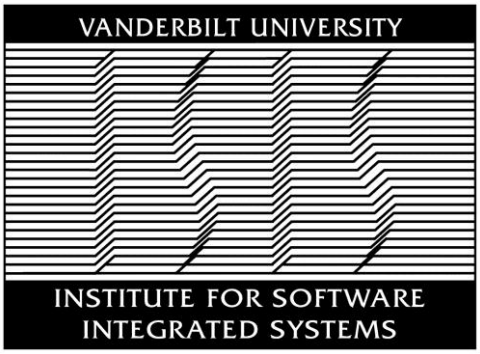


Institute for Software-Integrated Systems

Technical Report

TR#: **ISIS-15-118**

Title: **Applying Decentralized Information Flow Labels to Component-Based Software Systems Deployment**

Authors: **David Lindecker, Janos Sztipanovits**

Applying Decentralized Information Flow Labels to Component-Based Software Systems Deployment

David Lindecker, Janos Sztipanovits

September 5, 2014

1 Introduction

Model-Based Engineering (MBE) emphasizes the use of models at all phases of system development. This approach has many advantages including reduced development time and cost as well as improved cohesiveness between development phases (i.e. progressing from one phase of development to another consists of model refinements). Furthermore, MBE shows considerable promise for addressing the immense complexity of modern embedded and Cyber-Physical systems.

Until recently, the topic of security has not been deeply considered for these systems. As these systems continue to become more complex, however, the lack of proven techniques for building in security becomes a more pressing concern. Simultaneously, the context of the increasing popularity of MBE and associated tools presents an opportunity for incorporating security concerns in an intuitive way.

One important aspect of security is controlling the flow of information throughout the system. In [4], Myers and Liskov introduce a model for decentralized information flow control. We have integrated this model into our embedded software modeling framework in order to provide a validation basis for the information flows in these systems. We have also created a formal framework based on FORMULA [1, 2] which allows us to perform automated analyses on models.

This paper describes in detail our FORMULA framework for decentralized information flow control for component-based software systems. Section 2 describes the principal hierarchy, which describes the relationships between the different entities important to the flow of information. The details of the policies and labels are described in Section 3 which are used to describe restrictions on the flow of information. Section 4 describes our annotation of component interaction models with information flow labels. In Section 5 we describe the annotation of hardware deployment models with information flow labels and in Section 6 we conclude.

2 Principal Hierarchy

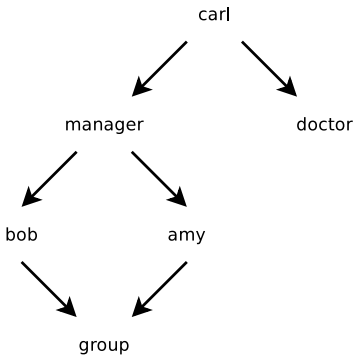
The distributed label model manages information flow restrictions in a manner that considers the expectations of multiple interested entities, or *principals*. For our purposes, a principal is an abstract concept consisting only of a name. We create the following type signature in FORMULA to denote this:

```
Principal ::= new (name:String).
```

It is useful to be able to specify that one principal can perform actions with the authority of another. In the distributed label model this is achieved with a binary relation on principals, which we call *ActsFor*. In FORMULA, we specify this binary relation in the following way:

```
ActsFor ::= new (Principal, Principal).
```

The *ActsFor* relation induces an authority hierarchy on the principals, and can be used to realize many different delegation types. Consider the diagram in Figure 1a, taken from [5]. In this diagram, the *ActsFor* relation is used to specify that the principal *carl* has two different roles: *doctor* and *manager*. As a



(a) Graph representation [5]

```
1  model M1 of PrincipalHierarchy
2  {
3    Principal("carl").
4    Principal("manager").
5    Principal("doctor").
6    Principal("amy").
7    Principal("bob").
8    Principal("group").
9    ActsFor(Principal("carl"),Principal("doctor")).
10   ActsFor(Principal("carl"),Principal("manager")).
11   ActsFor(Principal("manager"),Principal("bob")).
12   ActsFor(Principal("manager"),Principal("amy")).
13   ActsFor(Principal("bob"),Principal("group")).
14   ActsFor(Principal("amy"),Principal("group")).
15 }
```

(b) FORMULA encoding

Figure 1: A principal hierarchy

manager, Carl can perform actions on behalf of Bob or Amy. Additionally, Bob and Amy (as well as Carl by transitivity) have the authority to act as *group*. Figure 1b shows the encoding of this diagram as a FORMULA model, where the principals and relations have been translated to terms which adhere to the type signatures we have described.

