Formal Security Policy Verification of Distributed Component-Structured Software

Peter Herrmann

University of Dortmund, Computer Science Department, 44221 Dortmund, Germany,
Peter.Herrmann@udo.edu

Abstract. Component-structured software, which is coupled from independently developed software components, introduces new security problems. In particular, a component may attack components of its environment and, in consequence, spoil the application incorporating it. Therefore, to guard a system, we constrain the behavior of a component by ruling out the transmission of events between components which may cause harm. Security policies describing the behavior constraints are formally specified and, at runtime, so-called security wrappers monitor the interface traffic of components and check it for compliance with the specifications. Moreover, one can also use the specifications to prove formally that the combinations of the component security policies fulfill certain security properties of the complete component-structured application. A well-known method to express system security properties is access control which can be modelled by means of the popular Role Based Access Control (RBAC) method.
Below, we will introduce a specification framework facilitating the formal proof that component security policy specifications fulfill RBAC-based application access control policies. The specification framework is based on the specification technique cTLA. The design of state-based security policy specifications and of RBAC-models is supported by framework libraries of specification patterns which may be instantiated and composed to a specification. Moreover, the framework contains already proven theorems facilitating the formal reasoning since a deduction proof can be reduced to proof steps which correspond directly to the theorems. In particular, we introduce the specification framework and clarify its application by means of an e-commerce example.

1 Introduction

More and more e-commerce applications are composed from cost-effective software components which are developed independently from each other and are separately offered on an open market (cf. [1]). The component-structured applications can be tailored to the particular needs of their customers by selecting, configuring, and customizing suitable components which may come from various sources. The architecture of the component-structured software and, in particular, the heterogeneity of the component interfaces, however, aggravates the composition process. The coupling of components therefore is facilitated by using

H. König, M. Heiner, and A. Wolisz (Eds.): FORTE 2003, LNCS 2767, pp. 257–272, 2003.

— ideally legally binding — component contracts describing all context dependencies of a component. According to [2], a component contract consists of four parts. In the first part the methods, events, and exceptions of the component interfaces and the corresponding arguments are listed. The second part describes constraints of the interface behavior to be fulfilled by the component resp. its environment. The third part models synchronization aspects while the forth part lists quality-of-service properties. The contracts are utilized by visual application builder tools which facilitate the composition process. Well-established platforms for component-structured software are Enterprise Java Beans (EJBs) which are based on Sun's Java Beans and Microsoft's COM/DCOM. Moreover, OMG's middleware platform CORBA was extended by a component model.

Component-structured software also faces new security risks since, compared with monolithic applications, a higher number of principals is involved in the system development and deployment (i.e., application owners and users, component designers and vendors, system builders, component service providers). On the one hand, each principal introduces his own security objectives which have to be fulfilled by the other principals. On the other hand, a principal is a potential threat to the other principals as well as to the component-structured application. Taxonomies of security risks for component-structured software are introduced in [3, 4]. Our work concentrates on a main security risk which cannot be dealt with traditional security mechanisms: A malicious component may distort other components and, in consequence, spoil the application incorporating it. This risk is addressed by so-called security wrappers which are introduced in [5, 6]. We expanded the component contracts by formal state-based models specifying that components always behave in a way that the security of their environment components is guaranteed. Moreover, a security wrapper is inserted at the interfaces of a component. It is a special software wrapper (cf. [7]) which temporarily blocks all events passing an interface and checks them for compliance with the component contract security policy specifications. To perform the compliance checks, the wrapper simulates the specifications and accepts an event only if the transitions modelling the event are enabled in the current states of all simulations. An accepted event is forwarded to its destination. If, however, an event is rejected, the wrapper seals the component by blocking its interface traffic and notifies the application administrator. The performance effort of the security wrappers is between 5 and 10 %, but can be significantly reduced by applying trust management (cf. [4, 8]).

Similarly to us, Khan et al. [9] extend component contracts to describe security aspects in order to check requirement-assurance relationships between coupled components. The used description technique is relatively simple and does not represent behavioral properties used to model interface behavior.

