Secure UML Information Flow using FlowUML

Khaled Alghathbar[1], Duminda Wijesekera[1], and Csilla Farkas[2†]

[1] Dept. of Information and Software Engineering and CSIS,
George Mason University, MS 4A4, Fairfax, VA 22030

[2] Information Security Laboratory, Dept. of Computer Science and Engineering,
University of South Carolina, Columbia, SC 29208

Abstract. 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.

1 Introduction

As security becomes an important aspect of large software systems, it needs to be addressed through out the software development life cycle. Considered a non-functional requirement, security requirements have not been considered during the early phases of system development; thus resulting in vulnerable software [7]. In [13,14] Nuseibeh and Easterbrook present the need to formally analyze and validate security requirements during earlier phases to detect and remove design vulnerabilities Recent work has addressed information flow control security [3,1,12,16]. Information flow requirements need to be developed and evaluated during the requirements and design stages of the software life cycle, as validating information flow requirements at an early stage prevents costly fixes mandated 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 [17]. 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 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.

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

Alghathbar K., Wijesekera D. and FarkasâĂă C. (2005).
Secure UML Information Flow using FlowUML.
In *Proceedings of the 3rd International Workshop on Security in Information Systems*, pages 229-238
DOI: 10.5220/0002569802290238

FlowUML does not assume the existence of a particular meta-policy and can support a variety of policies. FlowUML advances over existing secure information flow models by applying a formal framework to the early phases of the software development life cycle. Towards that end, 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 has the syntax and semantics of FlowUML. Section 7 shows the expressibility 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. 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 [17]. *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 Fig. 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.

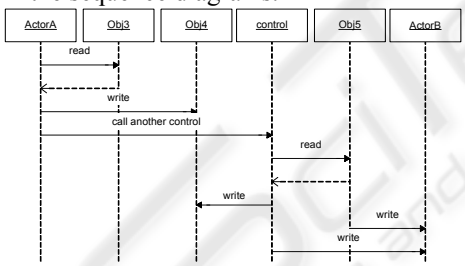

Fig. 1. A sample sequence diagram

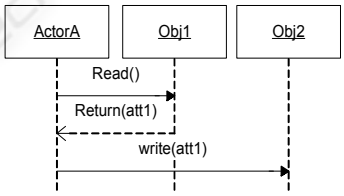

Fig. 2. The sequence diagram for use case1

3 Running Example

Our example consists of two use cases. Use case 1 has a simple scenario shown in Fig. 2, and use case 2 has a more complex scenario shown in Fig. 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 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 Fig. 1 and the fine grain view shown in Fig. 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.

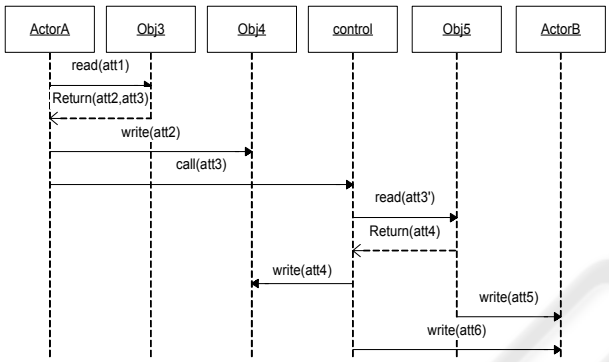

Fig. 3. Detailed sequence diagram for use case 2

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 as externals. 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 and 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 based satisfying certain 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 the 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 flow 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. An attribute that flows out of an object depends upon a set of attributes flowing into the same object. We say that the output attribute is derived from the input attributes.*

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

Axiom 4: *Pairs of exact attribute s are entered into the exact attribute table and dependent attributed have an entry in the dependent 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 Fig. 4. Coarse grain policies require less detail than fine grain policies. Fine grain policies incorporate attribute values and their dependencies.

