Securing UML Information Flow using FlowUML

Khaled Alghathbar
Information Systems Department, College of Computer and Information Sciences
King Saud University, P.O. Box 51178, Riyadh, Saudi Arabia 11543
ghathbar@ccis.ksu.edu.sa

Csilla Farkas
Information Security Laboratory, Dept. of Computer Science and Engineering,
University of South Carolina, Columbia, SC 29208
farkas@cse.sc.edu

Duminda Wijesekera
Dept. of Information and Software Engineering and CSIS,
George Mason University, MS 4A4, Fairfax, VA 22030
dwijesek@gmu.edu

FlowUML is a logic-based system to validate information flow policies at the requirements specification phase of UML based designs. It uses Horn clauses to specify information flow polices that can be checked against flow information extracted from UML sequence diagrams. FlowUML policies can be written at a coarse grain level of caller-callee relationships or at a finer level involving passed attributes.

Keywords: Unified Modelling Language, flow control, policies, logic
ACM Classification: D.2, H.4

1. INTRODUCTION

As security becomes an important aspect of large software systems, it needs to be addressed throughout the software development life cycle. Considered as non-functional, security requirements are not addressed during the early phases of system development; thus resulting in possibly vulnerable software (Chung *et al*, 2000). In Nuseibeh *et al* (2001); Nuseibeh and Easterbrook (2000); Devanbu and Stubblebine (2000) the authors present the need to formally analyze and validate security requirements during early phases of system development to detect and remove design vulnerabilities. Recent work has addressed information flow control security (Bertino and Atluri, 1999; Alghathbar and Wijesekera, 2003; Myers, 1999; Samarati *et al*, 1997). Information flow requirements need to be developed and evaluated during the requirements and design stages of the software life cycle, because validating information flow requirements at an early stage prevents costly fixes during latter stages of the development life cycle. In this paper we propose a logic-based system (FlowUML) to validate information flow policies at the requirements specification phase of UML-based designs (UML v. 1.5, 2003). FlowUML uses locally stratified Horn clauses to enforce user specifiable information flow policies via three processes: (1) Extract information flows predicates from UML sequence diagram, (2) Derive all inherited and indirect

Copyright© 2006, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: 12 April 2005
Communicating Editor: Julio Cesar Hernandez

flows, and (3) Check for compliances with specified policies – specified at two levels of granularity. At the coarse level, a flow is represented as a directed arc between components, going from its source to its sink. At the fine level, the flow is defined based on the semantics of methods, attributes passed between the components, and the roles played by the components.

FlowUML does not assume the existence of a particular meta-policy and can support different policies. In contrast to existing information flow models, FlowUML applies a formal framework to the early phases of the software development life cycle.

The rest of the paper is organized as follows. Section 2 shows information flow specifications embedded in UML sequence diagrams. Section 3 provides a running example. Section 4 describes our notations and assumptions. Section 5 explains the FlowUML process. Section 6 describes the syntax and semantics of FlowUML. Section 7 shows the utility of FlowUML by means of the running example. Section 8 describes related work and Section 9 concludes the paper.

2. FLOW SPECIFICATION IN UML

The UML is the de-facto design language for large software projects (Booch *et al*, 1999). The UML uses multiple diagrams (views) to represent requirements and designs. *Use Cases* support the specification of usage scenarios between the system and its intended users (UML v. 1.5, 2003). *Interaction diagrams* specify how objects in the use case interact with each other. *Sequence diagrams* are specific interaction diagrams that show such interactions as a sequence of messages exchanged between objects over time. The sequence diagrams shown in Figure 1 consists of four objects and two actors, where *Actor A* initiates the first flow as a request to read information from *Obj3* which then returns the requested information. Secondly, Actor A writes information to Obj4 and the control object. The flow to the *control* object triggers additional flows. FlowUML extracts flows from the sequence diagrams.

