Introduction to the Guardol Programming Language and Verification System

David Hardin* Konrad Slind* Michael Whalen†
Tuan-Hung Pham†

September 15, 2011

Abstract

Guardol is a high-level programming language intended to facilitate the construction of correct network guards. The Guardol system generates Ada code from Guardol programs. It also provides specification and automated verification support: guard specifications are formally translated to SMT format and passed to a new decision procedure dealing with functions over tree-structured data. The result is that difficult properties of Guardol programs can be proved fully automatically.

Guardol is a programming language and support environment being developed by the Trusted Systems group in the Advanced Technology Center of Rockwell Collins. Guardol is aimed at making the process of specifying, implementing, and certifying high assurance guards more efficient, flexible, and retargetable. The motivation for developing Guardol comes from experience Rockwell Collins has in developing guard implementations. Although execution aspects of programs (*e.g.*, speed and size) are undoubtedly important, we have focused on a number of other significant aspects as well: the ability to target a wide variety of guard platforms; the ability to glue together existing or mandated functionality; the generation of both implementations and formal analysis artifacts; and sound and highly automated formal analysis.

What is a guard? A *guard* mediates information sharing between security domains according to a specified policy. Some typical guard operations on a packet stream are the following: read field values in a packet; change fields in a packet; transform a packet by adding new fields; drop fields from a packet; construct audit messages; and remove an entire packet from stream.

*Rockwell Collins Advanced Technology Center
†University of Minnesota

Design considerations An important consideration for us is that the messages to be guarded may not be of fixed size. For example, email messages, or XML encodings, can have branching tree structure, and Guardol is designed to be able to easily declare complex tree-structured data and guarding functionality that processes such data. Thus a major aspect of the design of Guardol is the provision of datatype declaration facilities similar to those available in functional languages such as SML [3]. Recursive programs over such datatypes then have to be supported, and ML-style pattern-matching is incorporated to facilitate such processing. However, appealing as it may be to some, Guardol is not simply an adaptation of a functional programming language to guards. In fact, much of the syntax and semantics of Guardol is similar to that of Ada: Guardol is a sequential imperative language with assignment, sequencing, conditional commands, and procedures with **in/out** variables. Roughly,

$$\mathbf{Guardol} = \mathbf{Ada} + \mathbf{ML}\ .$$

This hybrid language provides a high level of support for writing complex programs over complex datastructures, while also providing standard types and programming constructs from Ada, which should be familiar to our intended customers.

What Guardol doesn't have Guardol is not Turing complete, it has no pointers, and it has no I/O facilities. These choices are all made in order to support an effective verification environment, but have other justifications as well. A guard language should not be Turing complete *i.e.*, admit infinite computations, since it is expressly required that a guard, given an input, always returns an output. From the point of view of verification, termination means that induction can be used to reason about looping constructs; otherwise, less familiar reasoning methods need to be used. IO operations are also not part of Guardol since guards are typically embedded in a computation fabric that supplies inputs and deals with the output: we want to support the declaration and analysis of the guard itself, and not its support environment. Finally, pointers tend to complicate program analysis and verification, so we have dispensed with them. Tree-structured data of unbounded size is well supported in Guardol, but the programmer is not exposed to the pointer technology underlying the abstraction.

1 The Guardol System

The Guardol system integrates several distinct components, as illustrated in Figure 1. A Guardol program in file `x.gdl` is parsed and typechecked by the Gryphon framework [2] developed by Rockwell Collins. Gryphon provides a collection of passes over Guardol ASTs that help simplify the program.

