Information Flow Analysis of Component-Structured Applications

Peter Herrmann
University of Dortmund, Computer Science Department, 44221 Dortmund, Germany
Peter.Herrmann@cs.uni-dortmund.de

Abstract

Software component technology facilitates the cost-effective development of specialized applications. Nevertheless, due to the high number of principals involved in a component-structured system, it introduces special security problems which have to be tackled by a thorough security analysis. In particular, the diversity and complexity of information flows between components hold the danger of leaking information. Since information flow analysis, however, tends to be expensive and error-prone, we apply our object-oriented security analysis and modeling approach. It employs UML-based object-oriented modeling techniques and graph rewriting in order to make the analysis easier and to assure its quality even for large systems. Information flow is modeled based on Myers' and Liskov's decentralized label model combining label-based read access policy models and declassification of information with static analysis. We report on the principles of information flow analysis of component-based systems, clarify its application by means of an example, and outline the corresponding tool-support.

1. Introduction

Due to the increasing deployment of information technology in enterprises, information system security is getting more and more important. In order to guarantee secure and reliable operation, a security model is designed identifying relevant principals, components, attributes, functions, and component interactions for a class of systems on a relatively abstract level. Based on a model a system-specific security policy[1] is defined for a particular system enforcing certain security services which provide objectives concerning confidentiality, integrity, availability, and accountability [2].

In order to apply suitable security services for an existing or newly designed information system, it has to undergo a security analysis. Here, the system is audited for vulnerabilities, threats, and risks. Based on the audit results effective safeguards are selected, designed, and configured. A security analysis, however, is mostly expensive and laborious. Due to the complexity of modern IT systems, the audit has to be performed by well-trained experts. Moreover, system users often require that the analysis considers extensive recommendations and standards in order to guarantee certain security levels (e.g., UK Government recommends the use of the security method CRAMM for systems employed in government departments). Furthermore, one has to be constantly aware of changing system threats requiring a new security analysis in regular intervals.

A survey of security analysis approaches is provided in [6]. Typically, an audit comprises a possibly iterated series of phases concerning the following subtasks:

1. Identification of the system, its components, and the related principals,
2. valuation of the assets contained in the system and definition of the security objectives,
3. identification of vulnerabilities and threats,
4. assessment of resulting risks,
5. planning, design, and evaluation of suitable countermeasures.

To reduce the expense of a security analysis, abstract models of the systems as well as of the security-related requirements are developed recently (cf. [6]). The system model forms the basis for the introduction of problem solutions which are described by model modifications. Finally, the abstract solutions are refined to implementable countermeasures.

Our object-oriented security analysis approach [18] applies abstract modeling, model-based analysis, and logical transformation. Unlike some other approaches (e.g., [5, 11, 24, 26]) it uses object-oriented techniques and graph rewriting to facilitate the subtasks and to enable automation. The corresponding tool-support is similar to object-oriented design tools which are well established in the field of software engineering (e.g., [37, 43]). In fact, it re-uses the graph editing framework of the Argo project [43].

[1] This term should not be confused with the term "organizational security policy" describing a management's security strategy for protecting assets.

The tool supports the interactive design of graphical system models in form of class and object diagrams as they are defined in the well-known Unified Modeling Language (UML) [10]. The first two subtasks (i.e., system modeling, asset valuation, and objective definition) are supported by libraries of predefined object classes which model system component types. By multiple inheritance the classes can be associated in order to support separation of concern. Moreover, the classes define basic attributes and methods for automated analysis and evaluation.

The remaining three subtasks of a security analysis are performed in a highly automated fashion by application of graph rewrite systems (e.g., [4]). A rewrite system consists of a set of graph rewrite rules. Each rule is a tuple of two graph patterns — a pre-pattern and a post-pattern —, an application condition, and an effect function. The rule can be applied to a graph if the graph contains a subgraph which is an instance of its pre-pattern. Moreover, the object attributes in the subgraph have to fulfill the application condition. By application of the rule the subgraph is replaced by an instance of the post-pattern. The attributes of the replacement objects are set according to the effect function.

In our approach a separate rewrite system is defined for each subtask. For instance, the rules of the rewrite system for vulnerability identification contains pre-patterns describing IT system scenarios which come along with certain vulnerabilities for the systems. The corresponding post-patterns describe the scenario augmented by special objects representing vulnerabilities and threats on the scenario elements. Thus, by application of these rules the graphical system model can be provided by vulnerability and threat representations. Likewise, by the other graph rewrite systems one can augment the model by suitable risk and safeguard representations.

By provision of different sets of class libraries and rewrite systems one can facilitate security analysis in various application domains. A first domain specific realization [20] allows analysis of applications based on the middleware platform CORBA [34] which will be audited and protected according to the CORBA security specification [35].

This paper is centered on the domain of component-structured systems. We will introduce the use of our security analysis approach for analyzing information flow between software components. As a basis we use the *decentralized label model* of Myers and Liskov [30, 31, 32] which is well-suited to our work. In contrast to other information flow approaches it facilitates the checks at design time instead of run-time which makes it feasible to security analysis which is performed at design time as well.

In the sequel we will give a short introduction into component-structured software and outline its major security aspects. Thereafter we will sketch the decentralized label model and our security analysis approach. Afterwards, we will introduce the application of our approach for information flow analysis of component-structured systems. Finally, an example application will be discussed.