3. RUNNING EXAMPLE

Our example consists of two use cases. Use case 1 has a simple scenario shown in Figure 2, and use case 2 has a complex scenario shown in Figure 3. The actor in use case 1 reads information from object Obj1 returned as an attribute att1 and then writes it to Obj2. The actor in use case 2 transfers information between two objects and then calls another object leading to additional information flow.

During the analysis stage of the software development, selected interactions are specified between objects. These interactions are further refined by specifying the message's attributes or parameters passed between objects. Therefore, the expressiveness and detail in a sequence diagram

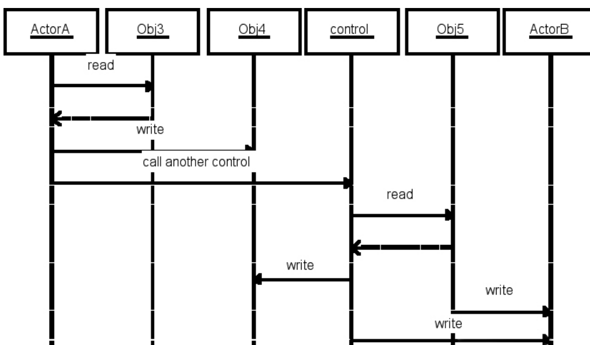


Figure 1: A sample sequence diagram

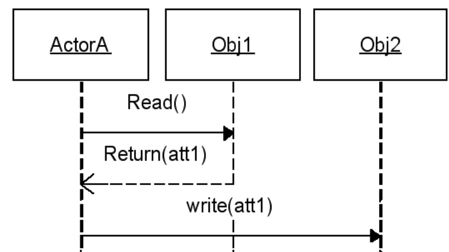


Figure 2: The sequence diagram for use case1

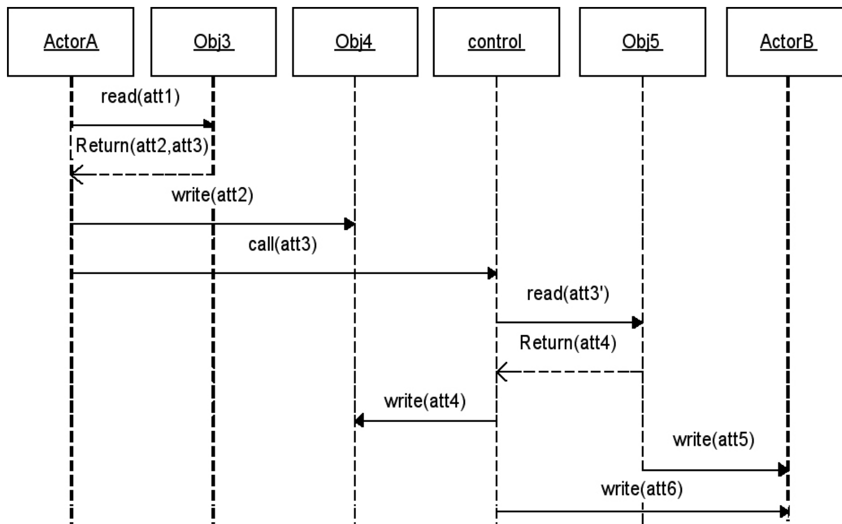


Figure 3: Detailed sequence diagram for use case 2

depends on the analysis stage. For example, the sequence diagram of use case 2 can be categorized into two views: coarse grain view shown in Figure 1 and the fine grain view shown in Figure 3.

Flow and access control requirements are expressed over the actors and objects of the sequence diagram. For example, we may specify that information is permitted to flow from Obj 3 to Obj. 4 but not vice verse. Similarly, we may permit of deny Actor A to read Obj. 3.

4. NOTATIONS AND ASSUMPTIONS

Objects in sequence diagrams belong to three categories: *actors*, *entities*, or *control objects*. Actors initiate flows and send/receive information to/from the system. Every use case has at least one actor. Entity objects are persistent and store information. Control objects provide coordination and application logic. Generally, actors and entity objects can be sources or sinks of information flows. Control objects do not store information, however, they may create new information without storing them. This is a basic axiom of FlowUML.

