Trust-Based Monitoring of Component-Structured Software[1]

Peter Herrmann and Heiko Krumm
University of Dortmund, Computer Science Department
Peter.Herrmann@udo.edu, Heiko.Krumm@udo.edu

Abstract

In contrast to traditional software, component-structured systems are developed by combining independently designed and sold software components. This technology promises an easier reuse of software building blocks and, in consequence, a significant reduction of the efforts and costs to produce software applications. On the other side, component-structured software is subject to a new class of security threats. In particular, a maliciously acting component may easily spoil the application incorporating it. In this paper we introduce an approach addressing this particular threat. A so-called security wrapper monitors the events passing the interface of a component and checks them for compliance with formally specified security policies guaranteeing a benevolent behavior of the checked component. We introduce the layout and functionality of the wrappers and outline the formal security specifications which can be easily derived from a set of specification patterns.

Unfortunately, the security wrappers cause runtime overhead which, however, can be significantly reduced by relaxing the degree of monitoring trustworthy components. In order to support the decision, whether a component can be trusted, we developed a special trust information service. This service collects evaluation reports of a particular component running in various applications which are provided by the different security wrappers. Based on the evaluation reports, the trust information service computes a so-called trust value which is delivered to the security wrappers, and a wrapper adjusts the degree of supervision of a component based on its trust value.

The use of the security wrappers as well as of the trust management approach is clarified by means of an e-commerce example realizing the automated procurement of goods for a fast-food restaurant.

1. Introduction

Component-structured software gets more and more popular since it can be developed with lower development efforts and reduced costs than traditional monolithic applications (cf. [Szyp97]). In order to build a component-based system, various components each realizing a subfunction of the desired application (e.g., the stock management, billing, delivery, etc. of an enterprise resource planning system) are selected from a free market. A so-called assembler composes the components to the resulting application. The overall system may be either executed on a single computer or the components may reside on various hosts which are coupled by means of a telecommunication system.

Since the software components are designed separately and the creator of a component does not know the interfaces of other components to be coupled with his component, the syntax and semantics of the component interfaces may be incompatible. In this case, the assembler creates special adapters merging different components. To facilitate the composition of

[1] Accepted for PIK, 2004.

components and the design of adapters, the component designer provides a component with a component contract. This contract is ideally legally binding (cf. [Szyp97]) and states all context dependencies of the component explicitely. According to Beugnard et al. [BJPW99], a contract should consist of four parts describing the static structure of the component (i.e., the methods, events, and exceptions provided resp. used by the component), the desired behavior of the component and its environment, synchronization aspects, and quality of service properties.

Meanwhile, four plattforms for component-structured software exist: Sun Microsystems developed both the Java Beans [Hami97] and the Enterprise Java Beans (EJB) [DeMi03]. Microsoft offers the (Distributed) Component Object Model (COM, DCOM, COM+) [Micr98] which is a part of their software architecture .NET. Finally, the OMG extended their middleware plattform CORBA with a component model [OMG02], too.

Today, the EJBs which are optimized for the creation of client/server software are the most popular plattform (cf. [Giga01]). Most EJB-based applications, however, are created from components built by only one company (e.g., [Bea03, IBM, JBo]). Thus, the advantage of heterogenous systems, that one always selects the component realizing a desired system function best, is not utilized. According to software development practitioners, the hesitation to use heterogeneous components is mainly based on the fear of a new class of security threats. These threats result from the larger number of principals involved in the design and deployment of the heterogenous systems which includes not only system owners, users, and host providers but also the different component designers and the assemblers. The higher number of participants, however, increase the likelihood that one of them intends to derive an improper advantage by behaving in a malicious manner. Since it is also more difficult to detect the true culprit of an attack, the principals and, in particular, the application owners, have a lower mutual trust in each other than in the case of monolithic or of homogeneous component-based software.

We detected three major threats which have to be tackled in order to rise the acceptance of heterogenous component-based software:

- Threats for component-structured systems based on maliciously acting member components,
- threats for component-structured systems due to a wrong coupling of components leading to new vulnerabilities, and
- wrong incriminations of component designers.

In this paper, we introduce an approach protecting applications from attacks by member components. It is based on the component contracts outlined above which are amended with descriptions of security policies to be guaranteed by the component designer. The security policies state certain constraints of behavior at the interface of a component in order to prevent malicious attacks on environment components. We use a formal model based on state transition systems constraining the events[2] triggered by a component in dependence of the order of preceding interface events. The applied specification technique is the linear-time temporal logic cTLA [HeKr00a] enabling modular specifications of various security constraints in separate specifications.

At design time, the application owner can check if the security policies of the components fulfill the overall security goals of the application embedding them. Since the application

[2] Below, we restrict ourselves to events as the only elements of component interfaces since other elements like method calls or exceptions can easily be realized by interface events as well.

security goals are also specified in cTLA, these checks can be achieved by logical deduction proofs (cf. [Herr03b]).

This publication concentrates on another type of inspections which are performed during runtime. The component security policies are defined by the component designers who might easily disregard malicious behavior of a component in its policies. Therefore the application owner has to check if the real interface behavior of a component complies with its security policies. This task is performed by special security wrappers [HeKr01, HeWK02] which are inserted at the interfaces between components and their environments. A security wrapper temporarily blocks all events passing a component interface and checks them for compliance with the cTLA-based security policy specifications of the corresponding component. If an event complies with the policies, it is forwarded to its destination. If it, however, violates a security policy, the component is blocked by refusing to forward further events and the application administrator is notified.

The component security policy specifications have to be designed in a way that, on the one hand, they are sufficiently detailed in order to detect all security-relevant deviations of the desired component behavior. On the other hand, one has to avoid that the specifications are as complex as reference implementations of the component which would cause an unfounded specification development and inspection effort. To facilitate the design of suitable security policies of the right granularity, we therefore developed a framework of cTLA specification patterns from which policy specifications can be created in a relatively easy fashion. The patterns guarantee that relevant security aspects of a component can be described in a concise way. Moreover, the patterns address typical vulnerabilities of component-based systems in order to facilitate logical proofs at design time.

A drawback of the security wrapper approach is the performance overhead of up to 10%. This can be reduced by restricting the checks to components which are not fully trusted by their owner. These components are either relatively new and therefore still unknown or they already proved malicious behavior deviating from their security policies. In order to supply application owners with a means to create well-founded trust to a component, we complement the security wrappers by a trust information service and by trust managers [Herr01a, Herr03a]. Similar to reputation systems (cf. [RZFK00]) like the feedback forum of the internet auction service eBay [eBay], the trust information service collects evaluation reports on a component issued by principals running the component in various applications. In each application, a trust manager links the local security wrapper with the trust information service. According to the behavior of a component, the trust manager sends a positive or negative report to the trust information service which, in intervals, computes trust values (cf. [Jøsa96]) from the reports of the different trust managers. The trust values are sent to the trust managers which adjust the degree of enforcement according to a special trust management policy. The monitoring a particular component may reach from a complete observation of all events, if the trust in the component is fairly low, via spot checks till the termination of the enforcement if the component is fully trusted.

