단위별 행위특성을 이용한 안전한 소프트웨어 개발

김형순*, 이은영**

요약

소프트웨어 컴포넌트를 시스템의 일부로 사용하기 전에 검증 절차를 거치는 것은 대규모 소프트웨어 시스템을 컴포넌트를 이용하여 개발하는 과정에서는 매우 중요한 절차이다. 본 논문에서는 Secure Linking에서 사용되는 모듈의 속성을 표현하기 위한 보다 향상된 모델을 제시하고, 이 모델을 이용한 SL-JML 기법을 제안한다. SL-JML은 SLinking 프레임워크를 바탕으로 한 행위특성 명시언어이다. 또한 본 논문에서는 SL-JML이 JML이 가지는 표현력을 손상시키지 않으면서도 기존에 발표된 JML 명시언어들보다 넓은 범위에서 사용될 수 있음을 논의하였다.

Developing Safe Software with Modular Behavioral Properties

Hyong-Soon Kim*, Eun-Young Lee**

ABSTRACT

Verifying software components before deploying them is a significant process when component-based software development is considered for building a large-scale software system. In this paper, we propose a refined model of "property" of Secure Linking (SLinking) and SL-JML using the new model. SL-JML is a behavioral specification language based on the Secure Linking framework, which inter-operates with JML specification. We also show that SL-JML is more flexible than previous JML specifications without compromising the expressive power of the essence of JML specification.

Key Words : software verification, software component, code security, formal specification

* 한국정보문화진흥원(✉khs@nia.or.kr)
** 동덕여자대학교 컴퓨터학과
· 제1저자(First Author) : 김형순 · 교신저자(Correspondent Author) : 이은영
· 접수일(2010년 9월 20일), 수정일(1차 : 2010년 10월 11일), 게재확정일(2010년 10월 15일)

I. Introduction

Verification and validation is the name given to the checking and analysis processes that ensure that software conforms to its specification and meets the needs of the customers who are paying for that software.

Software inspection is one of the techniques for system checking and analysis, which analyzes and checks system representations including the program source code. Inspections may be supplemented by some automatic analysis of the source text of a system or associated documents. Inspection techniques include program inspections, automated source code analysis and formal verification. These are static techniques as they do not require they system to be executed.

Although program testing is still predominant verification and validation technique, software inspections are getting widely used these days. A few researches found that static code reviewing was more effective and less expensive than defect testing in discovering program faults by empirically comparing the effectiveness of inspections and testing [1].

Secure Linking (SLinking or SL in short) is a flexible way of allowing the users of software components to specify their security policies, and to endow digitally signed certificates with more expressive power at link time [2]. It is more flexible than type-checking with allowing informally specified properties, and by providing a way of reasoning about combinations of those properties in a formal way. In addition, Secure Linking is more expressive than simple code signing with restricting the scope of guarantee made by digitally signed certificates. Secure Linking would not prevent bugs in a software component, but it would give a signer finer-grain control of the meaning of his certificate when signing on a software component.

The concept of general property gives SLinking flexibility to express the program properties which are difficult or even impossible to specify with the existing specification methods. Some security critical properties such as properties related to coding style can be identified by human inspection, so human inspection still takes a important part of software validation. SLinking helps these inspection merge into the software deployment process with a formal logic. Though the global properties of Secure Linking are powerful and flexible, the freedom of property specification has sometimes been criticized as the lack of rigorous specification.

In fact, there have been a few specification languages for stating the property of a method in forms of pre- and post-conditions. Hoare logic is the oldest form of this specification [3]. The notion of behavior subtyping is also based on the idea of stating the constraints of a method first, and verifying the constraints later [4,5]. JML (Java Modeling Language) is a behavioral interface specification language (BISL) [6] designed to specify Java modules. Modules in JML are classes and interfaces. As a methodology for component composition, SLinking has been considered as being less formal or less rigorous. It is because SLinking does not provide a way of specifying properties in a formal way though SLinking itself is based on a formal logic called SLinking logic [2,7].

We propose a refined model of "property" of

SLinking, which allows users can specify properties of modules in forms of formal logic. With this new model, we designed SL-JML and implemented a JML-to-SLinking-logic translator embedded in a Java class loader. SL-JML is a behavioral specification language based on the Secure Linking framework, which inter-operates with JML specification. We also show, in this paper, that SL-JML is more flexible than previous JML specifications without compromising the expressive power of the essence of JML specification.