In this paper, we concentrate on another utilization of the component contract security models. Besides of runtime enforcement by the security wrappers, the specifications can also be used at design time to verify that the application incorporating the constrained components fulfill more general system-oriented security policies. These policies can be regarded as safeguards for a system which

make successful attacks more difficult. Well-known techniques to describe system security policies are access control and information flow models. The use of formal methods to prove that the component contract models realize system security policies is of great practical interest since the current standard for security analysis, the so-called Common Criteria (CC, [10]), demands a formal-based analysis for the certification of systems operating in a highly sensitive environment. As a description technique for the system security properties we use the popular Role-Based Access Control method (RBAC, [11]) which is well-suited to rule out attacks on the confidentiality and integrity of systems.

To develop specifications of component contract security policies or system security policies and to perform refinement proofs, we consider the specification technique cTLA [12] suitable. cTLA facilitates the creation of state transition systems in a modular and process-oriented fashion. In particular, it enables the composition of system descriptions from implementation-oriented processes (e.g., contract models describing aspects of component interface behavior), constraint-oriented processes (e.g., certain constraints of an RBAC model), and combinations (cf. [13]). Process composition in cTLA has the character of superposition (cf. [14]). Here, a relevant property of a process or a subsystem is also a property of the embedding system. Therefore structured verification is possible (i.e., a proof that a system fulfills a property can be reduced to the verification that an — often very small — subsystem fulfills this property). Moreover, structured verification facilitates the establishment of specification and verification frameworks which contain libraries of cTLA process types. System and subsystem specifications can easily be developed by instantiating and composing the process types from the framework libraries. In particular, a specification framework contains libraries of theorems which are proven by the framework designer. A theorem states that, if some side conditions hold, a subsystem consisting of more detailed processes fulfills a more coarse-grained process. Thus, in order to prove that a detailed system C fulfills a coarse-grained system S, one has to check only that for each process of S a theorem exists stating that a subsystem of C fulfills this process. Furthermore, one has to prove that the side conditions of the applied theorems hold and that the processes of C and S are consistently coupled. These checks, however, are usually simple and can be automated (cf. [15]). Therefore the specification frameworks make formal proofs of real-sized problems possible (e.g., service verification of transport communication protocols [16]). Below, we will introduce a specification framework for the verification that component contract security policy specifications fulfill RBAC-based system security policy models. The cTLA processes and theorems of this framework are depicted in the WWW (URL: ls4-www.cs.uni-dortmund.de/RVS/P-SACS/eReq).

2 cTLA

Leslie Lamport's Temporal Logic of Actions (TLA, [17]) is a linear time temporal logic which describes safety and liveness properties (cf. [18]) of state transition systems by means of canonical formulas. cTLA (compositional TLA, [12]) is

```
PROCESS IntegrityEnablingHistorySTS
          (ComponentIds : ANY; InterfaceIds : ANY; EventIds : ANY;
           Args : ANY; ConstrEv : EvDfType; States : ANY;
           InitState : States; ExecuteCond : SET[States → SUBSET(Args)];
           Trans : SET[(States × EvDfType × Args) → States])
CONSTANTS
  EvDfType ≜ (ComponentIds × ComponentIds × InterfaceIds × EventIds);
BODY
  VARIABLES
    var : States;
  INIT ≜ var = InitialState;
  ACTIONS
    Event (callee : ComponentIds; caller : ComponentIds;
           if : InterfaceIds; ev : EventIds; arg : Args) ≜
      ((callee, caller, if, ev) ≠ ConstrEv ∨ arg ∈ ExecuteCond[var]) ∧
      var′ = Trans[var, (callee, caller, if, ev), arg];
END
```

Fig. 1. Process type `IntegrityEnablingHistorySTS`

based on TLA, but enables the suitable specification of systems in the notation of processes. A process acts as a modular specification component and a system can be specified by a set of coupled processes. A process has either the form of a simple process or of a process composition. Simple processes refer directly to state transition systems and can represent implementation parts as well as logical constraints.