In the next section, we give a survey about alternative approaches protecting component-based systems against attacks by member components. Thereafter, we outline the architectures of the trust management approach as well as the security wrapper and sketch the framework of security policy patterns. Afterwards, we introduce an e-commerce application performing the automated procurement of goods for fast-food restaurants. To clarify the enforcement and trust management approaches, we finally utilize the example application by showing the security policies of its relevant component as well as the trust value-based enforcement strategy of its trust manager.

2. Related Work

Since still only few safeguards against the threats listed above exist, we will look at first on mechanisms addressing the threats against mobile code systems which are similar to those of component-based software. Besides of man-in-the-middle attacks against agents on the transmission between computers, mobile agents have mainly to be protected against attacks from the hosts executing the agents and vice versa. A popular method to guard a host against a malicious agent is running the agent in a protected system kernel (cf. e.g., [BSP+95]). Other safeguards guaranteeing secure control flow, memory access, and method calls are centered on the instrumentation of the agent code (cf. [Koze99]). Here, the code is modified in order to enable design time or runtime checks for unexpected behavior which points to a malicious attack. Schneider [Schn97] as well as Biskup and Eckert [BiEc94] model security policies of agents by special automata. Moreover, an agent is executed in a special runtime environment performing an operation only if it complies the automata. In contrast to our security wrappers, not only the interface events but all code operations are checked which leads to complex model descriptions and to a significant runtime overhead.

Another code instrumentation technique is programming language-based security. Here, special parsers retrieve security relevant data from the source code of an agent which are used to check if the code complies with the security policies of the agent. A well-known example is the Java byte code verifier (cf. e.g., [NiPe97]) checking Java programs for type correctness and other security-relevant properties. Other approaches are based on Proof Carrying Code (cf. [Koze99]) which contains formal specification elements like pre- and post-conditions or loop invariants. The specification elements can be used for formal proofs that the program code fulfills certain properties. Examples are the Touchstone Compiler [Necu98], the Efficient Code Certification [Koze98] as well as approaches for type checking [TMC+96, MWCG98] and information flow analysis [FSBJ97, MyLi98, Myer99].

Since, like in our approach, a formal specification is added to the Proof Carrying Code by the code developer, one has to check at runtime if the code complies with the formal specification elements, too. This is also performed by software wrappers which mostly are integrated into firewalls or intrusion detection systems (cf. [Monr93, AvRa94, GWTB96]). Similar to us, Fraser et al. [FrBF99] use software wrappers to protect components against malicious attacks but restrict themselves to a special class of system calls.

Like us, Khan et al. [KhHZ01] utilize the component contracts to check security aspects. They concentrate on the modeling of assume-guarantee conditions between two coupled components. While they propose a solution for the threat of coupling components in a wrong way (cf. Sec. 1), their model is too simple to specify more complex component-based application security properties. These more complex property specifications, however, are needed to guard against the threat of malicious member components.

An alternative to inspections performed by the code users are certifications carried out by trusted third party authorities which check components for various security properties and issue certificates if all tests are passed. Voas [Voas98] recommends black box testing [MiFS90], fault injection [VMGM96], and operational runtime tests [WLAG93] as security and functional tests for software components. Moreover, one can use source code inspections and, if the source code is interpreted (e.g., Java programs), byte code verification as certification methods.

3. Trust Information Service

The trust information service facilitates trust management of component-structured software by collecting evaluation reports and passing a summary of the reports to interested application owners. In order, to enable automated trust-based runtime enforcement of components, the summary of the reports is specified as a trust value of a component (cf. [Jøsa96]) which is put to the disposal of the application owners and other interested parties. Based on them the application owners decide about the degree of monitoring a component. Moreover, somebody interested in buying one of a group of competing components may consider their trust values for his procurement decision.

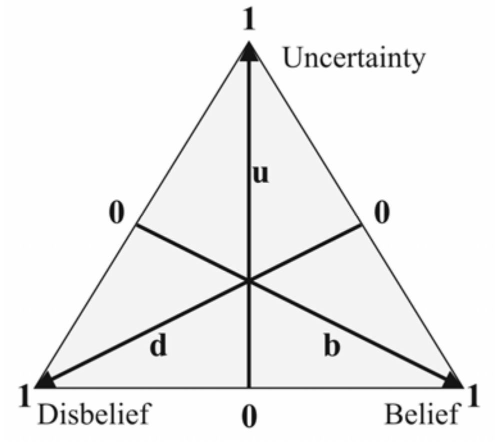

Figure 1: Opinion Triangle (taken from [Jøsa99])

A trust value is a mathematical formula describing the belief resp. the disbelief in a component. Furthermore, one can state that the number of experience reports is too low to give a decent evaluation. In [Jøsa99], Jøsang introduces the so-called opinion triangle (cf. Fig. 1) as a trust value model. Belief, disbelief, and uncertainty are specified by means of the values *b, d*, and *u*, each being a real number in the interval [0,1]. Since the side constraint b + d + u = 1 holds, a trust value can be depicted by a point in the triangle. A point close to the top of the perpendicular states uncertainty about a component due to a low number of reports. If more reports are received, the uncertainty value decreases. Then points on the right or left side describe belief resp. disbelief in the component.

The trust values are computed from the numbers *p* of positive and *n* of negative evaluations by applying certain metrics. Jøsang and Knapskog [JøKn98] calculate the values *b*, *d*, and *u* of the opinion triangle from *p* and *n* by applying the following formulas:

$$b = \frac{p}{p+n+1} \qquad d = \frac{n}{p+n+1} \qquad u = \frac{1}{p+n+1}$$

This metric expresses a relatively tolerant trust management philosophy since a negative experience report can be compensated by a sufficient number of positive reports. In contrast, the metric of Beth et al. [BeBK94] expresses an unforgiving philosophy. The probability that *b, d*, and *u* exceed certain values is stated by the formulas

$$b = \begin{cases} 1-\alpha^p; & n=0 \\ 0; & n>0 \end{cases} \qquad d = \begin{cases} 0; & n=0 \\ 1; & n>0 \end{cases} \qquad u = \begin{cases} \alpha^p; & n=0 \\ 0; & n>0 \end{cases}$$

According to this metric only a single negative experience report destroys the belief in a component forever. If all reports are positive, *b* is increased in dependence on *p* and the constant α. α is freely selectable within the open interval (0,1) and determines the rate of increase of b for a given number of positive reports. The two metrics describe two different ways to gain trust and the decision which metric to select in order to adjust the observation of components should be guided by the damage, a successful component attack has on the institution running the malicious component. To support different enforcement policies, the trust information service offers two trust values for each component computed based on the both metrics listed above.