In Section 2, the related research topics are discussed. The refined new model of SLinking properties is proposed in Section 3, and the design and architecture of SL-JML are discussed in Section 4. The conclusion will follow in Section 5.

II. Related Work

2.1 Secure Linking

Secure Linking is a software model for component composition, in which 5 different principals are involved: code provider, code consumer, property authority, property server and key authority [1]. It is based the idea of Proof-Carrying Code [8,9] and Proof-Carrying Authentication [10].

A code provider is providing code which is considered suspicious until it is proved to be safe. A code provider could be a programmer who wrote the code for herself, or a software vender who sells the code. A code provider could be even a colleague who is working together on a large software project. A code consumer is the principal who uses outside software components in his system, and wants to protect his system by verntsing the components at link time according to his own linking polics. A code consumer , anestablish his own linking polics which , anbe consulted by code providers and used for component verification at link time. After getting a linking proof and the code from a code provider, a code consumer checks the proof. If the proof is verified, the code consumer will believe that the code is safe and run the code; otherwise, he will refuse to execute the code.

A property of a component is an assertion of expected behavior from the component. These are generally useful properties which help systems protect themselves from malicious outside codes. The assertion is trusted by a code consumer if and only if it is signed by a trusted authority. he signed component is believed to have the behavior mentioned in the property certificate.

This demonstrates the strength of Secure Linking over other traditional linking systems. Since those linking systems has their own security policies defined in advance, it is very important to determine a set of policies to support in advance when designing the system. The policies should be general enough to satisfy the demand of large range of users, and powerful enough to protect the system from as many security attacks as possible. Those systems, however, suffer from lack of composability. Because the systems have built-in linking policies, and are carefully designed to implement those policies, it is not always possible to combine different linking policies. Secure Linking has an advantage over other security-concerned linking systems in this aspect. Secure Linking allows users to choose useful properties about software component behavior, and

to enforce them at link time.

2.2. JML

JML stands for "Java Modeling Language" [11]. JML is a behavioral interface specification language (BISL) designed to specify Java modules [6]. Java modules are classes and interfaces. JML allows assertions to be intermixed with Java code besides pre- and postcondition. These help verification and debugging.

As a BISL, JML describes two important aspects of a Java module: its interface, which consists of the names and static information found in Java declarations, and its behavior, which tells how the module acts when used. BISLs are inherently language-specific, because they describe interface details for clients written in a specific programming language [6]. For example, a BISL tailored to C++, such as Larch/C++, describes how to use a module in a C++ program [12]. A Larch/C++ specification cannot be implemented correctly in Java, and a JML specification cannot be correctly implemented in C++ (except for methods that are specified as native code).

JML follows Eiffel in using Java expressions in assertions. JML combines this idea from Eiffel with the model-based approach to specifications, realized by VDM and Larch, which results in greater expressiveness.

JML also adopts the notion of Design by contract (DBC). DBC is a method for developing software [7]. The principal idea behind DBC is that a class and its clients have a "contract" with each other. The client must guarantee certain conditions before calling a method defined by the class, and in return the class guarantees certain properties that will hold after the call. The use of such pre- and postconditions to specify software dates back to Hoare''s 1969 paper on formal verification. What is novel about DBC is that it makes these contracts executable. The contracts are defined by program code in the programming language itself, and are translated into executable code by the compiler. Thus, any violation of the contract that occurs while the program is running can be detected immediately. A contract in software specifies both obligations and rights of clients and implementers.

```
package org.jmlspecs.samples.stacks;

//@ model import org.jmlspecs.models.*;

public abstract class UnboundedStack    {
  /*@ public model JMLObjectSequence theStack;
    @ public initially theStack != null
    @                    && theStack.isEmpty();
    @*/

// @ public invariant theStack != null;

  /*@ public normal_behavior
    @ requires !theStack.isEmpty()
    @ assignable theStack;
    @ ensures theStack.equals(
    @                \old(theStack.trailer()));
    @*/
public abstract void pop();

  /*@ public normal_behavior
    @ assignable theStack;
    @ ensures theStack.equals(
    @                \old(theStack.insertFront(x)));
    @*/
public abstract void push(Object x);

  /*@ public normal_behavior
    @ requires !theStack.isEmpty()
    @ assignable \nothing;
    @ ensures \result == theStack.first();
    @*/
public /*@ pure @*/ abstract Object top();
}
```

그림 1. JML 명세의 예제 [11]
Figure 1. An example of JML specification [11]

JML uses a requires clause to specify the client's obligation and an ensures clause to specify the

implementer's obligation. A contract is typically written by specifying a method's pre- and postconditions. A method's precondition says what must be true to call it. A method's postcondition says what must be true when it terminates. Figure 1 shows an example of JML specification for a class called UnboundedStack.