Fig. 1 depicts the simple cTLA process type `IntegrityEnablingHistorySTS` used to specify a pattern of a typical component contract specification. It is introduced to more detail in Sec. 5. In the header the process type name and generic module parameters (e.g., `ComponentIds`) are declared. The parameters facilitate the specification of a spectrum of similar but different processes by a single process type. In the section`CONSTANTS` constant and type definitions are listed. The body of the process type defines the state transition system modelling a process instance. The state space is specified by the state variable `var` which carries values of the data type defined by the process parameter `States`. The initial condition `INIT` refers to state variables and defines the set of initial states (i.e., `var` initially has the value expressed by the parameter `InitState`). The state transitions are specified by actions (i.e., `Event`). An action is a predicate on a pair of a current and a successor state modelling a set of state transitions. The state variables referring to the current state are described by ordinary variable identifiers (i.e., `var`) while variables referring to the successor state occur in the so-called primed form (i.e., `var'`). An action may be supplemented by action parameters (e.g., `callee`). The next state relation of the modelled state transition system corresponds to the disjunction of the actions.

Systems and subsystems are described as compositions of concurrent processes. As in the ISO/OSI specification language LOTOS, a set of processes interact in a rendezvous-like way by performing actions jointly, and the data

parameters of the actions can model the communication of values between processes. Each process encapsulates its variables and changes its state by atomic execution of its actions. The system state is the vector of the process state variables. State transitions of the system correspond to simultaneously executed process actions or to so-called process stuttering steps (i.e., the process does not change its state). Since, moreover, a process participates in a system action either by exactly one process action or by a process stuttering step, one can define a system action by a conjunction of process actions and stuttering steps. In consequence, concurrency is modelled by interleaving while the coupling of processes corresponds to joint actions. The design of cTLA process types as well as the composition of processes to systems is supported by a compiler tool [19].

3 Global Formal Model

The component contract security policy descriptions and the RBAC-based system security models are specified by means of cTLA process types. In order to provide a common understanding of the interaction between process instances facilitating the composition of the processes to system specifications, we defined a simple global formal model. In a specification of a component-structured application, this model represents the components, the component interfaces, the events transferred between components, and the event arguments. The process instances describing the global model form the core of all component contract or system security specifications. In particular, the global model defines naming conventions for process parameters, action names, and action parameters guiding the instantiation and composition of the other process types to system specifications. Thus, by enforcing identical names for similar behavior aspects in the component contract specifications and in the descriptions of the system security properties, refinement proofs are made easier. To keep the global model simple, only static component-structured systems (i.e., systems without adding or removing components during runtime) are currently supported[1]. Moreover, we use events as the only means to describe cooperation between components since other interaction constructs like method calls or exceptions can also be realized by events (e.g., a method call is the combination of a calling and a return event).

The main cTLA process type is `GlobalSystem` (cf. Fig. 2). By means of the generic process parameters `ComponentIds`, `InterfaceIds`, and `EventIds` one can define identifiers for the components, interfaces, and events of a component-structured application while the arguments of the events are defined by the process parameter `Args`. Identical parameters occur in other cTLA process types where they should be instantiated with the same sets as in `GlobalSystem`. The parameter `Interfaces` of the type `InterfaceType`, which is defined in the section`CONSTANTS`, specifies the component interfaces. An interface consists of an identifier `iId`, a set `iEvents` of events passing the interface, and a function

[1] Dynamic changes of the component structure can be modelled by means of special life cycle variables (cf. [20]).

```
PROCESS GlobalSystem (ComponentIds : ANY; InterfaceIds : ANY;
                      EventIds : ANY; Args : ANY;
                      Interfaces : SUBSET(InterfaceType);
                      CompIfs : SET[ComponentIds → SUBSET(Interfaces)])
CONSTANTS
  InterfaceType ≙ [[ iId : InterfaceIds; iEvents : SUBSET(EventIds);
                     iEvArgs : SET[EventIds → Args] ]] ;
BODY
  INIT ≙ True;
  ACTIONS
    Event (callee : ComponentIds; caller : ComponentIds;
           if : InterfaceIds; ev : EventIds; arg : Args) ≙
      ∃ i ∈ InterfaceType :: i ∈ CompIfs[callee] ∧ i.iId = if ∧
                             ev ∈ iEvents ∧ arg = i.iEvArgs[ev];
END
```

Fig. 2. Process type `GlobalSystem`

`iEvArgs` assigning a set of arguments to each event. Finally, the parameter `CompIfs` maps component identifiers to interfaces and specifies which interfaces belong to a component. Since the process type is stateless, the process body does not contain variables and the initial condition is always true. `Event` is the only action used in this or in other process types. It models that an event with the identifier `ev` and the argument `arg` is sent from the component `caller` to the interface `if` of the component `callee`.