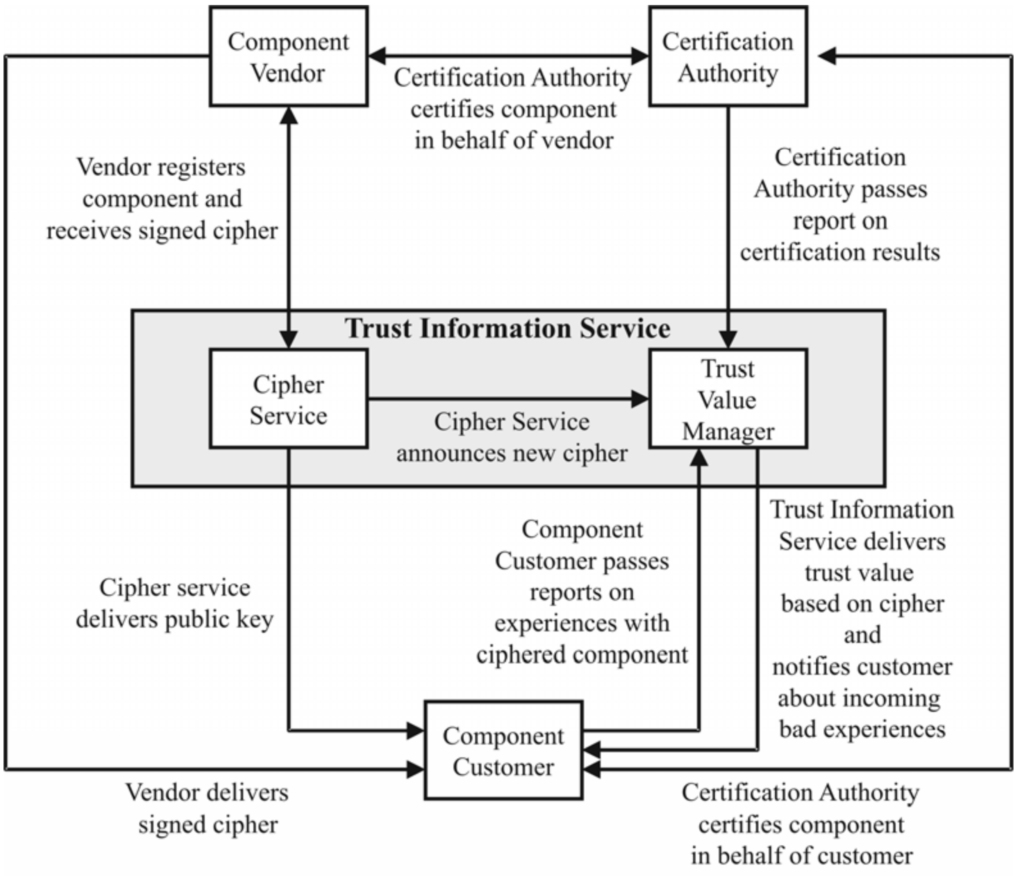
Component Vendor
Certification Authority
Certification Authority certifies component in behalf of vendor
Vendor registers component and receives signed cipher
Certification Authority passes report on certification results
Trust Information Service
Cipher Service
Trust Value Manager
Cipher Service announces new cipher
Cipher service delivers public key
Component Customer passes reports on experiences with ciphered component
Trust Information Service delivers trust value based on cipher and notifies customer about incoming bad experiences
Component Customer
Vendor delivers signed cipher
Certification Authority certifies component in behalf of customer

Figure 2: Trust Information Service

The trust information service is delineated in Fig. 2. It is linked with component vendors resp. developers, with interested principals procuring components and monitoring them during runtime, and with authorities certifying components in behalf of vendors or users. Component vendors may register any software component offered commercially or free of charge declaring to agree that experience reports are collected from users and that corresponding trust values are offered to interested parties. Component customers and application owners may inquire trust values from the trust information service at any time. Moreover, they can subscribe to an alarm service which notifies them immediately if another user reported a serious malicious attack caused by a component of interest. To receive as many reports as possible, all application owners are called in intervals to send an experience report on the components of their application. Moreover, if a component user detects a serious malicious attack, he should notify the trust information service immediately which sends alarm messages to the other subscribers of the alarm service. Furthermore, the results of component certifications are considered in the calculation of trust values.

To guarantee a high degree of privacy for the component vendors, the experience reports are separated from the component descriptors. Therefore the trust information service consists of a cipher service and a trust value manager. The cipher service stores the registration data of a component and its vendor and generates a cipher which the vendor hands over to interested component users. The trust value manager stores the experience reports and trust values based on the ciphers without knowing the true identity of a component. Thus, neither the cipher service nor the trust value manager have complete knowledge about a component and its trust values.

Unfortunately, the trust information service is vulnerable against wrong incriminations of components by application owners. A trust manager may send in behalf of its application owner wrong evaluations on a component which may lead to a wrong trust value. To avoid this, we extended the trust information service by an experience report checker. The trust manager is obliged to send a log of component interface events together with an evaluation report and the experience report checker compares the evaluation with the log. If a component user is convicted of forging evaluations, his reports on a component are not considered anymore for the computation of trust values. A detailed description of this extension is introduced in [Herr03a].

4. Runtime Enforcement

The enforcement of the security objectives specified in the component contract models is performed by the security wrappers (cf. [HeKr01, HeWK02]). Figure 3 depicts a wrapper implementation for component-structured software based on Java Beans [Mall00]. Each observed bean is wrapped by an adapter component discerning all events passing the bean interface. The adapter blocks an incoming or outgoing event temporarily and notifies the observer components. Each observer checks the compliance of the event with a component contract model by simulating the model. If the event is accepted by all observers, the adapter forwards it to its destination. If an observer, however, detects a contract violation, the adapter blocks all further events until receiving a release order from the application administrator. Moreover, the adapter notifies the administrator by means of the monitor component which forms the administrator's interface. Another component is the adapter generator which creates adapters automatically by applying the Java introspection mechanism in order to detect the event structure of a bean. Finally, the wrapper utilizes the built-in Java security manager to prevent hidden data channels of a wrapped bean. The security manager permits only events coming from or going to the adapter of an observed bean.