2. Component Security

Applications coupled from software components (cf. e.g., [42]) get more and more popular since their creation is quite easy and cost-effective and they can be tailored to the particular needs of their customers. Moreover, component-based software can easily be extended and modified by changing components and their coupling dynamically. The components are supplied by different developers and offered on an open market. The application designer selects and configures suitable components — probably from various sources — and couples them to the target application. The combination process utilizes the concept of explicit contracts. A contract is legally binding and describes the agreed properties of a component, especially its interface, its operations, and the relation with its environment. Moreover, component coupling is comforted by rich run-time support like special component methods providing reflection and introspection [40] (i.e., the exploration of component properties, methods, and interfaces at run-time). Furthermore, coupling is facilitated by scripting languages or visual application builder tools. One can also realize distributed component-structured systems. Here, components are executed on remote hosts and can be booked as communication services. Platforms for component-structured software comprise Java Beans [40], COM/DCOM [29], and the CORBA Component Model [33].

Component-structured systems, however, impose new security aspects since in comparison with ordinary distributed object-oriented systems more principals and roles are involved. Besides users and application owners, one has to consider also component vendors, remote host providers, and application builders. These new principals introduce new types of threats since, generally, they cannot trust each other to full extent. In particular, a malicious component may spoil security objectives of the whole application if it, for instance, secretly lacks data to principals who have not the privilege to read them. Therefore, like other applications the security of component-structured systems has to recognize the definition of user classes and roles, the authentication of users, and access control. Since components may be coupled via networks, services for secure communication must be enforced as well. Moreover, distributed components are executed on different hosts. As the host and application owners not necessarily trust each other, components have to be protected against malicious host environments and vice versa (cf. [13, 23]). Similarly, application owners and component vendors have to be pro-

tected against each others, too. An application has to be safeguarded against malicious components which may violate the integrity of the application and its resources, the confidentiality and availability of the managed information and supported operation, and the accountability of the performed actions. In contrast, a component vendor has to be protected against wrong accusations due to spite of application administrators, malicious surrounding components, and host malfunctions. Moreover, the vendor needs protection against unlicensed use of components.

The corresponding security model identifies various roles like users, resource owners, application owners, and component vendors. Objects in the model comprise software components, component interfaces, resources, host environments, and communication facilities. Associations between objects express the configuration of components to an application and the contribution of a component to the functionality of the application. A component accesses resources and forwards information and control to other components resp. the application environment. Component interfaces provide several types of mechanisms like method invocations, event coupling, and infobusses [41]. During run-time a series of interface events occur which forms the behavior of the component. Based on the security model a component-specific system security policy constrains the behavior by defining static and dynamic conditions in order to guarantee security-related objectives for a certain component-structured system.

One can check that a component fulfills these conditions by utilizing the explicit component contracts (cf. [42]) which can be extended in order to describe security-relevant obligations (e.g., a certain data unit must only be forwarded via an explicit interface). The obligations may be specified formally in the form of behavior constraints (cf. [3, 17]) enabling the component user to prove in two steps that a component fulfills owner's security policy. At first, one has to verify that all obligations and their combinations fulfill the conditions of the security policy. This proof can be performed at design time by means of a security analysis. For instance, by analyzing the data flow between components one can check that data units pass only principals who are entitled to read them. At second, one must check that the obligations are kept by the components which is performed either at design time by source code analysis resp. byte code verification or at run-time by security wrappers [19]. Here, the interface behavior of a component is observed and checked for correspondence with the obligations.

Access control and information flow models are well-suited to prove that components guarantee the confidentiality of applications. Access control (and also authentication) is used to limit the access to single components to a set of entitled principals (cf. [7, 25, 38]). Our security analysis approach can also be used for introducing access control and authentication systems to IT systems and components which is described in [18].

While access control models constrain the release of information, they do not limit its propagation between the components of the application (cf. [31]). In contrast, by application of information control models one may prevent unauthorized disclosure of information due to wrong propagation paths. By these models one can prove that the path of an data unit contains only components granting reading access to readers allowed to read the data unit. Moreover, one can distinguish harmless from dangerous components. Unlike a harmless component, a dangerous component C is coupled to a component $\overline{C}$ granting access to a principal who is not privileged to read a data unit D passing C. C is potentially dangerous since it may maliciously or erroneously leak D to $\overline{C}$ exposing it to a read access by a non-privileged principal. Dangerous components have to be scrutinized for compliance with the information flow policies of the system owner.

3. Decentralized Labeling

In order to carry out information flow control in a security analysis at design time, we have to use a model facilitating static analysis. We selected Myers' and Liskov's *decentralized label model* [30, 31, 32] since it combines static analysis with labeling and declassification. Labels are special identifiers assigned to system units like components, interfaces, and information. A unit owner may specify a separate read access policy for each system unit by setting its label. In our object-oriented security analysis approach labels can easily be integrated in the form of special class properties. In contrast to its customary definition, here, declassification describes a special operation on labels which permits a unit owner to relax the unit's read access policy. Since in real-life systems the read access policies of data have to be relaxed sometimes, declassification is a necessary means to apply information flow control in practice.

Information flow control was dealt with in various approaches. For instance, other work proposes the application of labels [15, 27, 28, 39] and declassification [14], too. The correctness checks, however, are carried out during run-time in these approaches which makes them less suited to our application. In contrast, most approaches for static analysis of information flows treat information flow analysis only by type checking of secure programming languages (e.g., [1, 16, 36, 44]) which is also unfavorable for security analysis. Therefore we decided to apply the decentralized label model.

In this model each component, component interface, and data unit is provided with a label describing its individual read access policy (cf. Sec. 5). Labels refer to a set of identifiers indicating principals, principal groups, or roles. As

an example we will refer to the label $L = \{b : a, b; c : b, c\}$ below where $\{a, b, c\}$ is a set of principals. Labels consist of a list of sub-labels[2] separated by semicolons (e.g., $b : a, b$). A sub-label contains an owner identifier and a list of reader identifiers which are separated by colons. The owner identifier denotes an owner of system unit who defines a read access policy. The readers, to whom access is granted by the owner, are indicated by the reader identifiers. Thus, in the first sub-label of L principal b defines that principal a and herself may read the corresponding system unit. A labeled system unit has to fulfill all sub-label policies. Therefore read access is granted to principals only who are readers in all sub-labels (e.g., label L allows only reading by principal b). So, the corresponding set of so-called effective readers is the intersection of the set of readers in the sub-labels.

