UiO : Department of Informatics
University of Oslo

An Applied Example of IMAG in JastAdd

for Specification and Dynamic Enforcement of Security Policies

Toktam Ramezanifarkhani

Research report 490, April 19, 2022

ISBN 978-82-7368-455-4
ISSN 0806-3036

An Applied Example of IMAG in JastAdd

for Specification and Dynamic Enforcement of Security Policies

Toktam Ramezanifarkhani
Department of Informatics, University of Oslo, Norway, (e-mail: toktamr@ifi.uio.no), *Member, IEEE*

Abstract

Attribute Grammars have been powerful and the most widely applied formalism in language processing in a wide variety of application domains. Inline Monitoring Attribute Grammar (IMAG) is an extension of AGs providing a framework with the ability to specify programming languages with the innate power of security policy enforcement at runtime for written programs in specified languages. In this document, we provide an applied example of IMAG in JastAdd for specification and dynamic enforcement of security policies.

Index Terms

Language-based security, Attribute Grammar, Semantic Specification, Inlined Monitoring, Security Policy Enforcement, Data Flow Integrity

We provide an applied example in JastAdd as an open-source system and an extensible Java compiler [1]. We built our example on top of the running state machine example of JastAdd [2], which supports several analyses and properties. More implementation details are available in the JastAdd documentation, such as manual [3] and the tutorial [2], and see [4] for more detail of our example.

JastAdd allows attributes to be programmed declarative. A parser that builds the AST in the semantic actions from a text can be generated using a conventional parser generator. JastAdd generates a Java API from the abstract grammar including classes with AST constructors and AST traversal methods. Moreover, accessing the attributes is through methods. Due to intertype declarations in JastAdd, the attributes and semantic rules are defined in aspects. The attribute declarations that appear in the aspect file belong to the related AST classes, and JastAdd reads the aspect files and inserts these intertype declarations into the AST classes.

The state machine example that we are going to extend includes states with directional labeled transitions used to model programs with the aim of analysis purposes. To specify a simple policy and its enforcement, let us assume that multiple branches like those caused by switch statements are used for security purposes such as authentication. Moreover, a common policy in such cases is to empty all the used temporal variables afterward. It means in the state machine of programs satisfying this policy, the next instruction after the switch statements should be an instruction to empty the temporal variables, let us call it variable-emptiness or VE. Therefore, the state after a switch statement should have a transition labeled VE to the next state. To check the emptiness, we consider a policy p_{emp} dictating that the label for those transitions that their source state has several incoming edges should correspond to the specific label VE. Such a policy might be selected for security-critical applications and IRM implementation, and applicable to the programs labeled state transition machine model.

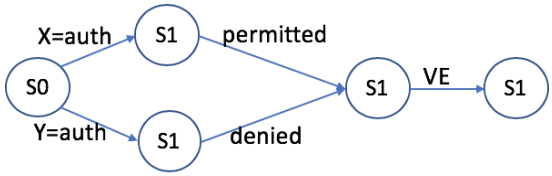
A simple example of such a code with its state machine is shown in Fig. **??**. The first part of this example, Fig. **??** (a), is a switch statement for the authentication purpose and (b) is its state machine which will be the input code text for the IMAG, and (c) shows a graphical view of the state machine. To provide an automatic approach to specify and enforce the p_{emp} policy in the state machine attribute grammar, we explain p_{emp}-IMAG in JastAdd with more details available at [4]. As the result of syntax and semantic analysis of the input, the policy specification and its automatic enforcement will be a part of the state machine attribute grammar and involved in the process of parsing, analysis, compilation, and code generation.

A. p_{emp}-IMAG

The state machine example in the form of inline monitoring attribute grammar, p_{emp}-IMAG, is explained to enforce the policy p_{emp}, automatically. p_{emp}-IMAG is shown in Fig. 1 and 2, which later would be explained in the notation of JastAdd abstract grammar, APIs, and aspects.