The *ActsFor* relation is considered to be transitive and reflexive. Rather than requiring models to include all of the terms necessary to realize these closures, we can compute them automatically with the following logical rules:

```
ActsForT ::= (Principal, Principal).
ActsForT(x,y) :- ActsFor(x,y).
ActsForT(x,z) :- ActsForT(x,y), ActsFor(y,z).

ActsForTR ::= (Principal, Principal).
ActsForTR(x,x) :- x is Principal.
ActsForTR(x,y) :- ActsForT(x,y).
```

ActsForT is the transitive closure of *ActsFor* and *ActsForTR* is the transitive and reflexive closure. We compute the transitive closure by itself in order to facilitate our constraint that the hierarchy cannot contain a loop:

```
conforms no ActsForT(x,x).
```

3 Labels and Policies

A *policy* specifies a particular principal's expectations about the flow restrictions placed on a piece of information. It consists of an owner principal (the principal for which expectations are specified) and a list of allowed reader principals. The notation described for a policy in [4] is

$$O : r_1, r_2, ...$$

where O is the owner principal and r_n are the allowed reader principals. A policy functions as a whitelist, meaning that any principal not listed as a reader is not allowed to read information restricted by the policy. We use the following type signatures to specify this structure in FORMULA:

```
Policy   ::= new (owner:Principal,
  readers:any PrinList).
PrinList ::= {NULL} + PrinNode.
PrinNode ::= new (pr:Principal,
  rest:any PrinList).
```

The list structure is specified with a recursive type signature. The `any` keyword means that the subterm does not need to be a top-level term in the model (e.g. the term `PrinNode(pr,rest)` does not imply `rest` must be present in the model, but it does imply `pr`).

The key to specifying the expectations of multiple parties about the flow of information is the *label*, which is merely a list of policies, written as

$\{P_1; P_2; ...\}$. We encode this in FORMULA with the same list structure we used for reader lists within policies:

```
Label      ::= PolicyList.
PolicyList ::= {NULL} + PolicyNode.
PolicyNode ::= new (pl:any Policy,
  rest:any PolicyList).
```

We note that a label encoded as a FORMULA term is much more verbose and difficult to read than the standard notation. For example, the label $\{A : B, C; D : E, F\}$ would be expressed as

```
PolicyNode(Policy(Principal("A"),
PrinNode(Principal("B"),PrinNode(Principal("C"),
NULL))),PolicyNode(Policy(Principal("D"),
PrinNode(Principal("E"),PrinNode(Principal("F"),
NULL))),NULL))
```

This issue can be largely alleviated with automated translation. Currently, we have implemented an automated translation from the standard notation to the FORMULA term notation.

Additionally, it is difficult to access individual elements of these recursive list structures in FORMULA. Fortunately, FORMULA provides a feature which helps with this:

```
LabelPolicy  ::= sub (Label, Policy).
LabelSub     ::= sub (Label, Policy,
  PrinList, Principal).
```

The `sub` keyword indicates a special type of derived term for which FORMULA will automatically create the inferrence rules. In this case, we derive `LabelPolicy` terms for each combination of a `Label` term and a `Policy` term where the former is a top-level term in the model and the latter is a subterm of it. This pattern continues from left to right. For `LabelSub`, we further infer terms with readers within the reader list within the policies. We can use the term `LabelSub(lbl,pl,_,pr)` to match a reader principal `pr` within a policy `pl` within a label `lbl`.

