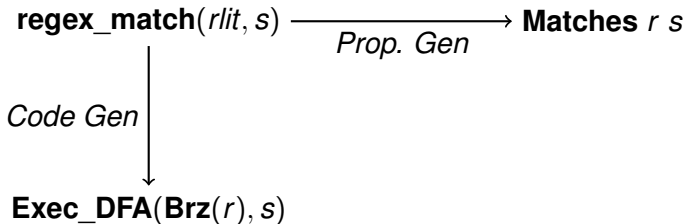
Regular expression literals in Guardol largely conform to the syntax in the Python programming language.

```
\d = 0..9
\w = [a-zA-Z0-9_]
. = any char except \n
\s = whitespace = [ \n\r\t\f] (* Note the space character! *)
\t = tab
\n = newline
\r = return
\f = formfeed
\c = escape c
```

```
rs = concatenation
r|s = disjunction
r* = Kleene star
r+ = rr*
r? = "" | r
r{n} = r^n
r{m,n} = r{m} | r{m+1} | ... | r{n} (m<=n)
r{m,} = r{m}r*
r{,n} = r{0,n}
(r) = grouping
[...] = character set
```

Regular Expressions in Guardol (cont'd.)

A regex expression in a Guardol program is treated differently depending on whether code is being generated, or whether properties are to be proved.



Guardol Demo program utilizing regular expressions

```
package Regex =  
  
-- Filter for full syslog message. Meant to handle messages conforming to either  
-- RFC 5424 or RFC 3164. Skips over leading information by looking for an occurrence  
-- of a space followed by an open bracket, i.e., " [". After that, it expects  
-- the remainder of the structured data portion of the message.  
  
function syslog_5424_or_3164_filter (input : in string) returns verdict : bool =  
{  
  verdict := regex_match(  
    \.* \[{\{"time": "\d{13}(\:\d{3})?"}, "\w{1,20}": {(\{"\w{1,25}": "\w{1,30}", ?)+\}\}\}\},  
    input);  
}  
end
```

In English:

"A JSON array contains an object, with key "time" and a value of a string of numbers, surrounded by quotes, 13 characters long, and possibly ending with a colon and 3 additional numerals. Next is another key, 1 to 20 alphanumeric (plus '_' characters) long, whose value is an object having one or more groups of keys that are a quoted string of 1 to 25 alphanumeric (plus '_' characters), and values that are 1 to 30 alphanumeric (plus '_' characters) long, also quoted."

Generated Ada code

```
package body Regex is

function execDFA_1 (str : in String) return Boolean is
  verdict : Boolean;
  state : uint;
  len : uint;
  i : uint;
begin
  state := Regex.DFA_1.start;
  i := 0;
  len := str'Length;
  while (i < len) loop
    i := (i + 1);
    state :=
      Regex.DFA_1.trans (Natural (state), Natural (Character'Pos (str(i))));
  end loop;
  verdict := Regex.DFA_1.final (Natural (state));
  return (verdict);
end;

function syslog_5424_or_3164_filter (input : in String) return Boolean is
  verdict : Boolean;
begin
  verdict := Regex.execDFA_1 (input);
  return (verdict);
end;
end Regex;
```

Results

- We have generated UDP syslog packets that conform/fail to conform to the customer's regular expression input, and have filtered these packets with a Turnstile two-way guard.
- For high-to-low traffic, we filter based on destination IP address, UDP protocol, syslog port number, and value of PRI header field (e.g., <9>).
- For low-to-high traffic, we filter based on destination IP address, UDP protocol, syslog port number, and a regular expression provided by the customer.
- Audit message (in syslog format) is generated when a "bad" packet is detected, and sent over Turnstile's audit network.

Future Work: Automating Regex Proofs

Although many properties of regular languages are decidable, such problems are typically a question of whether a given concrete literal string has a particular property.

However, in our work we expect that the string has been constructed by application of a function. This makes things more interesting.

As a simple example, suppose that we have implemented a dirty-word function (**Elim_Dirty**) that replaces a particular string, say “monkey” by dashes:

"monkey" --> "-----"

Future Work: Automating Regex Proofs (cont'd.)

Then we would be able to write a specification of the following form:

```
spec Elim_Dirty_Correct = {  
  var s1,s2 : string;  
in  
  s2 := Elim_Dirty(s1);  
  check regex_match(`~(. *monkey. *)`, s2);  
}
```

Note the use of negation to succinctly express the property.

Future Work: Automating Regex Proofs (cont'd.)

The resulting proof obligation is to show that

$\forall s. \mathbf{Matches}(\mathbf{parse} \text{ '}\neg(. * \mathit{monkey}.*)\text{'}) (\mathbf{Elim_Dirty} \ s)$

i.e., that the string "monkey" is not a substring of the result returned by **Elim_Dirty**.

This will only be provable by reasoning inductively about **Elim_Dirty** and **Matches**.

We hope to follow the catamorphism approach taken by our work with Rada.

Future Work: Verified Regex Compilation

We will be strengthening the link between the generation of DFAs and applications of the **Matches** function by formally showing that the DFA and the **Matches** function always return the same result. This is essentially Brzozowski's proof.

Once that is accomplished, we can data-refine to the array-based code ultimately generated:

$$\begin{aligned}
 s \in \mathcal{L}(r) &\text{ iff } \mathbf{Matches} \ r \ s && \text{; already proved} \\
 &\text{ iff } \mathbf{Exec_Fmap_DFA}(\mathbf{Brz}_{fmap}(r)) \ s && \text{; finite map} \\
 &\text{ iff } \mathbf{Exec_Array_DFA}(\mathbf{Brz}_{2Darray}(r)) \ s && \text{; 2D array}
 \end{aligned}$$

Future Work: Verified Code Generation

We are currently investigating the use of the verified **cakeML** compiler (Kumar, Myreen, Norrish, and Owens, POPL'14) to compile the code, and apply the **cakeML** correctness theorem to establish a Guardol-to-binary correctness result.

This “direct to machine code” approach would eliminate the need to trust the translation to source code (Ada, C, ...) as well as eliminate the need to trust the output of a compiler.

THE END