Moreover, the decentralized label model facilitates the definition of principal hierarchies by a so-called *acts for*-relation $\succeq$. If $a \succeq b$ holds, one may add a as a reader to each sub-label which contains b as a reader. Thus, if a acts for b, to each system unit, which may be read by b, a is also granted reading access. Moreover, b may be replaced by a as an owner of a sub-label. Thus, a may adopt an access policy of b. Due to these rules label L may be replaced by the label $\widehat{L} = \{a : a, b; c : a, b, c\}$. The relation $\succeq$ is reflexive and transitive but not anti-symmetric. Thus, two distinct principals may act in behalf of each other.

The *relabeling* operator $\sqsubseteq$ enables to compare the access policies of two labels. If $L_1 \sqsubseteq L_2$ holds, the policy of L_2 is more restrictive than that of L_1, i.e., the effective readers of L_2 are equal to or a subset of the effective readers of L_1. Defining

$$L_1 = \{o_1 : r_{1,1}, \ldots, r_{1,k_1}; \ldots; o_m : r_{m,1}, \ldots, r_{m,k_m}\}$$
$$L_2 = \{p_1 : s_{1,1}, \ldots, s_{1,l_1}; \ldots; p_n : s_{n,1}, \ldots, s_{n,l_n}\}$$

then $L_1 \sqsubseteq L_2$ corresponds to the condition

$$\forall i \exists j [p_j \succeq o_i \wedge \forall t \exists u [s_{j,t} \succeq r_{i,u}]]$$

Thus, $L_1 \sqsubseteq L_2$ is valid if L_1 may be replaced by a label $\widehat{L_1}$ according to the rules of the acts for-relation and for each sub-label in L_2 exists a sub-label in $\widehat{L_1}$ with the same owner and the same or a subset of readers. Relabeling is important for the definition of a suitable information flow policy. For example, in a component-structured system the information flow is not violated if a data unit with label L is only propagated to components the labels M of which fulfill $L \sqsubseteq M$.

Declassification is the counterpart of relabeling. Here, the access policy of a system unit may be relaxed by adding readers to a sub-label or by removing a sub-label. In order to prevent policies of other principals, a declassification may be performed only by components which have the permission to act for the particular sub-label owner.

Finally, the decentralized label model defines functions for the combination of labels. For our application the *join*-operator $\sqcup$ is of particular interest. A *join* $N = L \sqcup M$ of two labels L and M is the least restrictive label which is a relabeling of L and M. Thus, N guarantees exactly the access policies of L and M. In general, N is the concatenation of the labels L and M. Assuming $L = \{b : a, b; c : b, c\}$ and $M = \{a : a, b; c : b\}$, the *join* $N = L \sqcup M$ is equal to $\{a : a, b; b : a, b; c : b, c; c : b\}$. In order to keep the number of sub-labels in a joined label short, one can omit redundant sub-labels. Since, for instance, in N the sub-label $c : b, c$ is superseded by the sub-label $c : b$ which defines a more restrictive policy, the *join* N corresponds to the label $\{a : a, b; b : a, b; c : b\}$, too. A complementary operator is the *meet* $\sqcap$ of two labels L and M. It is defined as the most restrictive label which can be relabeled to both L and M.

4. Object-Oriented Security Analysis

A set of so-called Common Criteria (CC) [21] standardizes the security analysis of IT systems and provides a methodology for vulnerability detection, risk assessment, and countermeasure integration. Fig. 1 delineates the main security classes and associations defined by the CC. Often, computer systems and system components store and maintain essential data and therefore are assets for their owners. These assets, however, are constantly exposed to threats by intruders, called threat agents, exploiting the vulnerabilities of the assets for attacks (e.g., a software may contain Trojan Horse code which may be utilized for eavesdropping data). Thus, the threat agents lead to security risks for the assets. The asset owners try to minimize these risks by imposing countermeasures which reduce the vulnerabilities (e.g., by a source code analysis the malicious Trojan Horse code may be detected and removed). The countermeasures, however, contain vulnerabilities themselves which have to be reduced by other countermeasures.

Our approach supports the creation of CC-compliant sys-

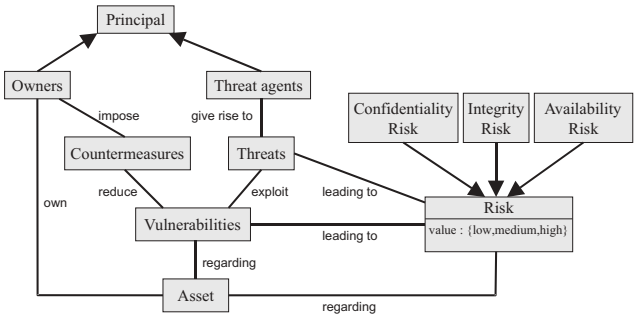

Figure 1. CC security classes

[2] Myers and Liskov speak about "label components". We altered this term to "sub-labels" in order to avoid confusion with software components.