Besides of `GlobalSystem`, the global formal model consists of the process type `ComponentCoupling` describing which components are enabled to send events to an interface of a particular component. This process is listed in the WWW (URL: ls4-www.cs.uni-dortmund.de/RVS/P-SACS/eReq). The complete global model is specified by means of an instance of process type `GlobalSystem` and a separate instance of `ComponentCoupling` for each component being coupled to the modelled component-structured application.

4 RBAC System Models

A malicious component may attack its environment in many different ways violating the confidentiality, integrity, availability, or non-repudiation of the embedding application. In the following, we sketch some threats relevant for the confidentiality and integrity of a component-structured system while availability and non-repudiation threats are listed in [6]. Confidentiality attacks are particularly significant for distributed component-based systems where the components reside in different security domains with varying user-access policies. Here, the information flow between components may be maliciously altered in a way that data is forwarded to system parts controlled by principals who are not allowed to read the data. Moreover, hidden channels may be used where certain information is illegally concealed in transferred data — so-called steganography — or

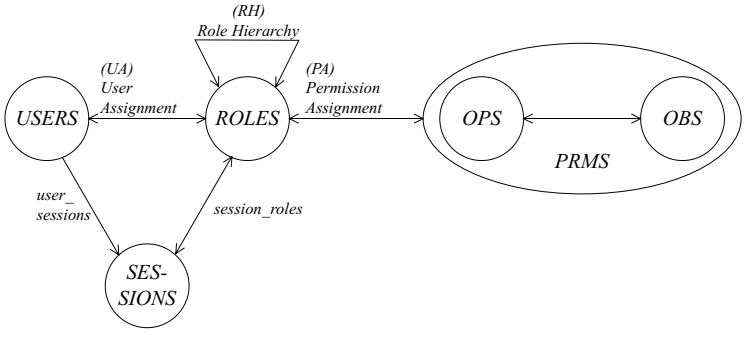

Fig. 3. RBAC model (taken from [11])

by defining secret agreements in the order, the number, or the execution time of events. A component may attack the integrity of its environment by triggering incorrect events, by modifying the event parameters, or by manipulating the system configuration parameters. Due to these attacks the application may deviate from the specified system behavior harming the system owners and users.

A well-known technique to protect systems against confidentiality or integrity attacks is access control in which the read or write access on various system resources is limited to certain users. A popular access control technology is Role-Based Access Control (RBAC, [11, 21]) which allows to express a wide range of security policies (cf. [22]). Fig. 3 outlines the major structure of RBAC. It uses roles (*ROLES*) as main construct to assign access permissions to users. The roles model certain positions in an application domain (e.g., in a hotel environment *guest*, *general manager*, *housekeeping manager*, and *housekeeping staff member* are examples of roles). The roles may be hierarchically organized. In this case the rights of a lower level role are inherited by a higher level role (e.g., the *general manager* has also the rights of a *housekeeping manager*). Access permissions (*PRMS*) describe the right to perform certain operations (*OPS*) on objects (*OBS*). The RBAC model defines a many-to-many relation between roles and permissions (e.g., a *general manager* may give reductions to the room prizes). Another many-to-many relation is defined between roles and users (*USERS*, e.g., *Alice* is a *general manager* while *Bob* and *Charlotte* are both a *housekeeping manager*). A user has the permission to access a resource if at least one role is assigned to both himself and the desired permission (e.g., *Alice* may decide about room prize reductions but not *Bob*). Moreover, to use a permission, the corresponding role must be active for the user requesting access. Therefore, the RBAC model includes so-called sessions (*SESSIONS*). Each session models the set of active roles by mapping a user to a subset of the roles assigned to him. Furthermore, one can define certain constraints limiting the assignment of roles or the number of active roles (cf. [11]). A constraint type is static separation of duty. Here, a user may not be assigned to two different roles in order to avoid conflicts of interest (e.g., the *general manager* may never be a *guest* in her

```
PROCESS ActiveRoles
         (ComponentIds : ANY; InterfaceIds : ANY; EventIds : ANY;
          Args : ANY; Roles : ANY; ConstrCallers : SUBSET(ComponentIds);
          ConstrRoles : SUBSET(Roles); InitRoles : SUBSET(Roles);
          ActivateRoles : SET[(EvDfType × Args) → SUBSET(Roles)];
          DeactivateRoles : SET[(EvDfType × Args) → SUBSET(Roles)])
CONSTANTS
  EvDfType ≜ (ComponentIds × ComponentIds × InterfaceIds × EventIds);
BODY
  VARIABLES
    act : SUBSET(Roles);
  INIT ≜ act = InitRoles ∩ ConstrRoles;
  ACTIONS
    Event (callee : ComponentIds; caller : ComponentIds;
           if : InterfaceIds; ev : EventIds; arg : Args;
           actRoles : SUBSET(Roles)) ≜
      (caller ∉ ConstrCallers ∨ act = (actRoles ∩ ConstrRoles)) ∧
      act′ = ((act ∪ ActivateRoles[((callee,caller,if,ev),arg)]) \
              DeactivateRoles[((callee,caller,if,ev),arg)]) ∩ ConstrRoles;
END
```