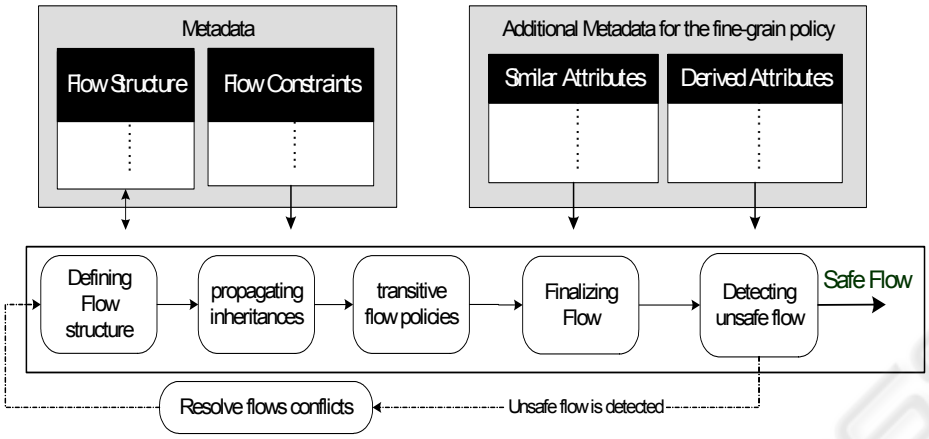

Fig. 4. Steps in FlowUML

Coarse grain policy analysis

In the first step FlowUML extracts flow structure, called the *defining flow structure,* from 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 Fig. 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 only entities and actors. Thus only flows from *Obj3* to *Actor A* and the one from A*ctor 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 Fig. 3, if there is a record in the *attribute derivation table* stating that *att6* is derived from *att3* in the *control* object then FlowUML concludes that there is a transitive flow from *Actor A* to *Actor B*.

6 Syntax and semantics

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. A unary predicate isEntity(X_{obj}) meaning X_{obj} is an entity.
2. A unary predicate isActor(X_{obj}) meaning X_{obj} is an actor.
3. A binary predicate specializedActor(X_a,X'_a)meaning X'_a is a specialized actor of X_a.
4. A binary predicate precedes(X_{uc}, X'_{uc}) meaning use cases X_{uc} and X'_{uc} are executed in that order.
5. A binary predicate sameAtt(X_{att}, X'_{att}) meaning attributes X_{att} and X'_{att} have the same value.
6. A binary predicate derAtt(X_{att}, X'_{att}, X_{obj}) meaning that attribute X'_{att} is derived from attribute X_{att} in object X_{obj}.
7. A ternary predicate ignoreFlow(X_a, X_{obj}, X'_{obj}) meaning to exclude the flow from object X_{obj} to object X'_{obj}.
8. A 6-ary predicate ignoreFlows(X_{a1}, X_{obj1}, X'_{obj1}, X_{a2}, X_{obj2}, X'_{obj2}) to exclude the flow from X_{obj1} to X'_{obj} and the flow from X_{obj2} to X'_{obj2} from being detected as a violation of flow control policies. Two similar 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. A binary predicate dominates(X_{obj},X'_{obj}) meaning that the security label of object X'_{obj} dominates or equals the security label of object X_{obj}. It is used in multi-level security policies.
2. A binary predicate ACL(X_{obj},X'_{obj},X_{AT}) meaning that object X_{obj} is in the access control list of object X'_{obj}. It is used in discretionary access control policies. X_{AT} is an operation such as *read* or *write*.
3. A binary predicate conflictingActors(X_{obj},X'_{obj}) meaning that actors X_{obj} and X'_{obj} are in conflict with each other for a specific reason. For example, both actors can not flow information to the same object or there must not be a flow of information between them.
4. A binary predicate conflictingEntities(X_{obj},X'_{obj}) meaning both entities X_{obj} and X'_{obj} are in conflict with each specific reason. For example, each entity belongs to different competitive company and information must not flow in between.

Predicates for coarse-grain policies

1. A 5-ary predicate Flow(X_a,X_{obj},X'_{obj},X_t,X_{uc}) meaning 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. A 5-ary predicate 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 scope of mayflow$_{interUC}$ is between use cases instead of focusing in one use case.
4. A 4-ary predicate finalFlow($X_a$$X_{obj}$,$X'_{obj}$,$X_t$) meaning a finalized flow.
5. finalFlow$_{interUC}$(X_a,X_{obj},X'_{obj},X_t) is similar to finalFlow but it covers flows between use cases.
6. A ternary predicate unsafeFlow(X_a,X_{obj},X'_{obj}) meaning there is an unsafe flow from X_{obj} to X'_{obj} initiated by actor X_a.

7. A 6-ary predicate unsafeFlows(X_{a1},X_{obj1},X'_{obj1}, X_{a2}, X_{obj2}, X'_{obj2}) meaning 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. A ternary predicate safeFlow(X_a,X_{obj},X'_{obj}) meaning that the flow initiated by actor X_a from object X_{obj} to object X'_{obj} is safe.

Predicates for 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: $flow_{att}(X_a,X_{att},X_{obj},X'_{obj},X_t,X_{uc})$, $mayFlow_{att}(X_a,X_{att},X_{obj},X'_{obj},X_t,X_{uc})$, $mayFlow_{interUC\_att}(X_a,X_{att},X_{obj},X'_{obj},X_t,X_{uc})$, $finalFlow\text{-}_{att}(X_a,X_{att},X_{obj},X'_{obj},X_t)$, $finalFlow_{interUC\_att}(X_a,X_{att},X_{obj},X'_{obj},X_t)$, $unsafeFlow\text{-}_{att}(X_a,X_{att},X_{obj},X'_{obj})$, $unsafeFlows_{att}(X_{a1},X_{att1},X_{obj1},X'_{obj1},X_{a2},X_{att2},X_{obj2},X'_{obj2})$, $safe\text{-}flow_{att}(X_a, X_{att1},X_{obj},X'_{obj})$.