If the user chooses to generate Ada code corresponding to `x.gdl`, then further passes over the program are made, *e.g.*, to enforce Ada requirements on

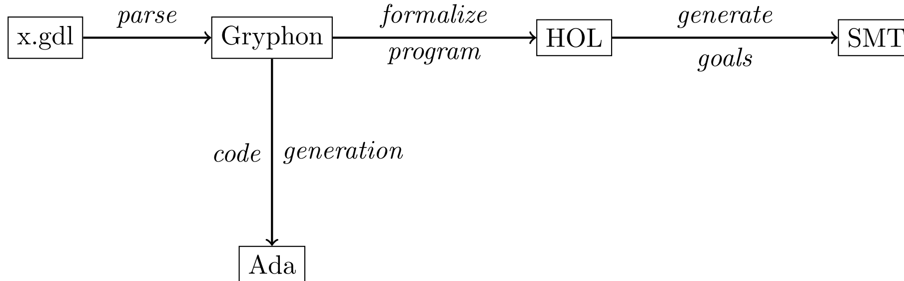

Figure 1: Guardol System Components

identifiers and string literals, before specification (`.ads`) and implementation (`.adb`) files are created for the program. Also generated are implementations of any datatypes declared in `x.gdl`. A reference-counting memory scheme using so-called *smart pointers* is used for allocating and freeing datatype elements.

If the user chooses to verify the code in `x.gdl`, the reasoning tools HOL4 [5] and OpenSMT [1] become involved. First, the programs in `x.gdl` are translated into formal analogues in HOL4. The formalized programs are based on a logical theory of the operational semantics for Guardol. One could reason in HOL4 about such programs, directly using the operational semantics. However, such an approach is awkward and requires advanced expertise in the use of a higher order logic theorem prover. We would instead like to make use of the high automation promised by current SMT systems.

There remain two problems; the first is that current SMT systems do not understand operational semantics. We surmount this by use of 'decompilation into logic', a technique established by Magnus Myreen in his PhD thesis [4]. It enables us to deductively map from properties of programs in an operational semantics to equivalent properties over mathematical functions. This automatic transformation moves the verification task into the realm of proving properties of recursive functions operating over recursive datatypes. This raises the second problem: until recently SMT systems were unable to reason about functional programs. We have surmounted this by implementing a decision procedure for functional programming due to Suter, Dotta, and Kuncak [6]. This decision procedure necessarily has limitations, but it is able to handle a wide variety of interesting programs and their properties fully automatically.

2 The Guardol language by example

In the following we will get a taste of Guardol by examining a simple guard. The guard applies a platform-supplied *dirty-word* operation over a binary tree of messages. When applied to a message, the operation could leave it unchanged, could change it, or could reject it. If the message is rejected, an audit message

is immediately returned; otherwise, processing the tree of messages continues.

Packages In Guardol, an entire program is represented by a set of *packages*. A package collects data declarations, program declarations (procedures and functions), and specifications. An element of one package may be referred to in another by use of the 'dot' notation, *e.g.*, `Pkg.foo`. In our example, there is only one package, named `DWOTree`:

```
package DWOTree = begin <body of package> end
```

Data declarations Now we will explore the body of the package. To provide a degree of abstraction, one can declare type synonyms, or abbreviations. In our case, the type of messages is `Msg`, which will stand for `string`:

```
type Msg = string;
```

The use of abbreviations allows altering the representation of messages in the future without having to further edit the remainder of the package. Now we define the recursive datatype of message trees. These are binary trees, with messages being held at internal nodes.

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

The two constructors of the datatype are `Leaf` and `Node`. A `Leaf` has no arguments, so is just used to signal the ends of paths in the tree. A `Node` has a single argument, a record with three fields. The `Value` field contains a message, and the `Left` and `Right` fields hold the two subtrees of the node.

When the guard processes a `MsgTree` it either returns a new, possiby modified, tree, or it returns an audit message. These two kinds of return values are represented in the following datatype declaration:

```
type TreeResult = { TreeOK : MsgTree | TreeAudit: string };
```

External procedures Now we discuss the behavior of the code that performs the dirty-word operation on messages in the tree. This operation will be provided externally, via a library, or some other means. All that is known about it is that it processes a message, and either: succeeds, returning a message; or fails, returning an audit string. These return values are elements of the following datatype:

```
type MsgResult = { MsgOK : Msg | MsgAudit : string};
```

The externally-provided dirty-word operation is declared using format which is, by design, quite similar to the header format for Ada progams. The declaration tells us only that the dirty word operation takes a message as input via the `Text` argument and assigns the result of its processing to the `Output` variable.