Axiom 1: *Sources and sinks of information flows are actors and entities.*
Information flows in sequence diagrams arise due to attributes passed in method calls such as read and write. For example, the method *read(att1)* exchanges information between *Obj3* and *Actor A*. FlowUML uses any message as a flow of information regardless of its name.

Axiom 2: *Any information-carrying message is considered an information flow.*
Sequence diagram builds complex messages from simple ones using three kinds of constructs: *creation messages* used to create new objects, *iteration messages* used to send data multiple times, and *conditional messages* used to control messages satisfying specified conditions. We consider a creation message as an information flow if the caller passes creation parameters to the callee. We consider an iterated message as a single information flow. We consider a conditional message to be an information flow regardless of the truth-value of the condition. We consider a simple message as a flow if it passes information.

4.1 Attribute Dependencies across Objects

We make the following observations about attributes:

1. Information flowing into an object may flow out unaltered, called *exact attribute flow in FlowUML*.
2. Some attributes flowing out of an object may be different in name but always have the same value as an attribute that flows into the same object. FlowUML requires this information to be in the *similar attribute table*.
3. Some attributes flowing out of an object may depend upon others that flow into the same object. FlowUML requires this information to be in the *derived attribute table*.

We formalize these observations in the following definitions and axioms.

Definition 1: *An exact attribute is one that flows through an object but does not change its value. If an attribute that flows out of an object depends upon a set of attributes flowing into the same object then we say that the output attribute is derived from the input attributes.*

Axiom 3: *Attribute names are unique over a system.*

Axiom 4: *All pairs of exact attributes are entered into the exact attribute table and all pairs of dependent attributes have entries in the derived attribute table.*

At the requirements specification stage flow analysis depends on (1) the types of components (i.e., actor, entity or control object), (2) the sequence of objects in a flow, and (3) whether the flow is within a use case or between many use cases. For example, entities or actors may alter information while control objects do not.

5. THE VERIFICATION PROCESS

FlowUML verify flow policies using five steps using two metadata sources as shown in Figure 4. Coarse grain policies require less detail than fine grain policies. Fine grain policies incorporate attribute values and their dependencies.