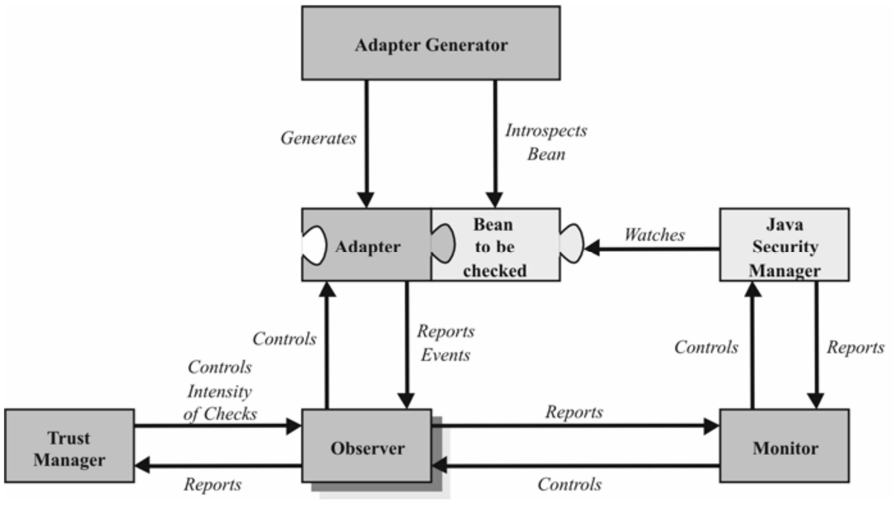

Figure 3: Security Wrapper Architecture and Trust Manager

The link between a security wrapper and the trust information service is formed by a trust manager object (cf. [Herr01a]). A trust manager is inquired by the trust information service in intervals on the experience with a monitored software component. It sends a positive report if the component did not violate its security policy specifications since the transmission of the last report. In contrast, a minor contract model violation of a component leads to a negative experience report while in the case of a major violation the trust manager notifies the trust information service immediately in order to enable an alarm message to other component users. If the trust manager receives an alarm message about a component, it causes the wrapper to notify the application administrator and the adapter to seal the component.

Moreover, a trust manager controls the contract enforcement process. In intervals, the trust manager inquires the current trust value of an observed component from the trust information service and adjusts the enforcement policies accordingly. If the trust value of a component states a high degree of uncertainty or disbelief, the component is permanently fully observed. If the value b indicating the belief in a component exceeds a certain limit, the monitoring is reduced to spot checks where the compliance checks are only performed in times. In this mode, however, the current states of the simulated contract models are adjusted permanently in order to enable the reactivation of the compliance checks. Finally, if a trust value describes a very high belief in a component, the component can be assumed to be correct and good-natured and its security wrapper is removed.

5. Security Policy Patterns

A malicious component may attack the confidentiality, integrity, availability, and non-repudiation of the embedding application in various manners. The confidentiality of a component-structured system and its data is vulnerable if its components are executed on various hosts with different access policies. Altering the flow of information, a component may maliciously send a data unit to a host which grants access to a principal who is not permitted to read the data. Furthermore, a malicious component may create hidden channels where information is either hidden in legitimate data transferred between components (steganography) or where the order, number, or time of triggering a data unit is assigned with a secret meaning. By attacks on the integrity of a component-structured application, the functionality of the system or its data is illegally altered in order to behave in an undesired way. This kind of modifications can be done by triggering erroneous interface events, by using wrong settings of event or method parameters, or by illegally modifying attributes of partner components. The availability of an application and its components can be spoiled in two different manners:

- A service offered by a component to a partner may be called by the partner very often preventing that the called component grants service to third components (denial of service attacks).
- A component blocks a partner component by refusing to trigger a desired event without which the partner cannot offer services to third components anymore.

A non-repudiation attack is performed if a component alters the integrity of an application in a way that the component’s designer or owner will be set to the position to repudiate later successfully that the component triggered or received a certain interface event.

In order to protect a component-based system against attacks from member components, the system owner defines suitable security goals as well as security policies enforcing the goals. A security policy constrains the behavior of a component in a way that an attack can either not be performed at all or only with very high efforts exceeding the potential gain of a successful attack. To prevent confidentiality attacks, the flow of data between two components has to be constrained in order to guarantee that data is transmitted to a component only if all principals who may access the component are permitted to read the data as well (cf. [Herr01b]). The use of hidden channels may be impeded by enforcing a deterministic interface behavior where a component reacts on a certain order of input events in only one possible way and at one possible time (cf. [ZFK+98]). Thus, the component is not able to act in different manners where the selected action may describe hidden information. One may impede integrity attacks by constraining certain interface events and their parameters. Moreover, one can perform plausibility checks of the events during runtime. Denial of service attacks can be prevented by enforcing minimum waiting times between two service calls in order to relax the called component which now can grant its service to other components, too. In contrast, one can avoid that a component refuses to send an event in order to block a partner component by demanding a maximum waiting time until triggering the event. More difficult is the enforcement of policies preventing the later repudiation of events. A solution for this is the application of a trusted third party component collecting digitally signed proofs of events which can act as a witness if the event is later repudiated.

Our security wrappers (cf. Sec. 4) are used to enforce the security policies listed above. Due to their position at the interface of a component, they can only monitor the interface events but neither check internal events of a component nor its internal attributes. Therefore, on the one hand, the security wrappers may constrain the transfer of interface events and their parameters where the history of preceding interface events may be considered. On the other hand, the wrappers may constrain the time when an event is sent by demanding minimum or maximum waiting times. Based on these limitations, we defined four basic specification patterns from which the more specific security policy patterns are derived:

- **Enabling condition**: The enabling of interface events and the argument values of the events are constrained.
- **Enabling history**: The enabling conditions of interface events depend on the context of preceding interface events.
- **Minimum waiting time**: Interface events may only be executed if some minimum waiting time periods elapsed since a preceding event.
- **Maximum waiting time**: Interface events have to occur before a maximum waiting time expired since a preceding event.

The basic specification patterns are refined to patterns specifying confidentiality, integrity, availability, and non-repudiation policies for components. We developed twelve different patterns. Security policies guaranteeing the confidentiality of component-structured software can be modeled by means of the following five policy patterns:

- **Data flow access**: A data unit may only be forwarded to a component if a corresponding read access permission exists.
- **Data flow history**: A data unit may only be forwarded in the context of certain preceding interface events.
- **Hidden channel functional dependency**: A forwarded data unit depends on previously transferred data according to a data dependency function.
- **Hidden channel enabling history**: The enabling condition of an interface event and its arguments depend on the context of preceding events according to an occurrence dependency function.
- **Hidden channel execution time**: An interface event has to be executed after a preceding interface event within a certain time period.

Security policy patterns enforcing the integrity of component-based systems are listed in the following:

- **Integrity enabling condition**: The enabling conditions of interface events and their arguments are constrained in order to guarantee plausible component interaction.
- **Integrity enabling history**: The enabling conditions of interface events and their arguments depend on the context of preceding interface events in order to guarantee plausible component interaction.

The four security policy patterns listed below are used to address the two ways attacking the availability of components:

- **Denial of service minimum waiting time**: An interface event may only be executed if a minimum waiting time period elapsed since a similar event was executed.
- **Denial of service enabling history**: The enabling condition of an interface event depends on the context of certain preceding interface events and additionally on a minimum waiting time period.
- **Blocking maximum waiting time**: An interface event has to be executed before a maximum waiting time period expired since a certain preceding interface event.
- **Blocking enabling history**: According to the context of certain preceding interface events an interface event has to be executed before a maximum waiting time period expired.