We honor all of the policies within a label by establishing an effective readers set for it, which consists of each reader that is listed by all of the policies, or rather the intersection of all of the reader sets. We can derive this in FORMULA as follows:

```
EffectiveReader ::= (Label, Principal).
EffectiveReader(lbl,pr) :-
  lbl is Label, pr is Principal,
  no { pl | LabelPolicy(lbl,pl),
    no LabelSub(lbl,pl,_,pr) }.
EffectiveReader(NULL,pr) :- pr is Principal.
```

Furthermore, we specify that a principal "can read" data associated with a label if the principal acts for a principal which is in the effective reader set:

```
CanRead ::= (Label, Principal).
CanRead(lbl,pr) :-
  ActsForTR(pr,pr'), EffectiveReader(lbl,pr').
```

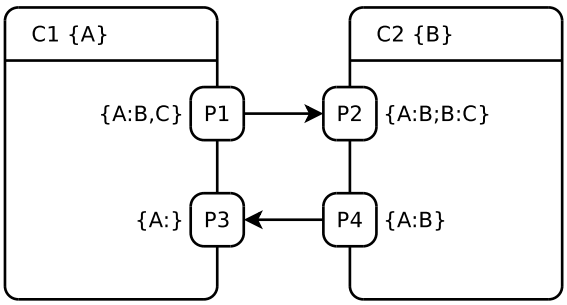
It is useful to be able to identify when one label constitutes a *declassification* of another label, in particular with respect to an individual owning principal's policy. A declassification is defined as adding a reader to a policy or removing a policy from a label and is considered an invalid information flow unless performed by a principal which can act for the owning principal of the respective policy:

```
DeclassificationOf ::= (Label, Label, Principal).
DeclassificationOf(lbl,lbl',owner) :-
  lbl is Label, lbl' is Label, owner is Principal,
  LabelPolicy(lbl,pl), LabelPolicy(lbl',pl'),
  pl.owner = owner, pl'.owner = owner,
  LabelSub(lbl,pl,_,reader),
  no LabelSub(lbl',pl',_,reader).
DeclassificationOf(lbl,lbl',owner) :-
  lbl is Label, lbl' is Label, owner is Principal,
  LabelPolicy(lbl',pl'), pl'.owner = owner,
  no { pl | LabelPolicy(lbl,pl),
    pl.owner = owner }.
```

4 Component Interaction

In this section we expand on the label model that we have encoded in FORMULA in order to reason about information flow in component-based software systems. In [4], the authors define a concept called a *slot*, which is an object that can hold a value and is tagged with a label. The value of a slot is dynamic, the label is static. When a value flows from slot A to slot B, we need to ensure that the label of slot B does not constitute an invalid declassification of the label of slot A.

We consider an abstract component interaction model, consisting of components, each with owned input and output ports, and information flow links



(a) Visualization

```
1  model M2 of Components
2  {
3    a is Principal("A"). b is Principal("B"). c is Principal("C").
4    p1_lbl is PolicyNode(Policy(a, PrinNode(b, PrinNode(c, NULL))),NULL).
5    p2_lbl is PolicyNode(Policy(a, PrinNode(b, NULL)),
6      PolicyNode(Policy(b, PrinNode(c, NULL)),NULL)).
7    p3_lbl is PolicyNode(Policy(a, PrinNode(b, NULL)),NULL).
8    p4_lbl is PolicyNode(Policy(a, NULL),NULL).
9    p1 is Port("P1", p1_lbl). p2 is Port("P2", p2_lbl).
10   p3 is Port("P3", p3_lbl). p4 is Port("P4", p4_lbl).
11   c1 is Component("C1", a, PortNode(p3,NULL), PortNode(p1,NULL)).
12   c2 is Component("C2", b, PortNode(p2,NULL), PortNode(p4,NULL)).
13   Link(p1,p2). Link(p4,p3).
14 }
```