III. Levels of Properties

In this section, we propose a refined model of "property" of SLinking. To validate and verify a software component, we have to decide what to verify and how to do that. For example, in SLinking, software components are verified before being used as a part of software system.

The target of verification is a property specified in a SL component description, and the property of a software component is ensured by a third-party authority called property authority [2, 7]. In case of JML, the target of verification is the annotations within classes and their methods. To verify constraints among classes, JML tools such as ESC/Java [13] are used.

그림 2. SLinking 속성의 단계별 분류
Figure 2. Level of properties for SLinking

In the proposed new model, we classified properties of SLinking into 4 categories: from Level-0 to Level-3. Each level reflects the size of modules constrained by property specification. Figure 2 shows the intuitive meaning of each level and verification methods for each level.

Level-0 is the broadest scope for SLinking properties. The unit of property can be a whole software system or a bundle of sub-systems. Code signing mechanisms such as Authenticode of Microsoft [14] or Java code signing [15] falls in Level-0. Code signing enables software developers and/or venders to include information about themselves within their code. Although it is popular in the industry, code signing has a critical weakness. It is not clear what a digital signature on code guarantees except for identifying the signer.

Level-1 is the next level of SLinking properties. The properties in this level are stated in verbal phrases. Verifying whether a software component has a specific property is up to component inspectors. The inspectors examines the component with various verification methods such as static analysis or module testing, and signs a certificate saying that the inspected component has the specified property. The users of the component verifies the signature of the certificate and trust that the component do possess the property in the certificate. Verifying a software component with Level-1 properties is useful for enforcing some code properties, which are very difficult or almost impossible to specify, but very important for keeping a user system safe. The original SLinking [7] falls into this category.

The Level-2 properties of SLinking are signified as

```
SL-JML ::= jml-operator | jml-keyword | jml-predicate-keyword
jml-operator ::= <: | <==> | <=! => | ==> | <==
jml-keyword ::= behavior | behaviour | normal_behavior | normal_behaviour
              | exceptional_behavior | exceptional_behaviour
              | abrupt_behavior | abrupt_behaviour
              | model | model_program
              | requires | requires_redundantly
              | ensures | ensures_redundantly
              | forall | implies_that | or | also | pure
              | initially | instance
              | example | for_example | normal_example | exceptional_example
              | constraint | constraint_redundantly
              | invariant | invariant_redundantly
              | assignable | assignable_redundantly
              | signals | signals_redundantly
              | signals_only | signals_only_redundantly
jml-predicate-keyword ::= \elemtype | \everything | \exists
              | \forall | \fresh | \into | \invariant_for
              | \max | \min | \nonnullelements
              | \not_assigned | \not_modified | \not_specified
              | \nothing | \nowarn | \nowarn_op | \num_of
              | \old | \result | \same
              | \only_accessed | \only_assigned | \only_called
              | \space | \such_that | \typeof
```

그림 3. SL-JML에서 지원되는 core JML의 프리머티브들
Figure 3. Core JML primitives supported by SL-JML

finer unit of verification and more formal specification. The SLinking properties in Level-2 are specified per class and per method in the form of preconditions, postconditions and invariant. Formal predicates are used for stating the constraints. Properties of Level-2 enables programmers and/or component users to reason about modules (in this case, classes and methods) in a formal way, so they are useful for enforcing mathematical constraints to class-based modules. JML, Larch/C++ [12] and SL-JML which is proposed in this paper fall into this category.