FlowUML uses a set of predicates, as summarized in Tables 1 and 2 A FlowUML rule is of the form $L \leftarrow L_1, \ldots L_n$ where $L , L_1, \ldots L_n$ 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 [11].

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.
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 2. FlowUML's strata for fine 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

7 Applying FlowUML

This section shows how some 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.

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

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

isEntity(obj1)← , isActor(actorA)←, precedes(uc1,uc2)←, (2,3,4)

sameAtt(att3, att3')←, derAtt(att3, att6, control)← (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. In rules 8 and 9, for every flow from or to an actor, respectively, 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)

$Flow(X_a, X''_{obj}, X'_{obj}, X_t, X_{uc}) \leftarrow Flow(X_a, X_{obj}, X'_{obj}, X_t, X_{uc})$,$isActor(X_{obj})$, $specializedActor(X_{obj}, X''_{obj})$ (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) permits a subjects to read an objects only if the subject's security clearance dominates the security classification of the object (2). Rule 18 specify if information flows from *Obj1* to *Obj2*. If *Obj2* does not dominate or equals 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 subject's 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 develop richer flow control models. For example, Samarati et al. develops a DAC based model that prevents information leakage by Trojans [16]. Bertino et al. [3] present a logic based workflow model. Although important, these papers do not address flow policies for software design life cycle models.

Myers [12] 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. [18] present a language to annotate UML-based models representing authorization constraints. Jurjens [19] proposes to extend UML, allowing to represent security information within UML diagrams.

To the best of our knowledge, none of the existing works address the design and verification processes supported by FlowUML. However, some published work shares some of FlowUML's objectives. In the area of logic-based checking of policy violations during the requirement engineering, Alghathbar et al. [1] developed AuthUML, a logic program based framework for that analyzing access control requirements during the requirements engineering phase to ensure consistency, completeness and conflict-freedom. AuthUML is the access control analog of FlowUML.

FlexFlow of Chen et al. [6] 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. We demonstrated the expressiveness of FlowUML by validating existing example policies.

References

1. K. Alghathbar, D. Wijesekera. "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, DC. October 2003. ACM Press.
2. D. Bell and L. LaPadula. "Secure computer system: United exposition and Multics interpretation". Technical Report, ESD-TR-75-306, MITRE Corp. MTR-2997. Bedford, MA, 1975.
3. E. Bertino and V. Atluri. "The specification and enforcement of authorization constraints in workflow management". ACM transactions on Information Systems Security, February 1999.
4. B. Boehm. *Software engineering economics*. Englewood Cliffs, NJ: Prentice-Hall. (1981)

5. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.
6. S.Chen, D. Wijesekera, S. Jajodia. "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. August 2003.
7. L. Chung, B. Nixon, E. Yu, J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers (2000).
8. P. T. Devanbu and S. Stubblebine. "Software engineering for security:A roadmap". In A. Finkelstein, editor, The Future of Software Engineering. ACM Press, 2000.
9. D. Gabbay and A. Hunter, "Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning, Phase1 - A Position Paper", Proceedings of Fundamentals of Artificial Intelligence Research '91, 19-32, Springer-Verlag.
10. D. Gabbay and A. Hunter, "Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning, Phase2", In Symbolic and Quantitative Approaches to Reasoning and Uncertainty, 129-136, LNCS, Springer-Verlag, 1992.
11. M. Gelfond, V. Lifschitz. 1988. "The stable model semantics for logic programming". In Proceedings, 5th International Conference and Symposium on Logic Programming. Seattle, Wash. pp. 1070–1080.
12. A. Myers. "JFlow: Practical mostly-static information flow control". In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL), pages 228--241, San Antonio, TX, January 1999.
13. B. Nuseibeh, S. Easterbrook and A. Russo, "Making Respectable in Software Development", Journal of Systems and Software, 56(11), November 2001, Elsevier Science Publishers
14. B. Nuseibeh and S. Easterbrook. "Requirements engineering: A roadmap". In A. Finkelstein, editor, The Future of Software Engineering. ACM Press, 2000.
15. Rational Rose. http://www.rational.com.
16. P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia. "Information flow control in object-oriented systems. IEEE Transactions on Knowledge and Data Engineering", 9(4):524–538, July-Aug. 1997.
17. The Unified Modeling Language version 1.5. http://www.omg.org/uml/. Accessed in September 2003.
18. T. Lodderstedt, D. Basin, and J. Doser. "SecureUML: A UML-Based Modeling Language for Model-Driven Security," in Proceedings of the 5th International Conference on The Unified Modeling Language**,** pp. 426-441, 2002.
19. J. Jurjens. "UMLsec: Extending UML for Secure Systems Development," in Proceedings of the 5th International Conference on the Unified Modeling Language, pp. 412-425, 2002**.**