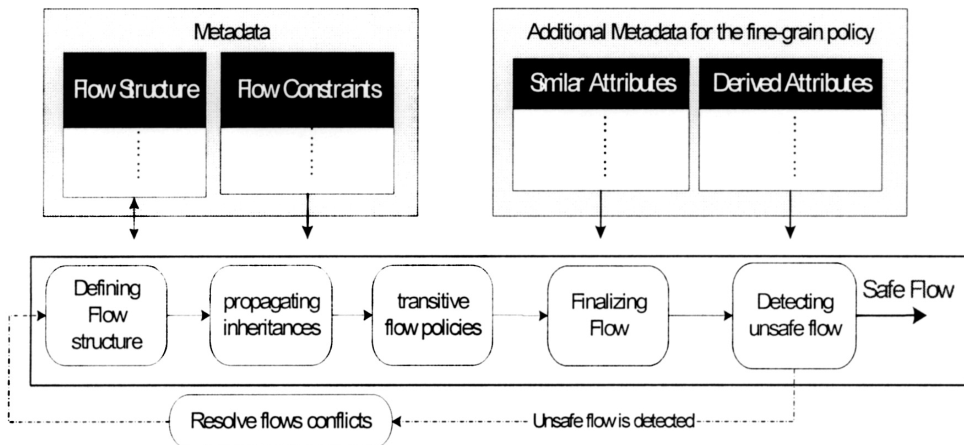


Figure 4: Steps in FlowUML

Coarse grain policy analysis

In the first step FlowUML extracts flow structure, referred to as *defining flow structure*, from the Sequence diagrams and transforms it into basic predicates (see Section 6). The second step propagates basic predicates using the actor (or role) hierarchy, deriving implied information flows. The third step derives all transitive flows. The fourth step complements the third step by filtering results of the third step to those flows that satisfy properties of interests as specified in the policies. For example, in Figure 1 the third step derives a flow from *Obj3* to *Actor A* directly, flow from *Obj3* to *Obj4* through *Actor A*, flow from *Obj3* to control through *Actor A* etc. In the fourth step, predefined policies may only apply to non-transitive flows between entities and actors. Thus only flows from *Obj3* to *Actor A* and the one from *Actor A* to *Obj4*, but not the flow from *Obj3* to *Obj4* may be of relevance. The fifth step detects flows that violate specified policies. Currently, FlowUML does not attempt to resolve the detected conflicts automatically.

Fine grain policy analysis

Fine grain policies require two kinds of additional information. The first, given in the *similar attribute table*, contains name of the attributes that always contain the same data value. The second, given in the *attribute derivation table*, lists attribute dependencies. This information is useful in controlling the flow of sensitive information. For example, in Figure 3, if there is a record in the *derived attribute table* stating that *att6* is derived from *att3* then FlowUML concludes that there is a transitive flow from *Actor A* to *Actor B*.

6. SYNTAX AND SEMANTICS

The FlowUML alphabet contains variables or constants over actors (A), objects (Obj), use cases (UC), attributes (Att), and times (T). Constants are members of the sets A, Obj, UC, Att, and T. Variables are represented as X_a, X_{obj}, X_{uc}, X_{att} and X_t, respectively. FlowUML uses the following predicates:

Basic predicates for supporting the specification of flows, called FlowSup.

1. isEntity(X_{obj}) denotes X_{obj} is an entity.
2. isActor(X_{obj}) denotes X_{obj} is an actor.
3. specializedActor(X_a,X'_a) denotes X'_a is a specialized actor of X_a.
4. precedes(X_{uc}, X'_{uc}) denotes use cases X_{uc} is executed before X'_{uc}.
5. sameAtt(X_{att}, X'_{att}) denotes attributes X_{att} and X'_{att} have the same value.
6. derAtt(X_{att}, X'_{att}, X_{obj}) denotes attribute X'_{att} is derived from X_{att} in object X_{obj}.
7. ignoreFlow(X_a, X_{obj}, X'_{obj}) says to ignore the flow from object X_{obj} to object X'_{obj}.
8. ignoreFlows(X_{a1}, X_{obj1}, X'_{obj1}, X_{a2}, X_{obj2}, X'_{obj2}) says to ignore the flow from X_{obj1} to X'_{obj1}. and the flow from X_{obj2} to X'_{obj2} from being detected as a violation of flow control policies. Predicates ignoreFlows$_{att}$(X_{a1},X_{att},,X_{obj1},X'_{obj1},X_{a2},X_{obj2},X'_{obj2}) and ignoreFlow$_{att}$(X_a, X_{att},X_{obj},X'_{obj}) are used in fine grain policies

Predicates for specifying flow constraints, called ConstSup.

1. dominates(X_{obj},X'_{obj}) syas that the security label of object X'_{obj} dominates or equals the security label of object X_{obj}, and is used in multi-level security policies.
2. ACL(X_{obj},X'_{obj},X_{AT}) says that object X_{obj} is in the access control list of object X'_{obj} – used in discretionary access control policies. X_{AT} is an action such as *read*.
3. conflictingActors(X_{obj},X'_{obj}) says that X_{obj} and X'_{obj} are conflicting actors. For example, both actors cannot flow information to the same object or there must not be a flow of information between them.

4. conflictingEntities(X_{obj},X'_{obj}) says that X_{obj} and X'_{obj} are conflicting entities. For example, each entity belongs to different competing companies.