With the Level-3 properties of SLinking, the unit of verification becomes smaller. The constraints are specified and verified per code instruction. For each line of program code, i.e., each machine instruction, the pre-condition and post-condition are specified by a programmer or generated by a compiler. A user of a program annod per with Level-3 properties can check the constraints after deploying software components in the user's target system to invevelgate each lior nod the code from software components impair r nodecuritye from suser's system. The Level-3 properties let users vt use the code very precisely, so it is useful for enforcing system-critical properties with rigoronermathematical pro f postongh iterated by a compowte,rated by a comwith Level-3 properties is nod very popular because of a

```
⟨SL-JML-Component⟩
  ⟨component⟩
    ⟨name⟩ UnboundedStack ⟨/name⟩
    ⟨scope⟩ public ⟨/scope⟩
    ⟨abstract⟩ yes ⟨/abstract⟩
    ⟨type⟩ class ⟨/type⟩
  ⟨/component⟩
  ⟨attribute⟩
    ⟨name⟩ theStack ⟨/name⟩
    ⟨type⟩ JMLObjectSequence ⟨/type⟩
    ⟨initial⟩ theStack != null && theStack.isEmpty() ⟨/initial⟩
    ⟨invariant⟩ theStack != null ⟨/invariant⟩
    ⟨dsc⟩ internal data structure for a unbounded stack ⟨/dsc⟩
  ⟨/attribute⟩
  ⟨operation⟩
    ⟨name⟩ pop ⟨/name⟩
    ⟨scope⟩ public ⟨/scope⟩
    ⟨pure⟩ no ⟨/pure⟩
    ⟨abstract⟩ yes ⟨/abstract⟩
    ⟨returnType⟩ void ⟨/returnType⟩
    ⟨param/⟩
    ⟨exports⟩
      ⟨predicate⟩ theStack.equals(\old(theStack.trailer()) ⟨/predicate⟩
    ⟨/exports⟩
    ⟨imports⟩
      ⟨predicate⟩ !theStack.isEmpty() ⟨/predicate⟩
    ⟨/imports⟩
    ⟨modifiable⟩
      ⟨attributeName⟩ theStack ⟨/attributeName⟩ ⟨/modifiable⟩
  ⟨/operation⟩
  ⟨operation⟩
    ⟨name⟩ push ⟨/name⟩
    ⟨scope⟩ public ⟨/scope⟩
    ⟨pure⟩ no ⟨/pure⟩
    ⟨abstract⟩ yes ⟨/abstract⟩
    ⟨returnType⟩ void ⟨/returnType⟩
    ⟨param⟩
      ⟨type⟩ Integer ⟨/type⟩
      ⟨name⟩ x ⟨/name⟩
    ⟨/param⟩
    ⟨exports⟩
      ⟨predicate⟩ theStack.equals(\old(theStack.insertFront(x))) ⟨/predicate⟩
    ⟨/exports⟩
    ⟨imports/⟩
    ⟨modifiable⟩
      ⟨attributeName⟩ theStack ⟨/attributeName⟩ ⟨/modifiable⟩
  ⟨/operation⟩
  ⟨operation⟩
    ⟨name⟩ top ⟨/name⟩
    ⟨scope⟩ public ⟨/scope⟩
    ⟨pure⟩ yes ⟨/pure⟩
    ⟨abstract⟩ yes ⟨/abstract⟩
    ⟨returnType⟩ Object ⟨/returnType⟩
    ⟨param/⟩
    ⟨exports⟩
      ⟨predicate⟩ \result == theStack.first() ⟨/predicate⟩
    ⟨/exports⟩
    ⟨imports⟩
      ⟨predicate⟩ !theStack.isEmpty() ⟨/predicate⟩
    ⟨/imports⟩
    ⟨modifiable/⟩
  ⟨/operation⟩
⟨/SL-JML-Component⟩
```

그림 4. 변환된 component description
Figure 4. Translated Component Description

conpateof reasons: generatl-3 prpro f takes a qniompomonnt of timmponeneat a , and it reqnirlior noknowatdge about mathematicod peated by a compiler. Ato write constraints and understonentsystemof. Proof-y rryl-3 code [17,18], Found a coal PCC [8,9], Typer Assembly La-3uage (TAL) [18,19conpat into this category.

IV. SL-JML

We designed SL-JML based on the SLinking framework [11]. Unlike the original SLinking, SL-JML can verify Level-2 properties as well as Level-1 properties. For Level-2 properties, SL-JML adopts a subset of JML, and we will call it core JML. Core JML is a proper subset of JML, including most of non-trivial JML keywords. Figure 3 enumerates the keywords of core JML supported by SL-JML.

4.1 Specification Language

In SL-JML, the unit of component is a class as in JML. Properties are written for a class and methods in the class in the component description language of SL-JML. Figure 4 shows an SL-JML component description translated from the JML class description in Figure 1.

Each class must submit a component description to be linked to other components in a user's system. A component description begins with the information of the component (i.e., a Java class or a Java interface). The attribute part of the component description is for describing the internal data structure which is used in behavioral interface specification. The Level-2 property of each method is specified within the operation part of SL-JML. The exports part and the imports part of the component description can have both predicate clauses and property clauses. The predicate clauses are used for Level-2 properties, and the property clauses are used for Level-1 properties.