In this grammar, the state machine is extended by having two other non-terminals $History$ and $Connect$. $History$ is an NT with a *history attribute*, His, which is used to collect basic information of statuses through semantic rules at attribute evaluation time. This attribute should be a write-access and protected live attribute. After semantic analysis and finalizing this information at compile-time, it will be saved in the protected memory by the compiler. Later, through the enforcement code, this information will be used and manipulated to provide the expected history state of the program at runtime. Providing the protected part and its secure interface to store the protected-live attributes can be done with different approaches. None of the examples or the IMAG process is dependent on any specific implementation approach. However, we used an approach similar to [5] for this aim. $Connect$ is an ISN defined in IMAG to instrument the program to securely connect to the basic information of the history attribute at runtime if the instrumentation flag is on. It is done in the semantic production having an instrumentation semantic rule, p_{sp_0}. StateMachine input attribute env is an *instrumentation flag* that can be considered an

$switch(auth)\{$
$case\ x : printf(\text{“}permitted\text{”});$
$case\ y : printf(\text{“}denied\text{”});$
$\}$
$EmptyVariavles();$
(a)
$state\ S0;$
$state\ S1;$
$state\ S2;$
$trans\ x = auth : S0 -> S1;$
$trans\ y = auth : S0 -> S2;$
$state\ S3;$
$state\ S4;$
$trans\ permitted : S1 -> S3;$
$trans\ denied : S2 -> S3;$
$trans\ emp : S3 -> S4;$
(b)



(c)

```
1  // Inline Monitoring Attribute Grammar
2
3  StateMachine ::= Declaration*; History Connect
4  (Declaration, History, Connect).env=StateMachine.env
5  Declaration.rhis =ref to History.His
6
7  Connect ::= \epsilon // (p_{sp_{0}})
8      if(Connect.env=1){
9      Connect.InstrumStr=
10     ``if !(ConnectToProtectedPart())
11      then print(``Error");"
12   Instbool=Instrument {
13     Connect by Connect.InstrumStr
14     through Connect ::= IfStmt}
15     }
16
17     if !Instbool then  Error
18
19     History ::= epsilon
20     if(History.env=1)
21     History.His=InitSetOfSus
22
23     Connect ::= IfStmt //(p_{IS_{1}})
24      Stmts.env=0
25
26 abstract Declaration;
27
28 State : Declaration ::= <Label:String>;
29 surec=(Label, 0)
30 AddSuHis(surec, Declaration.rhis)
```

Fig. 1. p_{emp}-IMAG: The state machine inline attribute grammar to automatically enforce p_{emp} policy (part 1).

```
31
32  Transition : Declaration ::=
33  <Label:String> <SourceLabel:String> <TargetLabel:String>; Inst
34  AddReachTrgSuHis(TargetLabel, Declaration.rhis)
35  Inst.Ihis =Declaration.rhis
36  Inst.SourceLabel=SourceLabel
37
38  Inst ::= <epsilon>
39    Inst.InstrumStr:= "if ("
40    sizeof(Inst.SourceLabel,Inst.Ihis)|
41    ">1)  then  if ("|
42    Inst.SourceLabel|
43    "!=VE"|
44    ") Printerror();"
45
46    Instbool=Instrument{
47     Inst by Inst.InstrumStr
48        through Inst ::= IfStmt}
49
50      if !Instbool then Error
51
52  Inst ::= IfStmt
```

Fig. 2. p_{emp}-IMAG: The state machine inline attribute grammar to automatically enforce p_{emp} policy (part 2).

inherited attribute of the start symbol. It is supplied as an initial value before attribute evaluation begins to turn the program monitoring on and off.

Status records here are in the form of $surec = (Label, reachablefrom)$ in which for each program state, $Label$ contains the state label and $reachablefrom$ stores the number of incoming edges to that state. The semantic rule in the state production defines the status. Moreover, in the state production, function $AddSuHis$ adds the status to the history attribute. The production rule related to Transition is extended by an ISN $Inst$ that will inherit the source label of the transition to inline the security code when it is necessary. Moreover, in the Transition production, function $AddReachTrgSuHis(TargetLabel, Declaration.rhis)$ increases the $surec.reachablefrom$ by one when $surec.label$ is the target of the transition. In the production rules of $Inst$, to check the policy p_{emp}, $sizeof(Inst.SourceLabel, Inst.Ihis)$ is used that returns $surec.reachablefrom$ when the $surec.label$ is $Inst.SourceLabel$.

ABS Grammar

In the JastAdd notation, we define the abstract grammar of p_{emp}-IMAG, explain the APIs to build the AST, and provide the aspects to define and access attributes, respectively. The abstract grammar of p_{emp}-IMAG is shown in Fig. 3 in which the non-terminals and productions are classes, and alternative productions are subclasses. The terminals are corresponding to the fields of classes. In JastAdd non-terminal attributes (NTAs) are shown like /C/. For simplicity, we use this notation for instrumentation semantic non-terminal ISN in the abstract grammar. So, A ::= B /C/ entails that A has two children: B, and C while C is an ISN that will be created by the parser, but must be extended later at attribution.