Predicates for coarse-grain policies

1. Flow(X_a,X_{obj},X'_{obj},X_t,X_{uc}) says there is a simple flow initiated by actor X_a from object X_{obj} to object X'_{obj} at time X_t in use case X_{uc}.
2. mayFlow(X_a,X_{obj},X'_{obj},X_t,X_{uc}) is the transitive closure of the previous predicate.
3. $mayFlow_{interUC}$(X_a,X_{obj},X'_{obj},X_t,X_{uc}) is similar to mayFlow but the variables of $mayflow_{interUC}$ are use cases.
4. finalFlow(X_aX_{obj},X'_{obj},X_t) is a finalized flow.
5. $finalFlow_{interUC}$(X_a,X_{obj},X'_{obj},X_t) is similar to finalFlow but it covers flows between use cases.
6. unsafeFlow(X_a,X_{obj},X'_{obj}) says there is an unsafe flow from X_{obj} to X'_{obj} initiated by actor X_a.
7. unsafeFlows(X_{a1},X_{obj1},X'_{obj1}, X_{a2}, X_{obj2}, X'_{obj2}) says there are two unsafe flow. The first, initiated by actor X_{a1} flows from X_{obj1} to X'_{obj1}. The second, initiated by actor X_{a2} flows from object X_{obj2} to object X'_{obj2}. They are unsafe because together they violate a flow constraint.
8. safeFlow(X_a,X_{obj},X'_{obj}) says that the flow initiated by actor X_a from object X_{obj} to object X'_{obj} is safe.

Predicates for defining fine-grain policies

The predicates used to specify fine-grain access control policies are similar to the ones for coarse grain policy, but include X_{att} as an attribute flowing between objects. The predicates are as follows: $flow_{att}$(X_a,X_{att},X_{obj},X'_{obj},X_t,X_{uc}),
$mayFlow_{att}$(X_a,X_{att},X_{obj},X'_{obj},X_t,X_{uc}), $mayFlow_{interUC_att}$ (X_a,X_{att},X_{obj},X'_{obj},X_t,X_{uc}), $finalFlow_{att}$(X_a,X_{att},X_{obj},X'_{obj},X_t), $finalFlow_{interUC_att}$ (X_a,X_{att},X_{obj},X'_{obj},X_t), $unsafeFlow_{att}$(X_a,X_{att},X_{obj},X'_{obj}), $unsafeFlows_{att}$ (X_{a1},X_{att1},X_{obj1},X'_{obj1},X_{a2},X_{att2},X_{obj2},X'_{obj2}), $safeflow_{att}$(X_a, X_{att1},X_{obj},X'_{obj}).

FlowUML uses a set of predicates, as summarized in Tables 1 and 2. A FlowUML rule is of the form $L \leftarrow L1, \ldots L_n$ where L , L_1, …L_n are literals satisfying the conditions stated in Tables 1 and 2. Rules constructed according to these specifications form a locally stratified logic program, and therefore has a unique stable model and that stable model is also a well-founded model (Gelfond and Lifschitz, 1988).

7. APPLYING FLOWUML

This section shows sample FlowUML policies and apply them to detect unsafe flows. Due to space limitation, we only present the basic flow predicates, propagation policies, and the security verification for discretionary access control.

Phase	Stratum	Predicate	Rules defining the predicate
	0	FlowSup predicates ConstSup predicates	base relations. base relations.
Coarse-grain	1	Flow(X_a, X_{obj}, X'_{obj}, X_t, X_{uc})	body may contain FlowSup predicates.
	2	mayFlow(X_a, X_{obj}, X'_{obj}, X_t, X_{uc})	body may contain literal from strata 0 to 2
	3	$mayFlow_{interUC}$(X_a, X_{obj}, X'_{obj}, X_t, X_{uc})	body may contain literal from strata 0, 1 and 3
	4	finalFlow(X_a, X_{obj}, X'_{obj}, X_t) $finalFlow_{interUC}$(X_a, X_{obj}, X'_{obj}, X_t)	body may contain literal from strata 0 to 3
	5	unsafeFlow(X_a, X_{obj}, X'_{obj}) unsafeFlows(X_{a1}, X_{obj1}, X'_{obj1}, X_{a2}, X_{obj2}, X'_{obj2})	body may contain literal from strata 0 to 4
	6	safeFlow(X_a, X_{obj}, X'_{obj})	body may contain literal from strata 0 to 5