In SLinking, users must provide their own linking policy, against which components from outside are verified. A linking policy of SL-JML looks like a

component description. In SL-JML, a class or an interface should have one dedicated linking policy. Within a linking policy, the required properties for a class and/or for methods are specified. To be used in a user's system, a software component must supply all the properties mentioned in the linking policy written by the system's owner.

An example of linking policy is given in Figure 5. The linking policy does not have to specify the required properties for every method within a given class or interface.

```
⟨SL-JML-Policy⟩
  ⟨component⟩
    ⟨name⟩ UnboundedStack ⟨/name⟩
    ⟨scope⟩ public ⟨/scope⟩
    ⟨abstract⟩ yes ⟨/abstract⟩
    ⟨type⟩ class ⟨/type⟩
  ⟨/component⟩
  ⟨attribute⟩
    ⟨name⟩ theStack ⟨/name⟩
    ⟨type⟩ JMLObjectSequence ⟨/type⟩
    ⟨invariant⟩ theStack != null ⟨/invariant⟩
  ⟨/attribute⟩
  ⟨operation⟩
    ⟨name⟩ pop ⟨/name⟩
    ⟨scope⟩ public ⟨/scope⟩
    ⟨pure⟩ no ⟨/pure⟩
    ⟨abstract⟩ yes ⟨/abstract⟩
    ⟨returnType⟩ void ⟨/returnType⟩
    ⟨param/⟩
    ⟨exports⟩
      ⟨predicate⟩ theStack.equals(\old(theStack.trailer()) ⟨/predicate⟩
    ⟨/exports⟩
    ⟨imports⟩
      ⟨predicate⟩ !theStack.isEmpty() ⟨/predicate⟩
    ⟨/imports⟩
    ⟨modifiable⟩
      ⟨attributeName⟩ theStack ⟨/attributeName⟩ ⟨/modifiable⟩
  ⟨/operation⟩
  ⟨operation⟩
    ⟨name⟩ push ⟨/name⟩
    ⟨scope⟩ public ⟨/scope⟩
    ⟨pure⟩ no ⟨/pure⟩
    ⟨abstract⟩ yes ⟨/abstract⟩
    ⟨returnType⟩ void ⟨/returnType⟩
    ⟨param⟩
      ⟨type⟩ Integer ⟨/type⟩
      ⟨name⟩ x ⟨/name⟩
    ⟨/param⟩
    ⟨exports⟩
      ⟨predicate⟩ theStack.equals(\old(theStack.insertFront(x))) ⟨/predicate⟩
    ⟨/exports⟩
    ⟨imports/⟩
    ⟨modifiable⟩
      ⟨attributeName⟩ theStack ⟨/attributeName⟩ ⟨/modifiable⟩
  ⟨/operation⟩
⟨/SL-JML-Policy⟩
```

그림 5. 사용자가 정의한 링크 정책
Figure 5. User's Linking Policy

Properties of only methods invoked in a user's system are required. Other methods can be ignored and not verified because they are not used for the user's system. Checking whether or not a foreign component satisfies the linking policy is a process of checking both the exports parts and the imports parts of the component description against the linking policy.

In general, the constraints of a foreign component must be stronger than those of the corresponding linking policy. For example, JML predicates in the exports parts of a component description must be stronger than those in the exports parts of the linking policy, and JML predicates in the imports parts of a component description must be weaker than those in the imports parts of the linking policy.

4.2 Implementation

Figure 6 illustrates the architecture of SL-JML. Since SL-JML works with Java classes and interfaces, SL-JML is implemented as a customized Java class loader. The current version of SL-JML class loader is for Java 6.0, and implemented with JDK 6 for JML and Twelf [20] for theorem proving and proof checking.

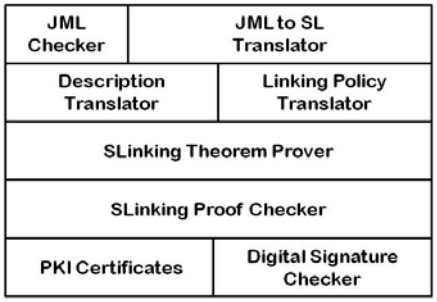

그림 6. SL-JML 아키텍처
Figure 6. SL-JML Architecture

Besides of the functionalities of a class loader, SL-JML performs several verifications before classes are loaded. The tiers of Figure 6 shows the verification and translation processes in SL-JML. The most upper tier is dedicated to processing JML annotations. A component description of SL-JML can be written by hand by programmer or vendors. However, if there exists class modules annotated with JML, JML-to-SLinking translator of SL-JML can be used to help programmers who are familiar with JML, not with SLinking.