Fig. 4. Process type `ActiveRoles`

hotel in order to prevent making room prize reductions to herself). In contrast, in dynamic separation of duty a user may be assigned to two conflicting roles which, however, must not be active at the same time (e.g., *Bob* and *Charlotte* who are also *housekeeping staff members* are not allowed to act as manager and staff member simultaneously). Finally, the number of users for which a role is currently active may be limited (e.g., only one person at a time may act as a *housekeeping manager*).

In the domain of component-structured software, the components linked to an application form the group of users. A permission corresponds to the right to send a particular event to a certain interface of another component. The definition of the roles is guided by the tasks performed by the software-based application. Fig. 4 depicts the cTLA process type `ActiveRoles` specifying which roles are currently active for a particular component. Besides of the basic process parameters `ComponentIds`, `InterfaceIds`, `EventIds`, and `Args`, the process type contains the parameter `Roles` describing the roles in the modelled RBAC model. In order to allow modular system specifications consisting of a large number of simple process instances, the components and roles constrained by a process can be limited. The parameters `ConstrCallers` and `ConstrRoles` describe the identifiers of the constrained callers resp. roles. The parameter `InitRoles` specifies which roles are initially active. Finally, `ActivateRoles` and `DeactivateRoles` define which roles get active resp. inactive. These two parameters are mappings from event references which are modelled by the data type `EvDfType` to sets of roles. The state variable `act` specifies which constrained roles are currently active. The action `Event` contains an additional action parameter `actRoles` spec-

ifying the set of roles which are active for the event caller. Thus, this set can also be accessed by other process instances in the RBAC specification. `Event` is always enabled if the caller is not constrained by the process instance. Otherwise, each role constrained by the process has to be included in both sets defined by the variable `act` and the process parameter `actRoles` or in neither of them.

Besides of `ActiveRoles`, we defined cTLA process types modelling the relations between roles and users resp. permissions as well as RBAC constraints.

5 Component Contract Security Specifications

To facilitate the design of specifications constraining the interface behavior of a component in order to guarantee security properties, we defined a group of cTLA process types, too. Each process type models a pattern for a component contract security policy. Component developers define contract security policies based on the patterns by instantiating the cTLA process types and by adding the process instances to the component contract. Reflecting that one can only consider the component interface behavior but not the internal attribute settings and internal events of a component, the component contract security policy patterns address the attack scenarios outlined in the beginning of Sec. 4 directly. Confidentiality attacks can be avoided by restricting the flow of data between components. A data unit must only be send to components which deny read-access to principals unauthorized to read it. Moreover, one can impede hidden channels by preventing non-deterministic interface behavior (cf. [23]). Therefore, to prevent steganography, corresponding security policies define deterministic functional dependencies between data units and preceding events and restrict the order, number, and execution time of the events. The confidentiality policies are realized by the following policy patterns:

- **Data flow access:** A data unit may only be forwarded to a component if its read access permissions are preserved.
- **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 a occurrence dependency function.
- **Hidden channel execution time:** An interface event has to be executed after a preceding interface event within a certain time period.

Integrity attacks are addressed by security policies restricting the execution of interface events and the selection of event arguments. Moreover, the policies can be used to check if events are plausible with respect to the context of preceding events. The policies are specified by the patterns listed below:

- **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.