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

The guard The guard procedure takes its input tree in variable `Input` and the return value, which has type `TreeResult`, is placed in `Output`. The body uses local variables for holding the results of recursing into the left and right subtrees, as well as for holding the result of calling the external dirty-word operation. Thus the beginning of the guard function is written as follows:

```
function Guard (Input : in MsgTree, Output : out TreeResult) =
begin
var
  ValueResult : MsgResult;
  LeftResult, RightResult : TreeResult;
in  ...   end
```

Match statements The guard processes a tree by a pattern-matching style case analysis on the structure of the tree. The format of such code is

$$\texttt{match}\ exp\ \texttt{with}\ pat_1 \Rightarrow stmt_1; \ldots;\ pat_n \Rightarrow stmt_n$$

where each of pat_i, is a constructor pattern giving one of the possible ways that *exp* could be built. In the guard, the `Input` variable is the *exp* being analyzed:

```
match Input with ...
```

Now let's consider the possible cases. If `Input` is a leaf node, processing succeeds. This is accomplished by tagging the leaf with `TreeOK` and assigning to `Output`:

```
MsgTree'Leaf => Output := TreeResult'TreeOK(MsgTree'Leaf);
```

Otherwise, the tree under scrutiny is an internal node. In that case the guard recurses through the subtrees and also applies the dirty-word operator to the message held at the node. This computation is fairly straightforward to describe, except for the added complication that an audit may be generated by any of the three subcomputations.

```
MsgTree'Node node => begin
  DIRTY_WORD_OPR(node.Value, ValueResult);
  ...
end
```

First, the pattern match checks that the tree is a node (`MsgTree'Node node`). In the remainder of the clause, the elements of the node can be referred to with the dot notation, *e.g.*, `node.Value`, `node.Left`, and `node.Right`. The clause starts by invoking the dirty-word operator on `node.Value`, storing the result in `ValueResult`. The result of the call must then be examined to

see if the operation succeeded or failed. This is again implemented by pattern-matching. If the operator decides that the message should be rejected, it returns `MsgAudit`, with an accompanying message `A`. This is then converted to a message rejecting the original input tree, by constructing a tree-level audit message, and assigning it to the `Output` variable.

```
match ValueResult with
   MsgResult'MsgAudit A => Output := TreeResult'TreeAudit(A);
   MsgResult'MsgOK ValueMsg => ...
```

Otherwise, computation proceeds to the subtrees. Since the processing is essentially symmetric, we will look at only the left subtree. A recursive call is made to `Guard`, and then the result (held in `LeftResult`) is scrutinized. An audit is immediately propagated.

```
Guard (node.Left, LeftResult);
match LeftResult with
  TreeResult'TreeAudit A => Output := LeftResult;
  TreeResult'TreeOK LeftTree => ...
```

On the other hand, if processing the left tree is successful, the right tree is scanned. If no messages in the right subtree generate an audit, processing of the entire tree succeeds and the result tree is assigned to `Output`:

```
Output := TreeResult'TreeOK (MsgTree'Node
            [Value: ValueMsg, Left: LeftTree, Right:RightTree]);
```

The entire guard code follows:

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

3 Generating Implementations

Once a guard has been written in Guardol, one of the things we may want to do is generate an implementation from it. The Guardol system automatically generates guard implementations—currently in Ada—from Guardol descriptions. We use the **Gryphon** framework developed at Rockwell Collins to translate from Guardol source text to Ada. For the most part, this is conceptually simple since much of Guardol is a subset of Ada. However, one major difference is that Guardol allows ML-style datatypes, plus pattern-matching. The latter requires an ML-style pattern-match compiler. The former requires automatic memory management, which we have implemented via a reference-counting style garbage collection scheme.