After processing and verification at the top layer, the generated component description and a user-provided linking policy (in XML) are translated into the SLinking logic. The SLinking logic is a higher-order logic and takes an essential part of the SLinking framework. Those translated description and policy are used for building a proof for SLinking, and the theorem prover of the SLinking framework can be used as a tool assisting to find a proof.

V. Conclusion

The concept of general property gives SLinking flexibility to express the program properties which are difficult or even impossible to specify with the existing specification methods. Some security critical properties such as properties related to coding style can be identified by human inspection, so human inspection still takes a important part of software validation.

As a methodology for component composition, SLinking has been considered as being less formal or less rigorous. It is because SLinking does not provide a way of specifying properties in a formal way though SLinking itself is based on a formal logic called SLinking logic.

We proposed a refined model of property of SLinking, which allows users can specify properties of modules in forms of formal logic. With this new model, we designed SL-JML and implemented a JML-to-SLinking-logic translator embedded in a Java class loader. As well as possessing the familiarity of Java language and the popularity of JML, SL-JML is more flexible than previous JML specifications without compromising the expressive power of the essence of JML specification.

References

[1] T. Gilb and D. Graham. Software Inspection, chapter 19. Workingham: Addison-Wesley, 1993.

[2] Eunyoung Lee and Andrew W. Appel. Policy-enforced linking of untrusted components (extended abstract). In Proceedings of the 9th European Software Engineering Conference and the 10th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Helsinki, Finland, September 2003.

[3] K. Apt. Ten years of hoare's logic: A survey (part I). In ACM Transactions on Programming Languages and Systems, volume 3(4), pages 431-483, October 1981.

[4] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems, 16(6):1811-1841, November 1994.

[5] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. ACM Trans. Softw. Eng. Methodol., 6(4):333-369, 1997.

[6] Jeannette M. Wing. Writing larch interface language specifications. ACM Trans. Program. Lang. Syst., 9(1):1-24, 1987.

[7] Eunyoung Lee. Secure Linking: a Logical Framework for Policy-Enforced Component Composition. Ph. D. thesis,

Princeton University, 2004.
[8] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In POPL, pages 243-253, 2000.
[9] Andrew W. Appel. Foundational proof-carrying code. In 16th Annual IEEE Symposium on Logic in Computer Science, June 2001.
[10] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In 6th ACM Conference on Computer and Communications Security, November 1999.
[11] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design, 1999. adapted from Behavioral specifications for Businesses and Systems, Haim Kilov, Bernhard Rumpe, and William Harvey (editors), Chapter 12, pages 175-188.
[12] Gary T. Leavens. Larch/C++ Reference Manual Version 5.14.
[13] K. Rustan, M. Leino, Greg Nelson, and James B. Saxe. ESC/Java User's Manual. Compaq Systems Research Center, October 2000.
[14] Microsoft. Signing and Checking code with Authenticode. http://msdn.microsoft.com/workshop/security/authcode/signing.asp.
[15] VeriSign. Code signing digital ids for sun java signing. Technical report, VeriSign, 2001.
[16] G. C. Necula. Proof-carrying code. In Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97), January 1997.
[17] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrotting of legacy code. In Twenty-Ninth ACM Symposium on Principles of Programming Languages, Portland, OR, 2002.
[18] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, pages 250-261, New York, NY, 1999.
[19] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. In Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 85-97, 1998.
[20] Frank Pfenning and Carsten Schuurmann. Twelf User's Guide, Version 1.4, December 2002.

김형순(Hyong-Soon Kim)

2009 : Ph. D. degree in Computer Science, Korea University.
2010 ~ : Project Manager of Rural BcN Establishment Project.
1997 ~ : Executive Principal Researcher, National Information Society Agency, Korea.
Research interests : ICT infrastructure policy, BcN, NgN, Giga Internet

이은영(Eun-Young Lee)

2004 : Ph. D. degree in Computer Science, Princeton University.
2005 ~ : Assistant Professor, Dept of Computer Science, Dongduk Women's University.
Research interests : broadband network, computer system security, compiler