Other policy patterns to prevent availability and non-repudiation attacks are introduced in [6]. An example of a cTLA process type specifying a specific component contract security pattern is `IntegrityHistorySTS` which is depicted in Fig. 1. It realizes the specific pattern **integrity enabling history** by modelling that a certain event (defined by the generic process parameter `ConstrEv`) with a corresponding argument setting must only be triggered if a particular history of events occurred before. To describe the history, in the process type a state transition system is explicitly modelled. Here, the generic process parameter `States` describes the state space, `InitState` the initial state, and `Trans` the next step relation which depends on executed events. Finally, the process parameter `ExecuteCond` defines for which event arguments the event `ConstrEv` may be executed in a certain state. The state variable `var` models the current state of the state transition system.

6 Verification

In order to verify that component contract policy patterns fulfill an RBAC-based system security model, the owner of the component-structured application develops two cTLA specifications S and C. S models the RBAC model which is created by instantiating the RBAC-oriented cTLA process types introduced in Sec. 4 and composing them with the process instances defining the global formal model (cf. Sec. 3). Likewise, a specification C describing component contract policies is designed by coupling the processes of the global model with instances of the security policy patterns (cf. Sec. 5).

The verification is performed by means of a TLA deduction proof of the implication $C \Rightarrow S$ (cf. [17]). We can reduce this proof into a series of simpler proof steps. In each proof step, we prove that a cTLA process instance of S is fulfilled by a subsystem of C which usually consists of a relatively small number of process instances. If all instances of S are realized by subsystems of C and the processes of C and S are consistently coupled with each other, $C \Rightarrow S$ holds due to the compositionality of cTLA[2]. The consistency of the process couplings in C and S is trivially true since in both specifications the process actions `Event` are coupled with each other to the system actions `Event`. The proof steps describing that subsystems of C realize process instances of S correspond directly to the already proven framework theorems. Thus, we can reduce the verification of $C \Rightarrow S$ to the selection of suitable framework theorems and to some consistency checks of process parameter instantiations. A tool supporting the theorem selection and performing most of the checks is introduced in [15].

An example theorem is depicted in Fig. 5. It states that a subsystem consisting only of an instance of the cTLA process type `IntegrityEnablingHistorySTS`

[2] The corresponding proof is included in [12].

```
LET
  Pars ≜ ∃ rm ∈ SET[States → SUBSET(ConstrRoles)] ::
           (rm[InitState] = InitRoles ∧
            ∀ s ∈ States ∀ edt ∈ EvDfType ∀ a ∈ Args ::
              rm[Trans[(s,edt,a)]] = (rm[s] ∪ ActivateRoles[(edt,a)]) \
                                     DeactivateRoles[(edt,a)]);
  Sys ≜ IntegrityEnablingHistorySTS
          (ComponentIds,InterfaceIds,EventIds,Args,ConstrEv,States,
           InitState,ExecuteCond,Trans);
IN Pars ∧ Sys ⇒ ActiveRoles (ComponentIds,InterfaceIds,EventIds,Args,
                             Roles,ConstrCallers,ConstrRoles,InitRoles,
                             ActivateRoles, DeactivateRoles);
```

Fig. 5. Theorem to prove `ActiveRoles` based on `IntegrityEnablingHistorySTS`

(cf. Fig. 1) fulfills an instance of the process type `ActiveRoles` (cf. Fig. 4) if the theorem condition `Pars` holds. Thus, the theorem proves that an instance of the specific security policy pattern **integrity enabling history** realizes the activation mechanism for certain roles in an RBAC model. To apply this theorem, we have to prove the condition *Pars* stating that we must find a mapping `rm` between the state variable `var` of `IntegrityEnablingHistorySTS` and the variable `act` of `ActiveRoles`. The mapping has to guarantee that the initial state of `var` maps to the initial set of active roles in `act` and that a state change of `var` leads to a corresponding change of `act`. Thus, `rm` is a so-called refinement mapping (cf. [17]) which was used to prove the theorem. In our experience, it is relatively easy to detect suitable refinement mappings due to the use of small subsystems in cTLA (cf. [16]).