Finally, security policies enforcing the non-repudiation of interface events can be modeled by means of the following pattern:

- **Event logging**: A component has to log an executed or received event together with a unique signature with a trusted third party logging service.

The basic specification patterns are modeled in the formal specification technique cTLA [HeKr00a, HeKr00b]. The security policy patterns are derived from the basic specification patterns by utilizing the cTLA composition operator (cf. [HeKr00a]) composing certain instances of the basic patterns to a more specific security policy specification. cTLA facilitates the modeling of system properties like the component security policies in a process-like style similar to high-level programming languages. Moreover, it enables the design of specifications in a constraint-oriented style (cf. [VSvS88]) where special properties (e.g., a mechanism of a telecommunication protocol) are specified by a separate process. The processes may be composed with each other to a system specification modeling the combination of all properties. Thus, different security policies may be specified separately and combined to an overall specification including all policies of the components forming a component-structured application. This specification can be used for a formal cTLA proof that the security policies guarantee certain system security goals (cf. [Herr03b]).

The cTLA processes specify security policies as state transition systems which are simulated by the observers of a security wrapper in order to decide if a certain event can be accepted or not. If the monitored component triggers an event, an observer checks if the transition modeling this event is enabled in the simulated current state. If the transition can be executed, the corresponding event is accepted and the observer sets the state resulting from firing the transition to the new current state of the simulation run. Otherwise, the event is rejected and the security wrapper notifies the application administrator.

The designer of a component defines a security policy for it by instantiating the parameters of one of the twelve security policy patterns. The resulting security policy instance is added to the component contract and may be applied by the component owner for design time security proofs as well as for the runtime checks performed by a security wrapper. For runtime enforcement, the cTLA specification is translated to Java code which is executed by an observer. A more formal introduction to the cTLA-based security policies including specification examples is provided in [Herr03b]. Moreover the various specification patterns can be retrieved from the WWW: http://ls4-www.informatik.uni-dortmund.de/RVS/P-SACS/.

6. Component-Structured E-Procurement Application

The example application (cf. also [HeWK02]) performs automated commodity management of fast-food franchise restaurants. It is based on the standard OBI (Open Buying on the Internet) [OBI99] which defines procedures for electronic procurement (e-procurement) of goods. In particular, OBI standardizes an architecture for procurement activities consisting of a buying organization, a number of selling organizations, a payment authority, and a requisitioner. The requisitioner performs procurements of goods from one of the selling organizations in behalf of the buying organization. The payment authority carries out the financial transaction of the procurement. Moreover, the standard defines a business-to-business (B2B) model consisting of the following steps:

1. The requisitioner asks the buying organization for hyperlinks to merchant servers of selling organizations.
2. The requisitioner requests the selling organization to offer tenders for the desired goods.
3. Some of the selling organizations create tenders and map them into OBI order requests which are compatible to the EDI standard [DISA01] and transfer the tenders to the buying organization either via the requisitioner or directly.
4. Based on the tenders, the requisitioner and possibly other entities of the buying organization select a winning selling organization and generate an order.
5. The completed order is formatted as an EDI-compatible OBI order object and is transferred to the winning selling organization.
6. The selling organization fulfills the order.
7. On behalf of the selling organization the payment authority issues an invoice to the buying organization and receives a payment.

In order to automate the procurement process, we modified the OBI standard which does not contain mechanisms for a mechanized request for tenders. In particular, we replaced the transfer syntax EDI by the more modern B2B encoding standard cXML (Commercial eXtensible Markup Language) [cXML01] which is based on the popular language XML. cXML defines syntaxes for tender requests, tenders, and purchase orders.

Our example application was developed as a component-structured application based on the SalesPoint-Framework [Schm99]. SalesPoint is non-profit and facilitates the construction of various shop systems by providing software modules. These modules realize various business functions like buying, selling, or leasing goods as well as administrative functions like accounting, storekeeping, and management of product catalogs. SalesPoint is programmed in Java but does not contain software components. Therefore we adapted it and created three components *Restaurant*, *Counting Stock*, and *Catalog* realizing the sales functions of the restaurant, the management of the counting stock, resp. the catalog of offered products.

Fig. 4 delineates the components of the example. Besides of *Restaurant*, *Counting Stock*, and *Catalog*, we created the component *OBI-E-Requisitioner* which realizes the automated requisition[3] of goods. Moreover, we added a *Directory of Sellers* containing the addresses and

[3] In order to use automated procurement, we extended the OBI standard which assumes that the requisition is performed by humans only.

range of goods for sale of suppliers. Finally, the *OBI-Buying Adapter* manages the formatting of tender requests, tenders, and orders according to the OBI specification and acts as an interface to the suppliers. The buying organization (i.e., the fast-food restaurant) is implemented by the combination of these six components. The corresponding selling organizations (i.e., the suppliers) were developed from the SalesPoint-Framework, too. Furthermore, we created a trusted third party logging service in order to support non-repudiation of transactions. The components can also be downloaded from the WWW (URL: ls4-www.cs.uni-dortmund.de/RVS/P-SACS/eReq).

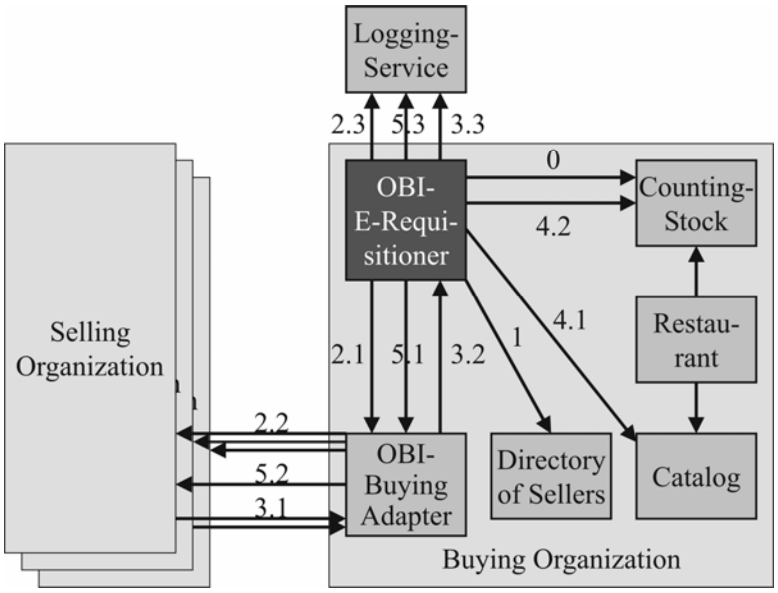

Figure 4: E-Procurement System for Fast-Food Restaurants

The various steps of a purchase are described by the edges of Fig. 4. Besides of carrying out procurements, the E-Requisitioner also decides about starting a purchase. Therefore we added a new step 0 where, in intervals, the E-Requisitioner inspects the counting stock for shortages. If a good is short, the E-Requisitioner, retrieves the addresses of suitable sellers from the Directory of Sellers (step 1). Thereafter it creates request for tenders which are forwarded via the OBI-Buying Adapter to the selling organizations (step 2). Some of the selling organizations will send tenders to the OBI-Buying Adapter which are handed over to the E-Requisitioner (step 3). Afterwards the E-Requisitioner consults the Catalog and the Counting Stock in order to select a winning seller (step 4). In the next step, the E-Requisitioner creates an order and sends it via the OBI-Buying Adapter to the selected selling organization (step 5). The steps 6 and 7 carrying out the delivery and payment were not realized. However, in order to prevent that requests for tenders, tenders, and orders might be later repudiated by the buying organization or the developer of the E-Requisitioner, the corresponding data are digitally signed and sent to the Logging-Service in the steps 2, 3, and 5.