API and Attributes in Aspects

Fig. 4 and Fig. 5 shows some parts of the generated API for the state machine language, including the attributes. In the first part of the API shown in Fig. 4, ISN is defined as an abstract class and both $Connect$ and $Inst$ in the second part, Fig. 5, have extended that. The explanation of the API is available in JastAdd documents such as [3].

Due to intertype declarations in JastAdd, the attributes and semantic rules of p_{emp}-IMAG are defined in aspects such as instrumentation.jrag in [4] that we briefly explain using Fig. 6 and 7. Here we define different attributes with a simple assignment or by defining a method. These attribute declarations in this aspect file belong to the related AST classes, and JastAdd reads this aspect files and inserts the inter-type declarations into the AST classes.

In Fig. 8, we show a program that makes the state machine shown in Fig. **??** with directly used API construction that manually constructs the AST. The process completes without any error, which entails satisfaction of the required policy in the state machine.

```
// Abstract syntax
// See section 2.1 in the JastAdd tutorial paper
// Reference manual: http://jastadd.org/

StateMachine ::= Declaration*; History /Connect/

History ::= epsilon

Connect  ::= epsilon // (p_{sp_{1}})
Connect  ::= Stmts    //(p_{IS_{1}})

abstract Declaration;

State : Declaration ::= <Label:String>;

Transition : Declaration ::=
<Label:String> <SourceLabel:String> <TargetLabel:String>; /Inst/

Inst ::= epsilon          // (p_{sp_{2}})
Inst ::= ifStmt            //(p_{IS_{2}})
```

Fig. 3. JastAdd abstract grammar for p_{emp}-IMAG

REFERENCES

[1] G. Hedin and E. Magnusson, "Jastadd-an aspect-oriented compiler construction system," in *Science of Computer Programming*, 2003, pp. 37–58.

[2] G. Hedin, "An introductory tutorial on jastadd attribute grammars," in *Generative and Transformational Techniques in Software Engineering III*. Springer, 2011, pp. 166–200.

[3] "Welcome to jastadd.org," http://jastadd.org/.

[4] "Example of imag in jastadd," 2016, https://github.uio.no/toktamr/IMAGJA.

[5] T. Ramezanifarkhani and M. Razzazi, "Control flow integrity via data flow integrity specification and dynamic enforcement," *Technical report, Department of Informatics, University of Oslo, Norway*, vol. 31, p. Nr 481.

```
class StateMachine  {
  StateMachine();                          // AST construction
  void addDeclaration(Declaration node);   // AST construction
  void addHistory(History node);           // AST construction
  History getHistory();                    // AST traversal
  void addConnect(Connect node);           // AST construction
  Connect getConnect();                    // AST traversal
  List<Declaration> getDeclarations();     // AST traversal
  Declaration getDeclaration(int i);       // AST traversal
  boolean env();                           // Attribute access
}
abstract class Declaration extends ASTNode {
  set<status> rhis ();                     // Attribute access
}
class History extends ASTNode {
}
class Stmt extends ASTNode { ...
}
abstract class ISN extends ASTNode {
abstract String getInstrumStr();               // AST traversal
abstract void setInstrumStr(string instrumStr); // AST traversal
abstract void addInstrum(ASTNode node);        // AST construction
abstract ASTNode getInstrum();                 // AST traversal
}
class Connect extends ISN {\
  Connect();                                  // AST construction
  String getInstrumStr();                     // AST traversal
  viod setInstrumStr(string instrumStr);      // AST traversal
  void addInstrum(ASTNode node);              // AST construction
  ASTNode getInstrum();                       // AST traversal
  void addStmt(Declaration node);             // AST construction
  List<Stmt> getStmts();                      // AST traversal
  Stmt getStmt(int i);                        // AST traversal
}
class State extends Declaration {
  State(String theLabel);         // AST construction
  String getLabel();              // AST traversal
  Set<Transition> transitions(); // Attribute access state has an attribute
  //called transitions of type of the set of transitions
  Set<State> reachable();        // Attribute access state has an attribute
  //called reachable of type of the set of State
  }
```