Even after code has been generated from a Guardol program however, some more work has to be done to achieve an executable:

- Implementations for external functions and types have to be provided, say from a pre-existing library.
- A computational environment for the guard to run in has to be implemented. This will supply the guard with messages and deal with the output of the guard (success or audit).

For lack of space, we will omit further details.

4 Specifying guard properties by example

A *specification* in Guardol looks somewhat like a parameterized unit test: it presents some code to be executed, sprinkled with assertions. An assertion needs to hold on every computation path reaching the assertion. Following is the specification for the guard:

```
spec Guard_Correct =
begin var t : MsgTree;
          r : TreeResult;
in
   if (forall (M : Msg). DWO_Idempotent(M)) then
   begin
      Guard(t,r);
      match r with
        TreeResult'TreeOK u => check Guard_Check(u);
        TreeResult'TreeAudit A => skip;
   end
   else skip;
end
```

How does this ensure correctness? Recall that the behavior of the external function `DIRTY_WORD_OPR` is completely unconstrained, other than that it takes a `MsgTree` as input and returns a value of type `TreeResult` as output. Thus the

guard code is, in essence, parameterized by an arbitrary policy on how messages are treated. One way of capturing the desired behavior is to require that the result tree has been cleaned *according to the policy.* In other words, suppose we run the guard on tree t, obtaining tree u. If we now run the dirty-word operation on every message in u, we should get u back unchanged, since all dirty words have already been scrubbed out in the passage from t to u. This property is a kind of *idempotence* check, which we have coded up in the function `Guard_Check`; note that it has the shape of a *catamorphism*, which is a simple form of recursion exploited by our automatic proof component.

```
function Guard_Check (MT : in Tree) returns Output : bool =
begin var R : MsgResult;
in
 match MT with
   MsgTree'Leaf => Output := true;
   MsgTree'Node node =>
   begin
     DIRTY_WORD_OPR(node.Value, R);
     match R with
        MsgResult'MsgOK M    => Output := (node.Value = M);
        MsgResult'MsgAudit A => Output := false;
     Output := Output and Guard_Check(node.Left)
                      and Guard_Check(node.Right);
   end
end
```

The ultimate success of the proof depends on the assumption that the external dirty-word operation is idempotent on messages. This is expressed by the function `DWO_Idempotent`[1] which calls `DIRTY_WORD_OPR` twice, and checks that the result of the second call is the same as the result of the first call. If the first call returns an audit, then there is no second call, so the idempotence property is vacuously true. On the other hand, if the first call succeeds, but the second is an audit, that means that the first call somehow altered the message into one provoking an audit, so idempotence is definitely false.

As we have shown, the correctness of the guard has been specified largely in Guardol code. There is almost no logical syntax involved except for one (necessary) quantifier. Auxiliary functions required to make the specification, like `Guard_Check` and `DWO_Idempotent`, have also been declared as Guardol programs. Now we turn to a discussion of how these properties are verified.

5 Verifying guard properties

The Guardol approach to verification uses formal modelling and proof, using the HOL4 theorem prover as a semantical conduit to SMT decision procedures. Higher order logic is used to give semantics to Guardol programs, and is also used

[1] Definition omitted for lack of space.

to translate from the semantic representation to a representation suitable for the SMT prover. There are a few interesting internal steps that are made, and we will examine them. First, the Guardol package is translated into a HOL theory. In this stage, Guardol types are are mapped to HOL types, Guardol expressions map to HOL expressions, and Guardol functions and procedures are translated to a HOL type of abstract syntax trees (ASTs). Program evaluation is captured by a conventional inductively defined operational semantics. Thus this phase of processing maps from Guardol syntax into a semantic representation. In order to reason about the program we reason about the semantic representation. One way to prove program properties would be to reason about program evaluation by using induction on the evaluation relation. However, experience has taught us that this is an overly difficult approach.