Table 1: FlowUML's strata for coarse-grain policies

Phase	Stratum	Predicate	Rules defining the predicate
	0	FlowSup predicates ConstSup predicates	base relations. base relations.
Fine-grain	1	$Flow_{att}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc})$	body may contain FlowSup predicates.
	2	$mayFlow_{att}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc})$	body may contain literal from strata 0, 1 and 2
	3	$mayFlow_{interUC_att}(X_a, X_{obj}, X'_{obj}, X_t, X_{uc})$	body may contain literal from strata 0, 1 and 3
	4	$finalFlow_{att}(X_a, X_{obj}, X'_{obj}, X_t)$ $finalFlow_{interUC\ att}(X_a, X_{obj}, X'_{obj}, X_t)$	body may contain literal from strata 0 to 3
	5	$unsafeFlow_{att}(X_a, X_{att}, X_{obj}, X'_{obj})$ $unsafeFlows_{att}(X_{a1}, X_{att1} X_{obj1}, X'_{obj1}, X_{a2}, X_{att2}, X_{obj2}, X'_{obj2})$	body may contain literal from strata 0 to 4
	6	$safeFlow_{att}(X_a, X_{att1}, X_{obj}, X'_{obj})$	body may contain literal from strata 0 to 5

Table 2: FlowUML's strata for fine-grain policies

Basic flow predicates: Examples flow information available in Figure 2 are given in rules (1). The rules (2) to (6) are instances of *FlowSup* predicates of Figures 2 and 3.

$Flow_{att(}ActorA,att1,Obj1,ActorA,1,uc1) \leftarrow Flow_{att}(ActorA,att1,ActorA,Obj2,2,uc1) \leftarrow$ (1)

$isEntity(obj1) \leftarrow$, $isActor(actorA) \leftarrow$, $precedes(uc1,uc2) \leftarrow$, (2,3,4)

$sameAtt(att3, att3') \leftarrow$, $derAtt(att3, att6, control) \leftarrow$ (5,6)

Propagation policies: The second step applies a policy that propagates flows along actor hierarchies. In our example, if there is a specialized actor say *Actor C* of *Actor B* then *Actor C* receives *att5* and *att6*. Policies stating acceptable inheritances can be stated in example rules such as (7) to (9).

Rule 7 say that every actor that plays a specialized role of an actor that initiates a flow also initiates an inherited flow. For every flow from or to an actor, rules 8 and 9 add new information flow for every specialized actor of the actor who sends or receives the information, respectively.

$Flow(X'_a, X_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow Flow(X_a, X_{obj}, X'_{obj}, X_t, X_{uc}), specializedRole(X_a, X'_a)$ (7)

$Flow(X_a, X''_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow Flow(X_a,X_{obj},X'_{obj},X_t,X_{uc}), isActor(X_{obj}), specializedActor(Xobj,X''_{obj})$ (8)

$Flow(X_a, X_{obj}, X''_{obj}, X_t, X_{uc}) \leftarrow Flow(X_a,X_{obj},X'_{obj},X_t,X_{uc}), isActor(X'_{obj}), specializedActor(X'_{obj},X''_{obj})$ (9)

Detecting flows unsafe with respect to policies: This section shows FlowUML specification of known flow control policies, and how to detect information flows unsafe with respect to them.