Security level	Threat seriousness level						
	1	2	3	4	5	6	7
1	0	0	1	1	2	3	3
2	0	1	1	2	3	3	4
3	1	1	2	3	3	4	5
4	1	2	3	3	4	5	5
5	2	3	3	4	5	5	6
6	3	3	4	5	5	6	7
7	3	4	5	5	6	7	7

Table 1. Matrix for calculating risk values

tem models by providing a library of basic asset classes like networks, stations, applications, and data as well as associations between the classes. Moreover, for each application domain more specialized classes (e.g., software components, component interfaces, declassification permissions) are also available. Based on the class libraries, our ARGO-based tool facilitates the modeling of systems, consisting of asset class and association instances, in the form of UML object diagrams (cf. [10]). The classes contain three properties in order to define the extent of protection needed for an asset with respect to the main security objectives confidentiality, integrity, and availability. The properties refer to the seven security levels corresponding to the evaluation assurance levels defined in the CC. For instance, level 7 shall be assigned to the availability property of an asset if its breakdown leads to total collapse of the attacked institution.

In the next analysis phase vulnerabilities and threats on the assets are identified which are modeled as classes, too. Objects of these classes are added to the UML object diagram description of a system by application of graph rewrite rules (cf. [4]). Moreover, one has to estimate the seriousness of vulnerabilities and threats (i.e., the likelihood of attacks) which depends on the kind of applied countermeasures. The seriousness is modeled by a class property which may contain seven values, too. Depending on the particular domain the estimations are performed interactively or by support of graph rewrite rules.

Thereafter a graph rewrite system is used for determining the risks on the system assets. For each pair of an asset and a vulnerability an instance of a special risk class is created stating a risk for the confidentiality, integrity, or availability of an asset. Moreover, the tool calculates the value of the risk which is modeled by a class property, too. The risk level depends on the security level of the asset and on the seriousness level of the vulnerability (cf. [12]). Currently, the risk level is determined by application of the matrix[3] in Tab. 1 which, however, can be altered according to the applied security policy. Finally, one has to assess the risks which is also supported by a graph rewrite system. If all risks are bearable, the security analysis will be terminated, here.

The last phase is supported by a countermeasure class library and a rewrite system for the introduction of countermeasures. Attributes of a countermeasure object describe a protection level and the cost of imposing the countermeasure. The protection level refers to the CC which define requirements of countermeasures in order to fulfill a certain security level of an asset. In a first step, the tool suggests for each pair of an asset and a risk all countermeasures with a sufficient protection level (i.e., the protection level must be equal or higher than the risk level). Thereafter the tool compares the costs of the countermeasures and selects one depending of its costs and its level of protection. Since countermeasures may contain vulnerabilities themselves, the analysis iterates the third and forth phases. If the newly calculated risks can be accepted as bearable, the analysis terminates. Otherwise, new countermeasures are suggested and further iterations take place.

5. Information Flow Analysis

In order to apply the object-oriented security analysis approach for evaluation of security flows of component-structured software, we had to develop new class libraries and graph rewrite systems. The five classes *Component*, *Channel*, *Data*, *DataStruct*, and *Permission* were added to the class library for the design of system models. These classes are multiply inherited from basic application resp. data classes (cf. Sec. 4) and from a new class *InfoFlow*. They inherit the properties describing security levels from the basic classes while *InfoFlow* introduces two new properties modeling an acts for-hierarchy and a label (cf. Sec 3). Moreover, the class library is extended by the association classes *Sends*, *Receives*, *Transfers*, *Stores*, *Permits*, and *Contains* specifying relations between objects.

Fig. 2 delineates an example system for the management of patient records in a hospital which is explained in Sec. 6. It consists of four software components specified by the objects *HospDB*, *Declass*, *D1*, and *D2* of class *Component*. Objects of class *Channel* model simplex information channels between two components (e.g., the object *HospDB-Declass* models information flow from *HospDB* to *Declass*). The channel object is linked with the transmitting component by an edge of association type *Sends* and with the receiving component by a *Receives* association. Data units stored in components or transferred via channels are specified by objects of class *Data*. In particular, one can define data structures utilizing instances of the class *DataStruct* which is inherited from *Data*. The data objects

[3] The risk level 0 states that no risk is assumed and the risk object is removed.

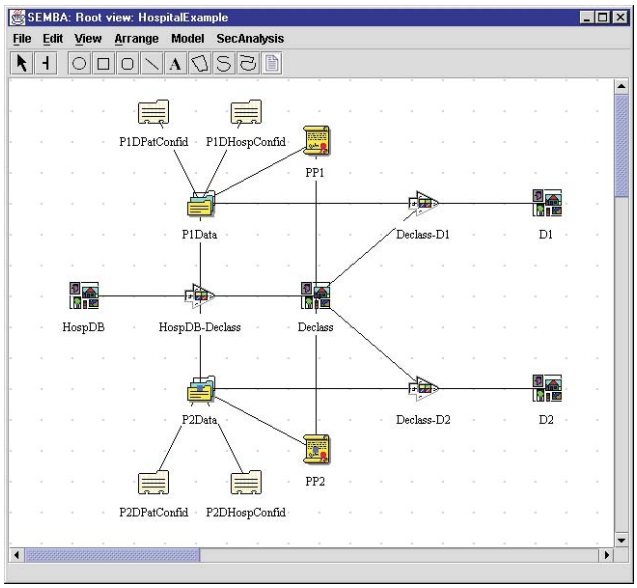

Figure 2. Patient records management example

forming the data structure are linked to the *DataStruct* instance by edges of type *Contains*. In Fig. 2, for instance, *P1Data* models a data structure consisting of the data objects *P1DPatConfid* and *P1DHospConfid*. A data or data structure object may be associated to a channel by an edge of class *Transfers* modeling that the data unit is transferred via this channel. Likewise, it may be linked to a component by a *Stores* association stating that the component stores the data. Another class inherited from *Data* is *Permission* which contains certain rules for declassifying data units (cf. Sec. 3). It is linked with the data object to be declassified by an edge of class *Permits*. In our example the object *PP1* is a permission object stating that the component *Declass* may declassify the data structure *P1Data* according to its declassification rules.