7. Component Behavior Enforcement in the Example System

The procurement process is controlled by the electronic requisitioner. Therefore the correct and secure execution of the component OBI-E-Requisitioner is crucial for the buying organization. If the E-Requisitioner behaves maliciously, it may cause various security violations of the application. For instance, competitor's tenders may be forwarded to preferred suppliers, not the least expensive seller may be selected for an order, the buyer may be hurted by too large, too small, resp. too late orders, or orders may be repudiated later. Assuming that the E-Requisitioner was developed by a non completely trustworthy company, we have to observe it by a security wrapper in order to protect the application. Acting as the designer of this component, we defined 13 cTLA-based component security policy specifications based on the policy patterns introduced in Sec. 5. In particular, we specified four confidentiality policies, four integrity policies, four availability policies, and a non-repudiation policy which are outlined in the following[4]. To protect the confidentiality of the example application, we use the following policies:

- A tender request contains only articles which, according to the directory of sellers, are in the range of articles offered by the particular seller (Data flow access). Thus, information about the existence of the procurement and the portfolio of the buying organization is only forwarded to appropriate sellers.
- A tender is requested only from sellers contained in the directory of sellers (Data flow access). Thus, information about the food and drinks offered by the restaurant are only forwarded to selling organizations which are accepted by the buying organization.
- For an article, the number of copies to be ordered is unambiguously computed from the number of copies which are currently in the counting stock (Hidden channel functional dependency). Thus, the E-Requisitioner cannot send secret additional information by varying the number of ordered goods.
- A tender request and an order may be executed only if the last order was carried out in the meantime (Hidden channel enabling history). This prevents that an order is split up in various suborders where the mode to divide the order may also contain additional hidden information.

The system integrity is protected by enforcing the policies listed below:

- An order may be generated only after a certain minimum number of tenders were received (Integrity enabling history). Thus, one prevents that the E-Requisitioner sends an order to a preferred seller quickly not considering tenders which are received later.
- The requisitioner orders one of the least expensive tenders (Integrity enabling history).
- The values in the counting stock, the catalog, the directory of sellers, the OBI-Buying Adapter, and the logging service are not changed (Integrity enabling condition “false” for modifying operations). This policy prevents attacks against the integrity of the environment components.
- The number of copies ordered is in an interval between a certain minimum and maximum (Integrity enabling condition). This policy avoids too large or too small orders of a good.

The following policies impede attacks against the availability of the component-structured example system:

- Operations of the counting stock, the catalog, the directory of sellers, and the buying adapter are called only after minimum waiting time intervals (Denial of service minimum waiting time). Thus, the E-Requisitioner cannot call its environment continuously which is therefore able to grant service to further parties (e.g., to the Restaurant component).
- The counting stock is polled within maximum waiting time intervals (Blocking maximum waiting time). This guarantees that the order of a good short in the counting stock is timely started.

[4] In the description of the security policies, the identifier set in brackets indicates the name of the cTLA security policy specification pattern used.

- If, according to the counting stock, the number of a certain article is low, a procurement process for this article is started within a maximum time interval (Blocking enabling history). Thus, the order process for this good cannot be continuously delayed.
- After receiving the threshold number of tenders, an order is executed within a maximum waiting time (Blocking enabling history). This also contributes to a timely carrying out of an order.

Finally, we use a non-repudiation security policy assuring that the buying organization can later prove against the selling organizations and against the designer of the E-Requisitioner that requested and incoming tenders as well as orders indeed took place. This is guaranteed by the following policy:

- Tender requests, tender deliveries, and orders are logged at the logging service (Event logging).

The security wrapper checks the events passing the interface of the component OBI-E-Requisitioner for compliance with the 13 security policies. In our reference implementation, the wrapper causes a performance penalty of about 5%. To reduce this penalty, we use the trust information service as well as a trust manager controlling the security wrapper of the E-Requisitioner. Thus, we have to define separate enforcement strategies for each of the 13 security policies. In order to decide, how to manage a policy, we need to discuss the consequences of an attack caused by its relaxation:

The first two confidentiality policies guarantee that the assortment of goods is forwarded only to certain legitimate sellers. This seems to be not too important since, in general, the range of goods sold in a fast-food restaurant is well-known. Nevertheless, a restaurant may be hurt if the introduction of a new article leaks out to the competitors early. Weighing these two contrary effects up, we decided to use a trust management policy of intermediate strength. The two security policies are only permanently enforced if the variable *b* of the current trust value (cf. Sec. 3) is below the value 0.99 according to the more tolerant metric of Jøsang and Knapskog [JøKn98]. Spot checks are allowed if *b* exceeds 0.99 while a value of greater 0.999 according to this metric enables the trust manager to remove the security wrapper at all. Thus, at least 100 resp. 1000 positive experience reports have to be collected and the number of positive reports must exceed the number of negative reports by the factor 100 resp. 1000 in order to start spot checks resp. terminating the monitoring.

In contrast, the demand that tenders may not be forwarded to competing selling organizations is of high relevance in order to prevent financial damage. Since the cooperation between the buying organization and the selling organizations is only performed by means of the cXML-based messages, an intruder may only use hidden channels to forward tender information. The use of hidden channels is impeded by the two other confidentiality security policies which are therefore considered as highly relevant. In consequence, these policies are constantly monitored by the security wrapper.

The four security policies guaranteeing the integrity of the example application are also of high importance. If one of the first two integrity security policies is violated, the buying organization may be severely damaged by excessively expensive procurements. The third policy is of similar relevance since illegal attribute changes of partner components are dangerous since they can cause further attacks like the removal of a less expensive seller from the Directory of Sellers component. Finally, the forth integrity security policy is of similar importance since too large orders of the mostly perishable goods cause significant expense as well. For these reasons, the security wrapper fully monitors all four policies constantly.

The availability security policies seem of less importance. Denial of service attacks are embarrassing since they can impede the sale of goods in the restaurant. Nevertheless, they are noticed by the sales people immediately who have problems to use the tills and the error sources can be relatively easily detected and cleared. Since this danger seems to be similar to the leaking out of the range of goods, we use also Jøsang's and Knapskog's metric for the first availability security policy and allow spot checks if *b* exceeds 0.99 and remove the supervision if *b* is larger than 0.999.