Fig. 4. Parts of the generated API regarding the state machine language (part 1)

```
43  class Transition extends Declaration {
44    Transition(String theLabel, theSourceLabel, theTargetLabel);
45                                // AST construction
46    String getLabel();          // AST traversal
47    String getSourceLabel();    // AST traversal
48    String getTargetLabel();    // AST traversal
49    State target();             // Attribute access
50    State source();             // Attribute access
51    void addInst(Inst node);    // AST construction
52    Inst getInst();             // AST traversal
53  }
54  class Inst extends ISN {
55    Inst(); // AST construction
56    String getInstrumStr();                 // AST traversal
57    viod setInstrumStr(string instrumStr);  // AST traversal
58    void addInstrum(ASTNode node);          // AST construction
59    ASTNode getInstrum();                   // AST traversal
60    void addIfStmt(IfStmt node);            // AST construction
61    List<Stmt> getStmts();                  // AST traversal
62    Stmt getStmt(int i);                    // AST traversal
63    }
64  class status{
65    string Label;
66    int ReachableFrom;
67    public status();
68    // Constructor, returns a new pair (null,0);
69    public void SetSu(string label);
70    // Adds the element su to this object.
71    public void IncSuRF(string label);
72    // Increments the ReachableFrom of this object.
73    public String getLabel();
74    // Returns label of this object.
75    public int getReachableFrom();
76    // Returns ReachableFrom of this object.
77  }
78  class statusSet<status> implements Set{
79    public statusSet();
80    // Constructor, returns a new empty set.
81    public void addSu(status su);
82    // Adds the element su to this object.
83    public int ReachableFrom(string label);
84    // Returns ReachableFrom if this set contains an status with the input label.
85  }
```

Fig. 5. Parts of the generated API regarding the state machine language (part 2)

```
aspect Instrumentation {
inh boolean StateMachine.env()=getFlagInput(boolean InstFlag);
//in-line equation

inh boolean StateMachine.getDeclaration(int i).env()=env();
//the right hand side of the equation is within the scope of StateMachine and
//thus it means StateMachine.env()

inh boolean StateMachine.getHistory.env()=env();

Syn statusSet History.His()=new statusSet(); //a synthesized %write-access
protected-live attribute which is a set of statuses.

inh StateMachine.getDeclaration(int i).rhis() = History.His()
//a reference to the his attribute

inh boolean StateMachine.getConnect.env()=env();


Syn string Connect.setInstrumStr(
"if !(ConnectToProtectedPart())
 then printError();"); // sets the Connect.InstrumStr by the input string

Connect.Instrument ()
{
rewrite Connect  { // Conditional rewrite rule
  when (Connect.env())
    to IfStmt {
      ...    // extending IfStmt  based on Connect.setInstrumStr
      return Instbool;
    }
```

Fig. 6. A short version of the aspect that instruments the code (part 1).

```
If !Instbool then println("Instrumentation Error");
}

Syn boolean Stmts.env()=0;

Syn status state.surec()=(SetSu(getLabel()),0);
Declaration.rhis.AddSu(state.surec);

Declaration.rhis.IncSuRF(Transition.lookup(getTargetLabel()));
inh boolean Inst.env()=Transition.env();

Inh statusSet Inst.Ihis=Declaration.rhis();
Inh string Inst.SourceLabel=Transition.lookup(getTargetLabel());
Inh string Inst.Label=Transition.lookup(getLabel());

Syn string Inst.setInstrumStr(
"if ("+Inst.Ihis. ReachableFrom(SourceLabel)+">1)  then  if ("+
  Inst.Label+ "!=VE"+ " then printError();");

Inst.Instrument ()
{
rewrite Inst{ // Conditional rewrite rule
  when (Inst.env())
    to IfStmt {
      ...    // extending IfStmt  based on Inst.setInstrumStr
      return Instbool;
    }
If !Instbool then println("Instrumentation Error");
}
inh boolean ifStm.env()=Inst.env()
}
```

Fig. 7. A short version of the aspect that instruments the code (part 2).

```
package exampleprogs;
import AST.*;

public class MainProgram {

    public static void main(String[] args) {
        // Construct the AST
        StateMachine m = new StateMachine(1);
        m.addDeclaration(new State("S0"));
        m.addDeclaration(new State("S1"));
        m.addDeclaration(new State("S2"));
        m.addDeclaration(new Transition("auth=x", "S0", "S1"));
        m.addDeclaration(new Transition("auth=y", "S0", "S2"));
        m.addDeclaration(new State("S3"));
        m.addDeclaration(new State("S4"));
        m.addDeclaration(new Transition("permitted", "S1", "S3"));
        m.addDeclaration(new Transition("denied", "S2", "S3"));
        m.addDeclaration(new Transition("VE", "S3", "S4"));

        // instrument the program and check the policy.
                m.Instrumentation();

    }

}

```

Fig. 8. A program with a state machine that satisfies the p_{emp} policy in p_{emp}-IMAG