After system design one defines the security levels and the read access policies of the system assets as well as the declassification rules of the permission objects by setting corresponding object properties. Moreover, one fixes the acts for-hierarchy which is modeled by identical property settings in all objects. We assume that this hierarchy is fixed during the whole security analysis process. Since, according to [32], hierarchy revocations occur only infrequently, after each change a new analysis can be expected. This corresponds to the fact that a security analysis has to be repeated from time to time in order to react to new attack strategies. As an example, Fig. 3 shows the property of object *PP1*. Besides the object name one sets the security levels for confidentiality, integrity, and availability objectives to a number between one and seven. For the

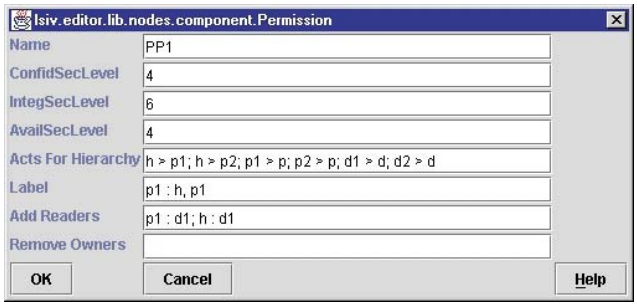

Figure 3. Properties of object *PP1*

particular domain, however, only the confidentiality level is used. The acts for-hierarchy is defined in the next property in which the operator $\succeq$ is represented by the symbol $>$. Beneath, one can set the label of object *PP1*. In contrast to other objects, instances of class *Permission* contain two further properties for defining declassification policy rules. By *Add Readers* the assignment of new readers in sub-labels is specified. In this example, the component *Declass* storing the permission may add the reader $d1$ to sub-labels with owners $p1$ or h in the label of object *P1Data*. The deletion of sub-labels is modeled by the property *RemoveOwners* which contains a list of principal identifiers. Sub-labels owned by list members may be deleted.

The instantiation of these properties, moreover, is supported by two graph rewrite systems. The user has to set the confidentiality level for data objects only. By applying rules of one rewrite system the confidentiality levels of the components and channels are set to the maximum security level of the stored or transferred data units. Likewise, the labels need to be adjusted for components and data units only. By the rules of the other rewrite system consistent labels of other system elements are fixed. For instance, the label of a data structure will be set to the *Join* of the data objects defining the structure (cf. Sec. 3).

The vulnerabilities and threats to the assets are modeled by a second class library containing two different types for describing vulnerabilities. The one class indicates inconsistent label settings which may cause eavesdropping of information. For instance, a component C may store a data unit D with a stricter read access policy than C itself. Thus, at least one principal may exist which may access C but not D. By accessing C this principal, however, may read the stored data unit D, too. Vulnerabilities due to inconsistent label settings are modeled by the class *IncorrectLabels* and inherited classes describing certain kinds of inconsistencies. The other vulnerability describes potential information leaks due to malicious component behavior. This vulnerability is assigned to components containing data which must not be transferred to each successor component. It is specified by an instance of class *IncorrectInfoFlow*. A vulnerability of an asset leads to a threat on the confidential-

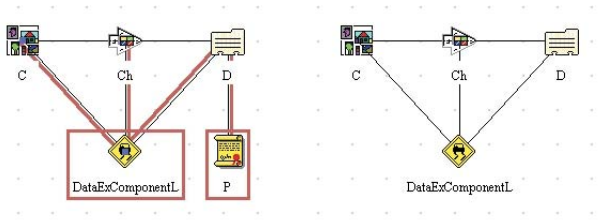

Figure 4. Pre- and post-pattern of a graph rewrite rule

ity of data which is modeled by objects of class *InformationLeakOut*. Moreover, the class contains two association classes. Edges of type *To* are used to describe the relation between a vulnerability and the corresponding asset while the link between vulnerabilities and threats are specified by edges of the class *Exploiting*.

The vulnerability and threat analysis is automatically performed by application of a further graph rewrite system. With respect to incorrect label settings one has to prove that each data unit D in the system is propagated to components and channels with read access policies which are at least as strict than that of D. Thus, the label of each system unit passed by D has to be a relabeling of the data unit label. If D, however, is assigned with a permission P enabling its declassification, the test is more subtle. The analysis has to check if D passed already a component permitted to classify it. Then the relabeling check has to include the label of D relaxed by the classification rules defined by P. The check for potentially malicious components is also performed by relabeling tests. Here, for each — possibly declassified — data unit D passing a component C one has to examine whether the labels of all successor components of C are relabelings of the label of D. If the result of this examination is false, the component is dangerous and the tool indicates it by an object of type *IncorrectInfoFlow*.

As an example, Fig. 4 outlines the pre- and post pattern of one rule of the rewrite system. The rule states that a label setting is wrong if the label of data unit D is stricter than the label of a component C receiving D from a transfer via channel Ch. The scenario of C, Ch, and D is listed in the pre-pattern on the left side of the figure. Moreover, this pre-pattern contains some inhibitory objects and edges which must not be in the subgraph mapping this pattern. In particular, D must not be associated to a permission P since declassified data is modeled by other rules. Furthermore, the scenario must not contain the vulnerability object to be added in order to avoid iteration of this rule. The right side of Fig. 4 delineates the post-pattern in which the vulnerability object *DataExCompomentL*, an instance of a class inherited from *IncorrectLabels*, and the corresponding edges are enclosed to C, Ch, and D. In addition, the rule contains an application condition avoiding its execution if the label of C is a relabeling of D.