(b) FORMULA model

Figure 2: A labeled component diagram

between these ports. We extend this model to support the decentralized label framework in the following ways: i) we unify the port and slot concepts such that each port is tagged with a label, and ii) we specify an owning principal for each component. The FORMULA type signatures for this are given here:

```
Component ::= fun (id:String -> owner:Principal,
  inputs:any PortList, outputs:any PortList).

PortList  ::= {NULL} + PortNode.
PortNode  ::= new (port:Port, rest:any PortList).

Port      ::= fun (id:String -> lbl:Label).
Link      ::= new (src:Port, dst:Port).
```

A simple example is shown both visually and as a FORMULA model in Figure 2, which contains two components, C1 and C2, owned by principals A and B respectively. C1 has an output port, P1, and an input port, P3. Similarly, C2 has an output port, P4, and an input port, P2. There are information flow links from port P1 to port P2 and from port P4 to port P3. In the visualization of this model, the labels that each port has been tagged with have been placed to the side of the corresponding ports.

Primarily, we are concerned with ensuring that the information flow links are valid. We differentiate between two cases: i) external links between components, and ii) internal links within components. We identify these cases with the following:

```
IntLink ::= (Port, Port, Component).
IntLink(src,dst,cmp) :-
  ComponentInput(cmp,src),
  ComponentOutput(cmp,dst),
  Link(src,dst).

ExtLink ::= (Port, Port).
ExtLink(src,dst) :-
```

```
Link(src,dst), no IntLink(src,dst,_).
```

For external links, we require that the destination's label is at least as restrictive as the source's label, or rather that the destination's label is not a declassification of the source's label:

```
conforms no { ExtLink(src,dst),
  DeclassificationOf(dst.lbl,src.lbl,_) }.
```

For internal links, we allow the exception where the destination's label is declassified with respect to a policy owned by a principal that the owner of the respective component acts for:

```
conforms no { IntLink(src,dst,cmp),
  DeclassificationOf(dst.lbl,src.lbl,pr),
  no ActsForTR(cmp.owner,pr) }.
```

Consider the model in Figure 2, which contains two external links. Both links are valid as the destination label is more restrictive than the source. Intuitively, the link from P1 to P2 removes a reader from the policy owned by A and adds a new policy. The link from P4 to P3 only removes a reader from the policy owned by A. These relabelings represent increased restrictiveness and are therefore valid. We could further imagine internal links from P2 to P4 and from P3 to P1. The link from P2 to P4 removes the policy owned by B. This would be an error for an external link, but since B owns the component, it is valid. The same is true of the link from P3 to P1 since readers are added to the policy owned by A, which is also the owner of the component.

5 Hardware Deployment

In addition to ensuring that logical component interaction models adhere to certain restrictions on information flow, we would like to ensure that when we deploy these systems to distributed hardware nodes, the information flow restrictions are not invalidated by the properties of the communication channels used for realizing component interaction. The deployment model consists of processing nodes and channels, as well as mappings from the component model's components to nodes and links to channels. The security properties of communication channels are represented by tagging them with a label in the same was as component ports. This label should be determined by the designer to indicate which principals could potentially eavesdrop on the information as it is communicated across the channel. The FORMULA type signatures are as follows:

```
Channel  ::= fun (id:String -> lbl:any Label).
Node     ::= fun (id:String ->
  chans:any ChanList).

ChanList ::= {NULL} + ChanNode.
ChanNode ::= new (chan:Channel,
  rest:any ChanList).

CompMap ::= fun (src:Component -> dst:Node).
LinkMap ::= fun (src:Link -> dst:Channel).

NodeChannel ::= sub (Node, Channel).
```

We include some restrictions on meaningless link mappings. Naturally, we want to prevent mapping a link between components executing on the same node to a channel as these components can communicate within the node using some sort of interprocess communication:

```
BadLinkMap :-
  LinkMap(Link(p1,p2),_),
  CompOutput(c1,p1), CompInput(c2,p2),
  CompMap(c1,n), CompMap(c2,n).
```