Mandatory Access Control (MAC) (Bell and LaPadula, 1975) permits a subject to read an objects only if the subject's security clearance dominates the security classification of the object. Rule 18 specify if information flows from *Obj1* to *Obj2*. If *Obj2* does not dominate the security label of *Obj1* then the flow is considered unsafe.

$unsafeFlow(X_a, X_{obj1}, X_{obj2}) \leftarrow finalFlow(X_a, X_{obj1}, X_{obj2}, X_t), \neg dominates(X_{obj1}, X_{obj2})$ (18)

Discretionary access control (DAC) allows subjects to access objects solely based on the subjects' identity and the authorization rule. Rule 19 specify unauthorized information flows from an actor to an object and rule 20 specify unauthorized flows from an object to an actor.

$unsafeFlow_{att}(X_a, X_{att1}, X_{obj1}, X_{obj2}) \leftarrow finalFlow(X_a, X_{att1}, X_{obj1}, X_{obj2}, X_t), \neg ACL(X_{obj1}, X_{obj2}, w)$ (19)

$unsafeFlow_{att}(X_a, X_{att1}, X_{obj2}, X_{obj1}) \leftarrow finalFlow(X_a, X_{att1}, X_{obj2}, X_{obj1}, X_t), \neg ACL(X_{obj1}, X_{obj2}, r)$ (20)

8. RELATED WORK

Secure information flow models have been studied by several researchers. Most of these studies aim to develop flow control models. For example, Samarati *et al* develops a Discretionary Access Control (DAC) based model that prevents information leakage by Trojan Horses (Samarati *et al*, 1997). Bertino and Atluri (1999) present a logic based workflow model. Although important, these papers do not address flow policies for software design life cycle models.

Myers (1999) presents JFlow, an extension to Java that adds statically checkable flow constraints. However, JFlow can be used during the implementation phase while FlowUML is to be used during the requirements, design and analysis phases of the software development life cycle. Lodderstedt *et al* (2002) present a language to annotate UML-based models representing authorization constraints. Jurjens (2002) proposes to extend UML to represent security information within UML diagrams.

To the best of our knowledge, none of the existing research addresses the design and verification processes supported by FlowUML. However, some of the published approaches share the objectives of FlowUML. In the area of logic-based checking of policy violations during the requirement engineering, Alghathbar and Wijesekera (2003) developed AuthUML, a logic program based framework for that analyzing access control requirements during the requirements engineering phase to ensure consistency, completeness, and conflict free design. AuthUML is the access control analog of FlowUML.

FlexFlow, developed by Chen *et al* (2003), is a logic-based flexible flow control framework to specify data-flow, workflow and transaction systems. Although FlowUML and FlexFlow analyze and prevent unauthorized flows, FlowUML is different from FlexFlow in many aspects such as FlexFlow is an abstract model while FlowUML is tightly integrated with UML diagrams.

9. CONCLUSIONS

FlowUML is a logic programming based framework to specify and validate information flow policies in UML based designs at the early phases of the software development life cycle. Due to early detection of errors or inconsistencies, correcting them is usually easier and cheaper than correcting them at a latter stage of the life cycle.

We demonstrated the usefulness of FlowUML by validating existing example policies.

ACKNOWLEDGEMENT

Farkas' work was partially supported by the National Science Foundation under grant number IIS-0237782.

REFERENCES

ALGHATHBAR, K. and WIJESEKERA, D. (2003): authUML: A Three-phased framework to analyze access control specifications in Use Cases. In *Proc. of the Workshop on Formal Methods in Security Engineering (FMSE)*. Washington D.C. October. ACM Press.

BELL, D. and LAPADULA, L. (1975): Secure computer system: United exposition and Multics interpretation. Technical Report, ESD-TR-75-306, MITRE Corp. MTR-2997. Bedford, MA.

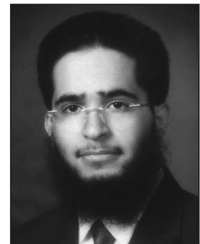
BERTINO, E. and ATLURI, V. (1999): The specification and enforcement of authorization constraints in workflow management. *ACM Transactions on Information Systems Security*.