Empty stocks due to late orders, however, are more significant than denial of service attacks since they lead to financial losses of the buying organization. Nevertheless, this kind of attack can be detected relatively easy since we assume that, as a precaution, the restaurant manager supervises the stock from time to time as well. Thus, a shortage of a product can probably be recognized and cleared before any damage is caused. Therefore we do not demand constantly monitoring of the three rules preventing the retarding of orders but use the metric of Beth, Borcherding, and Klein [BeBK94]. Spot checks are permitted if the belief value *b* exceeds 0.999. According to the probability formula of this metric (cf. Sec. 3), about 7000 positive experience reports[5] have to be received to reach this value of *b*. The security wrapper may be removed if *b* exceeds 0.99999 which corresponds to about 11600 positive reports.

Of high importance is also the non-repudiation policy since a violation of this policy may cause severe legal consequences for the buying organization. Therefore it is constantly monitored by the security wrapper, too.

In our E-procurement example, the trust manager could reduce the performance overhead of the security wrapper from 5% to about 3% if the trust value *b* increases above 0.99999 and the enforcement of all but the permanently monitored components is terminated.

8. Concluding Remarks

We introduced the utilization of formal methods to describe security policies in component contracts. Moreover, we outlined the security wrappers monitoring at runtime that the components fulfill the assigned security policies and our trust management approach in order to reduce the overhead of the security wrappers. Finally, the application of the approaches was clarified by means of an e-procurement example.

Except for the runtime checks, our work also concentrates on the design of a cTLA specification framework [Herr03b] facilitating the modeling of component security properties. The framework contains a collection of specification patterns modeling component contract policies as introduced in Sec. 5. Furthermore, we added another library of specification patterns enabling the creation of a cTLA-based role-based access model (cf. [FSG+01]). By means of the patterns of the two collections, security policies of components as well as access control models of the applications embedding the components may be specified in a rather simple manner since the patterns have only to be parameterized and combined to an overall system specification.

Moreover, the framework contains theorems, each stating that a combination of component security policies guarantees a certain aspect of a role-based access control model. Thus, a deduction proof, that the combined security policies of the components forming an application fulfill the access control policy of this application, can be carried out relatively simply since it can be reduced to lemmas which directly correspond to the already proven theorems. The formal verifications are very important for the most security-critical software since the ISO/IEC standard Common Criteria [ISO98] demands that the security properties of highly security-sensitive systems are formally proven. Besides of the component security

[5] The trust manager sets the value α to 0.999 leading to a very slow increasing of *b*.

specification framework [Herr03b], we already developed cTLA specification frameworks for telecommunication protocols [HeKr00a, HeKr98] and for hybrid technical systems [HeKr00b].

Moreover, we plan to replace the cTLA-based specifications of the security policies by graphical diagrams in the well-known description technique UML [BoRJ99]. On the one hand, a UML style to describe security properties of systems already exists (cf. [Jürj03]). On the other hand, we already spent significant work in modeling UML diagrams by means of cTLA processes (cf. e.g., [GrHK99, GrHe04]) which can be utilized to transfer the UML diagrams of security policies into the corresponding cTLA specifications.

References

[AvRa94] F. M. Avolio and M. J. Ranum. *A Network Perimeter with Secure External Access*. In Proc. Internet Society Symposium on Network and Distributed System Security, Glenwood, 1994.

[Bea03] Bea Systems. *BEA Web Logic Security Framework: Working with Your Eco-System*. BEA White Paper, 2003.

[BeBK94] Th. 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, Springer-Verlag, 1994.

[BiEc94] J. Biskup and Ch. Eckert. *About the Enforcement of State Dependent Security Specifications*. In T. Keefe and C. Landwehr (eds.), Database Security, pages 3-17, Elsevier Science (NorthHolland), 1994.

[BJPW99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. *Making Components Contract Aware*. IEEE Computer, 32(7):38-45, 1999.

[BoRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, 1999.

[BSP+95] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. *Extensibility, Safety, and Performance in the SPIN Operating System*. In Proc. 15th Symposium on Operating System Principles, pages 267-284. ACM, 1995.

[cXML01] cXML.org. *cXML User's Guide*. Edition 1.2.006, 2001.

[DeMi03] L. DeMichiel, *Enterprise Java Beans Specification, Version 2.1*. Available on the WWW: http://java.sun.com/products/ejb/docs.html, Sun Microsystems, 2003.

[DISA01] Data Interchange Standards Association. *X12 Standard*. Edition 4050, 2001.

[eBay] eBay Inc. *Feedback Forum*. Available via WWW: http://pages.ebay.com/services/forum/feed\-back.html.

[FrBF99] T. Fraser, L. Badger, and M. Feldman. *Hardening COTS Software with Generic Software Wrappers*. In Proc. 1999 IEEE Symposium on Security and Privacy, 1999.

[FSBJ97] 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.

[FSG+01] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. *Proposed NIST Standard for Role-Based Access Control*. ACM Transactions on Information and System Security, 4(3):224-274, 2001.

[GrHe04] G. Graw and P. Herrmann. *Transformation and Verification of Executable UML Models*. To appear in Electronic Notes on Theoretical Computer Science, Elsevier Science, 2004.

[GrHK99] G. Graw, P. Herrmann, and H. Krumm. *Constraint-Oriented Formal Modelling of OO-Systems*. Proc. 2nd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 99), pages 345-358, Helsinki, Kluwer Academic Publisher, 1999.

[Giga01] Giga Information Group. Available on the WWW: http://www.gigaweb.com, 2001.

[GWTB96] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. *A Secure Environment for Untrusted Helper Applications*. In Proc. 6th USENIX Security Symposium, 1996.

[Hami97] G. Hamilton. *Java Beans*. Available on the WWW: http://java.sun.com/beans/docs/spec.html, Sun Microsystems, 1997.

[HeKr98] P. Herrmann and H. Krumm. *Modular Specification and Verification of XTP*. Telecommunication Systems, 9(2):207-221, 1998.

[HeKr00a] P. Herrmann and H. Krumm. *A Framework for Modeling Transfer Protocols*. Computer Networks, 34(2):317-337, 2000.

[HeKr00b] P. Herrmann and H. Krumm. *A Framework for the Hazard Analysis of Chemical Plants*. In Proc. 11th IEEE International Symposium on Computer-Aided Control System Design (CACSD2000), pages 35-41, Anchorage, IEEE CSS, Omnipress, 2000.

[HeKr01] 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, IEEE Computer Society Press, 2001.

[Herr01a] P. Herrmann. *Trust-Based Procurement Support for Software Components*. In Proc. 4th International Conference on Electronic Commerce Research (ICECR-4), pages 505-514, Dallas, ATSMA, IFIP, 2001.