Additionally, we want to ensure that when a link is mapped to a channel, the nodes which the corresponding components are mapped to have access to the channel:

```
BadLinkMap :-
  LinkMap(Link(_,pt),ch), CompInput(cmp,pt),
  no { n | CompMap(cmp,n), NodeChannel(n,ch).
BadLinkMap :-
  LinkMap(Link(pt,_),ch), CompOutput(cmp,pt),
  no { n | CompMap(cmp,n), NodeChannel(n,ch).
```

Finally, we need to ensure that when we map a link to a channel, the information flow restrictions placed on the source of the link are not invalidated by the channel's security properties. Since information read while crossing a channel is considered to exit the system, we only need to check that the effective reader set of the channel's label is a subset of the effective reader set of the link source:

```
ChannelLeak :-
  LinkMap(Link(src,_),ch),
  EffectiveReader(chan.lbl,pr),
```

```
no EffectiveReader(src.lbl,ch).
```

6 Conclusion

In this paper we have documented our FORMULA-based framework for modeling and analyzing information flows in a component-based system. This work builds on an established information flow model, presented in [4], by including abstractions for component interaction models and their hardware deployment as well by providing a FORMULA-based implementation which supports automation of several analysis tasks.

Within our framework, we have assumed that information flows within a component can be reduced to links directly from input ports to output ports, however, we do not consider here the topic of identifying which links are required. If the system designer has control over the implementation of components, he can analyze this implementation to determine the information flows. It may also make sense to verify separately that each component's implementation adheres to the information flow restrictions of its interface, then verify that the interconnections are sound.

However, we would like to be able to address scenarios where the designer does not have control of some or even any of the component implementations. We can perform a "worst-case" information flow analysis by simply inferring links from all input ports to all output ports on these blackbox components and checking that these links are valid, but this is likely too restrictive to be useful in many cases. Another possibility is to implement a trusted platform for these components to execute on which enforces that components obey the information flow restrictions imposed by their interfaces. This is similar to what is done for the Jif language in [5].

In this paper, we have described a framework which would ideally serve as the backend for a development environment. We have previously used FORMULA as a backend for formalizing the structural and behavioral semantics of modeling languages, as described in [6, 7, 8]. For these previous works, we established a translation from the Generic Modeling Environment (GME) [3] to FORMULA. In this case, we will also likely want to automate importing elements from FORMULA into GME, because we plan to use FORMULA's SMT solver integration features to synthesize correct hardware deployments.

Another topic for further work on this framework is the issue of integrity, which the authors in [4] also mention as a possible extension. The idea is that labels would also include policies which list allowed writer principals. The application context could be where certain principals have the ability to intercept and alter communications on some channels and we want to ensure at certain points within the system that information has not been tampered with by a particular principal.

References

[1] Ethan Jackson and Janos Sztipanovits. Formalizing the structural semantics of domain-specific modeling languages. *Software & Systems Modeling*, 8(4):451–478, 2009.

[2] Ethan K Jackson, Wolfram Schulte, and Nikolaj Bjørner. Detecting specification errors in declarative languages with constraints. In *Model Driven Engineering Languages and Systems*, pages 399–414. Springer, 2012.

[3] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[4] Andrew C Myers and Barbara Liskov. *A decentralized model for information flow control*, volume 31. ACM, 1997.

[5] Andrew C Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[6] Gabor Simko, Tihamer Levendovszky, Sandeep Neema, Ethan Jackson, Ted Bapty, Joseph Porter, and Janos Sztipanovits. Foundation for

model integration: Semantic backplane. In *Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE*, pages 12–15, 2012.

[7] Gabor Simko, David Lindecker, Tihamer Levendovszky, Ethan K Jackson, Sandeep Neema, and Janos Sztipanovits. A framework for unambiguous and extensible specification of dsmls for cyber-physical systems. In *IEEE 20th International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)*, pages 30–39, 2013.

[8] Gabor Simko, David Lindecker, Tihamer Levendovszky, Sandeep Neema, and Janos Sztipanovits. Specification of cyber-physical components with formal semantics – integration and composition. In *ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2013.