Afterwards, one evaluates the seriousness of the vulnerabilities and threats in the system model which is performed by a graph rewrite system, too. Incorrect label settings refer to an inconsistent security policy which may easily be exploited by malicious intruders. Therefore, the seriousness of instances of class *IncorrectLabels* and inherited classes are always rated to the maximum value 7. In contrast, the likelihood of a successful attack caused by a malicious component depends on the selected countermeasures. According the regulations of the CC [21] the property is set to 1 if the component was scrutinized by a source code analysis. In case of a byte code verification we apply the value 2 while due to the danger of steganography the application of security wrappers leads to a seriousness value of 4. Finally, the application of components without any protection leads to the setting 7.

The creation of risk objects and the calculation of the risk level according to the matrix in Tab. 1 is also performed by a graph rewrite system. Moreover, rules are available which may be combined to a rewrite system fulfilling a certain risk assessment policy (e.g., the acceptance of all risks with a level of 1 or 2).

Countermeasures are imposed in two different ways. In order to get rid of vulnerabilities due to incorrect label settings, one applies a graph rewrite system making system unit labels stricter. For instance, the rewrite system reacts on the object *DataExComponentL* created by the rule outlined in Fig. 4 by replacing the label of component C by the Join of the previous label of C and of the label of data unit D. Due to the definition of Join the new label is a relabeling of the label of D and therefore fulfills the read access policy of D.

To reduce vulnerabilities due to malicious components, a countermeasure class library was designed. Currently, it consists of the classes *SecurityWrapper*, *ByteCodeVerification*, and *SourceCodeAnalysis*. The protection level properties of class instances are initialized according to rules of the CC evaluation assessment levels while the initial cost settings are estimated. For the security wrapper both properties are set to 3 reflecting the low deployment costs and the danger of Steganography. In contrast, the properties of the byte code verification, which is more laborious and therefore more expensive but provides a higher degree of protection, are rated to 5. Finally, for the source code analysis the protection value is set to 6 since disguising of information flows is harder in source code than in compiled byte code. Due to the usually enormous source code costs for commercial software the costs are rated to 7, here. In special cases (e.g., freely available source code) these values may be changed manually. Moreover, for this domain two further graph rewrite systems were developed facilitating the selection of suitable countermeasures with a sufficient

protection level as well as selecting a cost-efficient countermeasure.

In order to realize support for information flow analysis of component-based software, we developed 21 classes by inheriting already existing classes. Moreover, we created nine graph rewrite systems consisting of altogether 45 rewrite rules. The total design time including programming the operators of the decentralized labeling model amounted to eight working days for a single person.

6. Example Application

As an example to clarify our approach we use the patient records managing system delineated in Fig. 2 which is a simplified version of the example outlined in [32]. In this system the basic idea is to prevent that every physician in the hospital may read the record of each patient. Instead, a doctor shall have access only to the records of patients treated by herself. To specify this security policy, the model uses the following principal roles: h refers to the hospital owner, $p1$ and $p2$ to two single patients, p to the set of all patients, $d1$ and $d2$ to separate doctors, and d to the set of all doctors. These roles are related by the acts for-hierarchy $h \succeq p1; h \succeq p2; p1 \succeq p; p2 \succeq p; d1 \succeq d; d2 \succeq d$. The hierarchy elements $h \succeq p1$ and $h \succeq p2$ reflect that the hospital owner h keeps track of the records in behalf of the patients and therefore must be allowed to act for each patient. The other elements state that p describes the set of all patients and that each single patient may act for this role. Likewise, each doctor may act for the set of all doctors d.

The patient records of $p1$ and $p2$ are modeled by the data structures *D1Data* resp. *D2Data*. *D1Data* consists of the data units *P1DPatConfid* and *P1DHospConfid*. *P1DPatConfid* models a patient record part which, before selecting a treating doctor, may be only read by patient $p1$ himself and the hospital owner h (e.g., $p1$'s medical record), while data unit *P1DHospConfid* describes confidential information which may be accessed by h only. *P1Data* is associated to a permission object *PP1* facilitating its declassification in order to make the patient record readable for $p1$'s particular doctor (in this example $d1$). Similarly, the patient record *D2Data* of patient $p2$ is composed from the data units *P2DPatConfid* and *P2DHospConfid* and may be accessed by doctor $d2$ due to the declassification policy stated in permission *PP2*. The system is coupled from four software components: The patient records are stored in the database component *HospDB*. The assignment of patient records to particular doctors in order to enable treatment of patients is realized by the component *Declass*. In order to fulfill this task, *Declass* has to declassify the patient records and therefore stores the permission objects *PP1* and *PP2*. The components $D1$ and $D2$ allow the doctors $D1$ and $D2$ to read patient records. The transfer of patient records from *HospDB* via *Declass* to $D1$ or $D2$ is modeled by the channels *HospDB-Declass*, *Declass-D1*, and *Declass-D2*.

After designing the system model, one has to define component and data unit labels, data unit confidentiality security levels, and declassification rules in the permissions. The access policy to the patient records is constrained by the patients themselves as well as by the hospital owner. The patient requires that his record may be accessed only by h and himself and therefore adds the sub-label $p1 : h, p1$ to the objects *P1DPatConfid* and *P1DHospConfid*. The hospital prevents other principals reading the hospital confidential part of the record by attaching sub-label $h : h$ to the label of *P1DHospConfid*. Likewise the labels of the components of $p2$'s patient record are set to $p2 : h, p2$ resp. $p2 : h, p2; h : h$. The permission *PP1* is owned by $p1$ who enables access for h and himself (cf. label $p1 : h, p1$ in Fig. 3). Similarly, permission *PP2* carries the label $p2 : h, p2$. The four components are supervised by the hospital owner. *HospDB* and *Declass* are not intended to print out information to anybody. Therefore reading access is only granted to h itself for maintenance (label $h : h$). In contrast, the components $D1$ and $D2$ are used for the output of patient records. Therefore, the label of $D1$ is set to $h : h, d1$ and the label of $D2$ to $h : h, d2$. The labels of the data structures and channels are added automatically by a graph rewrite system. While the channels are set to the labels of the components linked by an edge of type *Receives*, the data structures *P1Data* and *P2Data* carry the label $h : h$ due to the stricter policy of the hospital owner.