BOOCH, G., RUMBAUGH, J. and JACOBSON, I. (1999): *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA.

CHEN, S., WIJESEKERA, D. and JAJODIA, S. (2003): FlexFlow: A flexible flow control policy specification framework. In *Proceedings of the 17th Annual IFIP WG 11.3 Working Conference on Database and Applications Security*. Estes Park, Colorado.

CHUNG, L., NIXON, B., YU, E. and MYLOPOULOS, J. (2000): *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers.
DEVANBU, P. T. and STUBBLEBINE, S. (2000): Software engineering for security: A roadmap. In *The Future of Software Engineering*. FINKELSTEIN, A. (ed). ACM Press.
GELFOND, M. and LIFSCHITZ, V. (1988): The stable model semantics for logic programming. In *Proceedings, 5th International Conference and Symposium on Logic Programming*. Seattle, Wash. 1070–1080.
MYERS, A. (1999): JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*. 228—241. San Antonio, TX.
NUSEIBEH, B., EASTERBROOK, S. and RUSSO, A. (2001): Making respectable in software development. *Journal of Systems and Software*, 56(11). Elsevier Science Publishers.
NUSEIBEH, B. and EASTERBROOK, S. (2000): Requirements engineering: A roadmap. In *The Future of Software Engineering*. FINKELSTEIN, A. (ed). ACM Press.
SAMARATI, P., BERTINO, E., CIAMPICHETTI, A. and JAJODIA, S. (1997): Information flow control in object-oriented systems. *IEEE Transactions on Knowledge and Data Engineering*. 9(4):524–538.
THE UNIFIED MODELING LANGUAGE VERSION 1.5 (2003). http://www.omg.org/uml/. Accessed in September 2003.
LODDERSTEDT, T., BASIN, D. and DOSER, J. (2002): SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, 426-441.
JURJENS, J. (2002): UMLsec: Extending UML for Secure Systems Development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, 412-425.

BIOGRAPHICAL NOTES


Khaled Alghathbar

Khaled Alghathbar is on the faculty of Information Systems in Kind Saud University, Riyadh, Saudi Arabia. He received a PhD in Information Technology from George Mason University in 2004 and has several certifications in information systems security. His dissertation is in the area of incorporating access and flow control policies in requirements engineering. He developed a framework to formally verify access and flow control policies during the early stages of the development life cycle. He has published several papers in international journals, conferences and workshops.

Csilla Farkas

Csilla Farkas is an assistant professor at the Department of Computer Science and Engineering of University of South Carolina, Columbia. Her current research interests include information systems security, semi-structure data security, database inferences, intrusion detection and database management systems. Csilla Farkas received her PhD from George Mason University, Fairfax. In her dissertation she studied the inference and aggregation problems in multilevel secure relational databases. She received a MSc in computer science from George Mason University and BSc degrees in computer science and geology from SZAMALK, Hungary and Eotvos Lorand University, Hungary, respectively.

Duminda Wijesekera

Duminda Wijesekera is an associate professor in the Department of Information and Software Engineering at George Mason University, Fairfax, Virginia. He has contributed to security multimedia, networks, avionics and theoretical computer science. His work in security has been in applying logical methods, telecommunication and TCP/IP based network security, multimedia security and securing the early phases of the software life cycle. He has been associated with the Center for Secure Information Systems (CSIS) at George Mason University since 1999. Prior to joining GMU, he was a senior systems engineer at Honeywell Space Systems in Clearwater, Florida. He has been a visiting post-doctoral fellow at the Army High

Performance Research Center at the University of Minnesota, and an assistant professor of Mathematics at the University of Wisconsin. Dr Wijesekera received a PhD in Computer Science from the University of Minnesota and a PhD in Mathematical Logic from Cornell University.