Instead, we employ decompilation into logic as a way to soundly map from (1) a situation in which we need to reason about programs using tools of operational semantics to (2) a situation in which we reason solely about mathematical functions. Without going into detail, $fn = \mathcal{D}(c)$ is the the mathematical function resulting from decompiling code c. It always returns the same result as c computes, and thus reasoning about fn can replace reasoning about c. In effect, the execution of c has been evaporated away.

Recall the `Guard_Correct` specification. Roughly, it says *If running the guard succeeds, then running `Guard_Check` on the result returns* true. Applying decompilation to the code of the specification and using the resulting theorem to map from the operational semantics to the functional interpretation, we obtain the goal:

$$\left(\begin{array}{l} (\forall m\ b.\ \mathsf{DWO_Idempotent}\ ext\ m) \land \\ (\forall r.\ \mathsf{Guard}\ ext\ t = \mathsf{TreeResult_TreeOK}\ t') \end{array} \right) \Rightarrow \mathsf{Guard_Check}\ ext\ t'$$

where DWO_Idempotent, Guard, and Guard_Check are the functional analogues of Guardol procedures produced by the decompilation. This goal has the form required by the SMT prover, namely that the catamorphism Guard_Check is applied to the result of calling Guard. After a few more preparatory steps the formula is passed to the decision procedure, which proves the goal automatically, thus showing that the `Guard` procedure is idempotent on message trees, given an external policy that is idempotent on messages.

The Suter-Kuncak decision procedure SMT systems have become very popular in recent years because of algorithmic advances in deciding common theories and because mechanisms for coordinating multiple decision procedures have been redesigned to use SAT technology. In recent work [6], Suter and Kuncak proposed a decision procedure for a fragment of functional programs operating over algebraic datatypes. The procedure decides formulas $P(cat\ t)$ where P is a formula in a decidable theory, t is an element of a tree-structured datatype, and cat is a catamorphism (also known as a *fold* to functional programmers). The system design is displayed in Figure 2; it is a refinement of the usual DPLL architecture used in SMT system design, necessitated by the fact

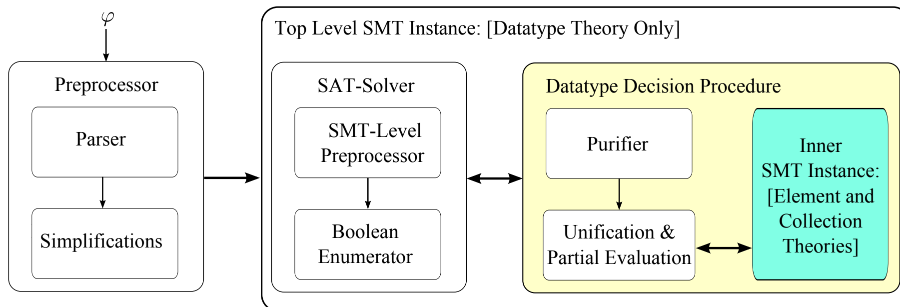

Figure 2: Architecture for SMT Solver containing Suter-Kuncak

that the Suter-Kuncak decision procedure needs to control a certain amount of pre-processing usually controlled by the DPLL architecture. As a result, our design re-architects a standard DPLL solver somewhat to get the solver to pass all theory terms through the Suter-Kuncak decision procedure where we could purify and partially evaluate them and feed them back to an SMT solver.

6 Conclusion

We have introduced the Guardol language and its verification subsystem, largely by example. Our goal is to develop the language and proof technology in order to effectively and automatically prove that a wide variety of guards satisfy important security properties. In the future we will be applying and evaluating Guardol on real-world guards.

References

[1] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT Solver. In *Proceedings of TACAS 2010*, pages 150–153, 2010.

[2] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53:58–64, February 2010.

[3] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[4] Magnus Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2009.

[5] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Proceedings of TPHOLs 2008*, volume 5170 of *Springer LNCS*, 2008.

[6] P. Suter, M. Dotta, and V. Kuncak. Decision Procedures for Algebraic Data Types with Abstractions. In *Proceedings of POPL 2010*. ACM.