The confidentiality security levels of the patient records are set to the value 4. This reflects that on the one hand eavesdropping of patient data does not lead to a break down of the hospital operations. On the other hand, however, the leaking of patient data is a serious privacy violation leading to costly compensations and distrust in the hospital. Weighing up the aspects, we consider a high to moderate valuation (level 4) as appropriate. By applying graph rewrite rules the value 4 is also set to the confidentiality security properties of the components and channels.

Finally, we have to fix the declassification rules of the permission. The patient record of $p1$ is constrained by h as well as $p1$. Since $p1$ accepts $d1$ as the doctor treating him, both record owners accept $d1$ as a reader and the *Add Readers* property of *PP1* is set to $p1 : d1; h : d1$ (cf. Fig. 3). Likewise, in *PP2* this property is set to the value $p2 : d2; h : d2$.

In the next step we analyze the system model for vulnerabilities and threats. In order to show incorrect label settings to the reader, we replaced the correct labels of component $D2$ and channel *Declass-D2* by wrong labels $h : h, d2, p$. Thus, at $D2$ access is mistakenly enabled to any patient. Fig. 5 delineates the distorted system model augmented by vulnerability and threat objects. As expected, the objects

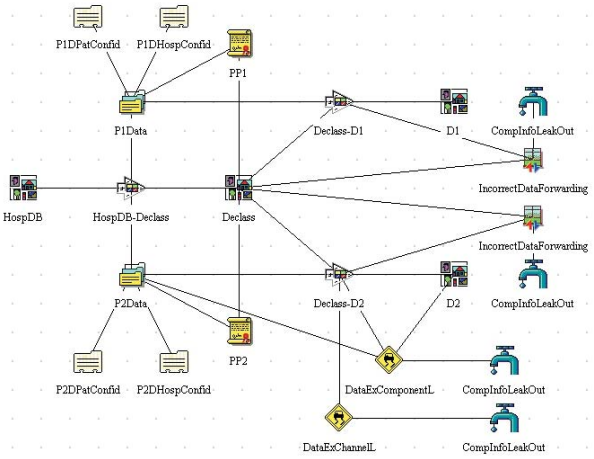

Figure 5. Example system with vulnerabilities and threats

DataExComponentL and *DataExChannelL* refer to the erroneous labels of component $D2$ and channel *Declass-D2*. Similarly, the two objects *IncorrectDataFowarding* indicate that component *Declass* may leak information intended to $D1$ by transferring it to $D2$ and vice versa. Each vulnerability object is associated to a threat object *CompInfoLeakOut* referring to the threat of leaking information. The seriousness of the vulnerability and threat objects is set to the maximum value 7 since no countermeasures are integrated, yet.

Thereafter, for each vulnerability a separate risk object is generated. According to the matrix in Tab. 1 the risk levels of all objects are set to the value 5. Since we want to accept only risks of levels 1 and 2, we have to continue the security analysis with planning countermeasures. With respect to the incorrect label settings, we apply a graph rewrite system which replaces the wrong labels of $D2$ and *Declass-D2* with the correct label $h : h, d2$ again. In order to reduce the information flow policies, we apply another rewrite system suggesting countermeasures. Since the protection level of a security wrapper for a component with risk level 5 is too low, the tool suggests byte code verification and source code analysis to guard against malicious behavior of component *Declass*. As byte code verification, moreover, is the less expensive safeguard, it is selected.

In the next iteration, the two vulnerability objects *IncorrectDataForwarding* are generated again. Due to the countermeasure, however, their seriousness values are now set to 2. Likewise, the levels of the corresponding risks are assigned with 2 which, according to our security policy, can be accepted as remaining risks. Thus, the information flow analysis will be terminated now and we have to perform a byte code verification of component *Declass*. In case of using Java Beans-based components, this verification is facilitated by powerful tool-support and can be performed with an acceptable expenditure.

Due to the extensive tool-support the specification and analysis of the example system could be performed in a couple of minutes.

7. Concluding Remarks

We reported on the application of object-oriented modeling and graph rewriting for the highly automated information flow control of component-structured software. The corresponding tool-support is well-suited to real-life systems since it supports UML-based description techniques for complex systems (e.g., hierarchical system models hiding subsystems in folders which are modeled by separate UML-object diagrams). Currently, more work is devoted to extensions of the class libraries and rewrite systems and, in particular, to the consideration of trust between principals which can be realized by applying trust management approaches (e.g., [8, 9, 22]). For instance, in order to determine the seriousness of a threat due to a malicious component leaking information, one considers not only the applied countermeasures but also the trust into the component designer.

Moreover, one has to consider integrity and availability aspects of component-structured software, too. Here, we will augment the component contracts by specification modules in a temporal logic-based modular specification and verification technique [17]. The contract specifications of all components may be composed to a system interface specification. Since the security policy can be modeled based on the same specification technique, one can prove formally that the combined interface specification fulfills the security policy. Moreover, one has to check that the real component behavior complies with the contract specification which can be performed, for instance, by security wrappers [19].

References

[1] M. Abadi. Secrecy by Typing in Security Protocols. In *Proc. 3rd International Conference on Theoretical Aspects of Computer Software*, 1997.

[2] E. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, New Jersey, 1993.

[3] R. J. R. Back and R. Kurkio-Suonio. Decentralization of process nets with a centralized control. *Distributed Computing*, (3):73–87, 1989.

[4] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of graph transformation to visual languages. In *Handbook on Graph Grammars and Computing by Graph Transformation, Volume 2*, chapter 1. World Scientific, 1999.