[Herr01b] P. Herrmann. *Information Flow Analysis of Component-Structured Applications*. In Proc. 17th Annual Computer Security Applications Conference (ACSAC'2001), pages 45-54, New Orleans, ACM SIGSAC, IEEE Computer Society Press, 2001.

[Herr03a] P. Herrmann. *Trust-Based Protection of Software Component Users and Designers*. In P. Nixon and S. Terzis (eds.), Proc. 1st International Conference on Trust Management, LNCS 2692, pages 75-90, Heraklion, Springer-Verlag, 2003.

[Herr03b] P. Herrmann. *Formal Security Policy Verification of Distributed Component-Structured Software*. In H. König, M. Heiner, and A. Wolisz (eds.), Proc. 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'2003), LNCS 2767, pages 257-272, Berlin, Springer-Verlag 2003.

[HeWK02] P. Herrmann, Lars Wiebusch, and Heiko Krumm. *State-Based Security Policy Enforcement in Component-Based E-Commerce Applications*. In Proc. 2nd IFIP Conference on E-Commerce, E-Business & E-Government (I3E), pages 195-209, Lisbon, Kluwer Academic Publisher, 2002.

[IBM] IBM. *Web Sphere Software Platform*. Available on the WWW: http://www-3.ibm. com/software/info1/websphere/index.jsp?tab=products/appserv.

[ISO98] ISO/IEC. *Common Criteria for Information Technology Security Evaluation*. International Standard ISO/IEC 15408, 1998.

[JBo] JBoss. Available on the WWW: http://www.jboss.org.

[JøKn98] A. Jøsang and S. J. Knapskog. *A Metric for Trusted Systems*. In Proc. 21st National Security Conference. NSA, 1998.

[Jøsa96] A. Jøsang. *The Right Type of Trust for Distributed Systems*. In Proc. UCLA conference on New security paradigms workshops, pages 119-131, Lake Arrowhead, ACM, 1996.

[Jøsa99] A. Jøsang. *An Algebra for Assessing Trust in Certification Chains*. In J. Kochmar (ed.), Proc. Network and Distributed Systems Security Symposium (NDSS'99). The Internet Society, 1999.

[Jürj03] J. Jürjens. *Developing Safety-Critical Systems with UML*. In P. Stevens, J. Whittle, and G. Booch (eds.), Proc. 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML2003), LNCS 2863, pages 360-372, San Francisco, Springer-Verlag, 2003.

[KhHZ01] K. Khan, J. Han, and Y. Zheng. *A Framework for an Active Interface to Characterise Compositional Security Contracts of Software Components*. In Proc. Australian Software Engineering Conference (ASWEC'01), pages 117-126, Canberra. IEEE Computer Society Press, 2001.

[Koze98] D. Kozen. *Efficient Code Certification*. Technical Report 98-1661, Computer Science Department, Cornell University, 1998.

[Koze99] D. Kozen. *Language-Based Security*. In M. Kutylowski, L. Pacholski, and T. Wierzbicki (eds.), Proc. Conference on Mathematical Foundations of Computer Science (MFCS'99), LNCS 1672, pages 284-298, Springer-Verlag, 1999.

[Mall00] A. Mallek. *Sicherheit komponentenstrukturierter verteilter Systeme: Vertrauensabhängige Komponentenüberwachung*. Diploma Thesis, University of Dortmund, Informatik IV, D-44221 Dortmund, 2000. In German.

[Micr98] Microsoft. *The Microsoft COM Technologies*. Available on the WWW: http://www.microsoft.com/com/comPapers.asp, 1998.

[MiFS90] B. P. Miller, L. Fredrikson, and B. So. *An Empirical Study of the Reliability of Unix Utilities*. Communications of the ACM, 32(12):32-44, 1990.

[Monr93] M. A. Monroe. *Security Tool Review: TCP Wrappers*. ;login:, 18(6):15-16.

[MWCG98] G. Morrisett, D. Walker, K. Crary, and N. Glew. *From System F to Typed Assembly Language*. In Proc. 25th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages, pages 85-97, San Diego, 1998.

[MyLi98] 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.

[Myer99] 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.

[Necu98] G. C. Necula. *Compiling with Proofs*. PhD Thesis, Carnegie Mellon University, 1998.

[NiPe97] P. Niemeyer and J. Peck. *Exploring JAVA*. O'Reilly, 2nd Edition, 1997.

[OBI99] OBI Consortium. *OBI Technical Specifications – Open Buying on the Internet*. Draft Release, Edition V2.1, 1999.

[OMG02] OMG. *CORBA Components, Version 3.0*. Edition formal/02-06-65, June 2002.

[RZFK00] P. Resnick, R. Zeckhauser, E. Friedman, and K. Kuwabara. *Reputation Systems: Facilitating Trust in Internet Interactions*. Communications of the ACM, 43(12):45-48, 2000.

[Schm99] L. Schmitz. *The SalesPoint Framework – Technical Overview*. Available on the WWW: http://ist.unibw-muenchen.de/Lectures/SalesPoint/overview/english/TechDoc.htm, 1999.

[Schn97] F. B. Schneider. *Towards Fault-Tolerant and Secure Agentry*. In Proc. 11th International Workshop on Distributed Algorithms (WDAG'97), LNCS 1320, pages 1-14. ACM SIGPLAN, Springer-Verlag, 1997.

[Szyp97] C. Szyperski. *Component Software – Beyond Object Oriented Programming*. Addison-Wesley Longman, 1997.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. *TIL: A Type-Directed Optimizing Compiler for ML*. In Proceedings of the Conference on Programming Language Design and Implementation, ACM SIGPLAN, 1996.

[VMGM96] J. Voas, G. McGraw, A. Ghosh, and K. Miller. *Glueing together Software Components: How good is your Glue?* In Proc. Pacific Northwest Software Quality Conference, Portland, 1996.

[Voas98] J. Voas. *A Recipe for Certifying High Assurance Software*. In Proc. 22nd International Computer Software and Application Conference (COMPSAC'98), Wien, IEEE Computer Society Press, 1998.

[VSvS88] C.-A. Vissers, G. Scollo, and M. van Sinderen. *Specification Styles in Distributed System Design and Verification*. Theoretical Computer Science, 89:179-206, 1991.

[WLAG93] R. Wabbe, S. Lucco, T. E. Anderson, and S. L. Graham. *Efficient Software-Based Fault Isolation*. In Proc. 14th Symposium on Operating System Principles, pages 203-216, ACM, 1993.

[ZFK+98] J. Zöllner, H. Federrath, H. Klimant, A. Pfitzmann, R. Piotraschke, A. Westfeld, G. Wicke, and G. Wolf. *Modeling the Security of Steganographic Systems*. In Proc. 2nd Workshop of Information Hiding, LNCS 1525, pages 345-355, Portland, Springer-Verlag, 1998.