[5] R. Baskerville. *Designing Information Systems Security*. Wiley & Sons, Chichester, 1988.

[6] R. Baskerville. Information Systems Design Methods: Implications for Information Systems Development. *ACM Computing Surveys*, 25(4):375–414, Dec. 1993.

[7] D. E. Bell and L. J. LaPadula. Secure Computer System: Mathematical Foundations. Technical Report MTR-2547, MITRE Corporation, Bedford, Mass., 1973.

[8] T. Beth, M. Borcherding, and B. Klein. Valuation of Trust in Open Networks. In *Proc. European Symposium on Research in Security (ESORICS)*, LNCS 875, pages 3–18, Brighton, 1994. Springer-Verlag.

[9] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The Role of Trust Management in Distributed Systems Security. In *Internet Programming: Security Issues for Mobile and Distributed Objects*. Springer-Verlag, 1999.

[10] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[11] CCTA. *SSADM-CRAMM Subject Guide for SSADM Version 3 and CRAMM Version 2*. CCTA, London, 1991.

[12] R. Courtney. Security Risk Assessment in Electronic Data Processing. In *AFIPS Conf. Proc. National Computer Conference 46*, pages 97–104, Arlington, 1977. AFIPS.

[13] W. Farmer, J. Guttman, and V. Swarup. Security for Mobile Agents: Issues and Requirements. In *Proc. 19th National Information Systems Security Conference (NISSC 96)*, pages 591–597, 1996.

[14] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 130–140, Oakland, 1997.

[15] R. Graubart. On the Need for a Third Form of Access Control. In *Proc. 12th National Computer Security Conference*, pages 296–303, Gaithersburg, MD, 1989.

[16] N. Heintze and J. G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'98)*, San Diego, 1998.

[17] P. Herrmann and H. Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[18] P. Herrmann and H. Krumm. Object-oriented security analysis and modeling. In *Proc. 9th International Conference on Telecommunication Systems — Modeling and Analysis*, pages 21–32, Dallas, 2001. ATSMA, IFIP.

[19] P. Herrmann and H. Krumm. Trust-adapted enforcement of security policies in distributed component-structured applications. In *Proc. 6th IEEE Symposium on Computers and Communications*, pages 2–8, Hammamet, 2001. IEEE Computer Society Press.

[20] P. Herrmann, L. Wiebusch, and H. Krumm. Tool-Assisted Security Assessment of Distributed Applications. In *Proc. 3rd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 2001)*, pages 289–294, Krakow, 2001. Kluwer Academic Publisher.

[21] ISO/IEC. *Common Criteria for Information Technology Security Evaluation*, 1998. International Standard ISO/IEC 15408.

[22] A. Jøsang. An Algebra for Assessing Trust in Certification Chains. In J. Kochmar, editor, *Proc. Network and Distributed Systems Security Symposium (NDSS'99)*. The Internet Society, 1999.

[23] G. Karjoth, D. Lange, and M. Oshima. A Security Model for Aglets. *IEEE Internet Computing*, pages 68–77, July/August 1997.

[24] D. M. Kienzle and W. A. Wulf. A Practical Approach to Security Assessment. In *Proc. Workshop New Security Paradigms '97*, pages 5–16, Lake District, 1997.

[25] B. W. Lampson. Protection. In *Proc. Princeton Conference on Information and Systems Sciences*, 1971.

[26] J. Leiwo, C. Gamage, and Y. Zheng. Harmonizer — A Tool for Processing Information Security Requirements in Organization. In *Proc. 3rd Nordic Workshop on Secure Computer Systems (NORDSEC'98)*, Trondheim, 1998.

[27] C. J. McCollum, J. R. Messing, and L. Notargiacomo. Beyond the Pale of MAC and DAC — Defining New Forms of Access Control. In *Proc. IEEE Symposium on Security and Privacy*, pages 190–200, Oakland, CA, 1990.

[28] M. D. McIlroy and J. A. Reeds. Multilevel Security in the UNIX Tradition. *Software — Practice and Experience*, 22(8):673–694, 1992.

[29] Microsoft. The Microsoft COM Technologies. available via WWW: http://www.microsoft.com/com/comPapers.asp, 1998.

[30] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, San Antonio, 1999.

[31] A. C. Myers and B. Liskov. A Decentralized Model for Information Control Flow. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, 1997.

[32] A. C. Myers and B. Liskov. Complete, Safe Information with Decentralized Labels. In *Proc. IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, 1998.

[33] Object Management Group. CORBA Component Model Request for Proposals, June 1997.

[34] Object Management Group (OMG). *A Discussion of the Object Management Architecture*, 1997.

[35] Object Management Group (OMG), CORBA. *Security Services Specification, Version 1.5*, 2000.

[36] J. Palsberg and P. Ørbaek. Trust in the λ-Calculus. In *Proc. 2nd Internation Symposium on Static Analysis*, LNCS 983, pages 314–329. Springer-Verlag, 1995.

[37] T. Quatrani. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley, 2 edition, 2000.

[38] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, pages 38–47, Feb. 1996.

[39] A. Stoughton. Access Flow: A Protection Model which Integrates Access Control and Information Flow. In *Proc. IEEE Symposium on Security and Privacy*, pages 9–18, Oakland, CA, 1981.

[40] Sun Microsystems. Java Beans Specification. available via WWW: http://java.sun.com/beans/docs/spec.html, 1998.

[41] Sun Microsystems, Palo Alto. *Infobus 1.2 Specification*, 1999.

[42] C. Szyperski. *Component Software — Beyond Object Oriented Programming*. Addison-Wesley, 1997.

[43] Tigris. *ArgoUML Vision*, argouml.tigris.org/vision.html, 2000.

[44] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):167–187, 1996.