7 Proof Example

In [6] we presented a component-structured application example performing the commodity management of fast-food franchise restaurants. The core of this system was developed on the basis of the SalesPoint-Framework [24], a Java-based framework of shop administration functions. Based on these functions we created three components which realize the restaurant sales functions, the counting stock management, and the product catalog. Moreover, we added four self-programmed components in order to adapt the commodity management of our shop to the Open Buying on the Internet (OBI) standard [25]. In this standard an architecture for electronic procurement of goods and a corresponding business-to-business model are defined. The architecture introduces a buying organization, selling organizations, a payment authority, and a requisitioner. In behalf of the buying organization the requisitioner carries out orders at the selling organizations and the orders are paid by means of the payment authority. In particular, in intervals, the requisitioner checks the counting stock for shortages of goods. If a shortage was detected, the requisitioner requests seller addresses from the buying organization, sends requests for tenders to the sellers, receives

tenders, decides about a winning seller based on the tenders, and sends an order to the winning seller. Thereafter the order is fulfilled and paid by means of the paying authority. We changed the OBI standard in one respect: The procurements are not performed by a human but are automatically carried out by a so-called e-requisitioner component. Furthermore, we added a directory-of-sellers component storing the seller addresses and the range of goods offered by a seller. An adapter component manages the communication with the sellers which is realized by standardized tender requests, tenders, and orders. Moreover, we use remote seller components which are also based on the SalesPoint-Framework and a notary logging service in order to support non-repudiation of the relevant transactions.

With respect to system security, we have to guarantee that the buying organization is not cheated to the advantage of certain sellers. This is of relevance, since we presume that the e-requisitioner is obtained from a non-trusted source and its code may be manipulated in order to favor certain sellers. We assume, however, that the contract of the e-requisitioner component contains security contract models which can be enforced by a security wrapper. The models are listed in [6] and the corresponding cTLA process instances can be obtained from the WWW (URL: ls4-www.informatik.uni-dortmund.de/RVS/P-SACS/eReq). As a system user, we have to specify the desired system security policies and prove that they are fulfilled by the component contract policy descriptions. An RBAC-based model is used to specify various system security objectives (e.g., a seller is selected which delivered one of the least expensive tenders).

In the following, we concentrate on an integrity security policy defining that a procurement may only be started if the e-requisitioner detected a shortage of a good in the counting stock. In the RBAC model this is modelled by roles which have to be active for the e-requisitioner in order to perform procurements. Since we assume, that procurements of food and beverages are carried out separately, we use the two roles `"foodReq"` and `‘‘beverageReq"`. The roles are activated if the e-requisitioner detects a shortage of food resp. beverages. They are deactivated if corresponding orders are carried out. The activation and deactivation mechanism for the two roles is modelled by the cTLA process `ActiveBuyingRoles` (cf. Fig. 6) which is instantiated from `ActiveRoles`. The process is modelled as a cTLA process composition and, in order to facilitate the understanding of the process parameter instantiations, we instantiate the generic parameters of `ActiveRoles` with constants carrying the same names as the parameters. The setting of the parameter `ConstrRoles` describes that the process constrains only the two roles `‘‘foodReq"` and `‘‘beverageReq"`. The instantiations of the parameters `InitRoles`, `ActivateRoles`, and `DeactivateRoles` model that the two roles are initially inactive, that they are activated by `‘‘RecvGetStock"` events if the counting stock for a certain good is below a threshold, and that they are deactivated by `‘‘CallSendOrder"` events.

To prove that the specification C consisting of component policy contract models fulfills the process `ActiveBuyingRoles`, we apply the component contract specification `AskDoSIfBuyerOnly` of the type `IntegrityEnablingHistory`

```
PROCESS ActiveBuyingRoles
CONSTANTS
  ...
  ConstrRoles ≜ {"foodReq","beverageReq"};
  InitRoles ≜ {};
  ActivateRoles ≜
    [ (edt,a) ↦
      IF (edt = ("eReq","stock","main","RecvGetStock") ∧
          a.no < threshold[a.good])
      THEN IF (a.type = "food") THEN {"foodReq"} ELSE {"beverageReq"}
      ELSE {} ];
  DeactivateRoles ≜
    [ (edt,a) ↦
      IF (edt = ("adap","eReq","main","CallSendOrder"))
      THEN IF (a.type = "food") THEN {"foodReq"}
           ELSE IF (a.type = "beverage")
                THEN {"beverageReq"} ELSE {"foodReq","beverageReq"}
      ELSE {} ];
PROCESSES
  p : ActiveRoles
        (ComponentIds,InterfaceIds,EventIds,Args,Roles,ConstrCallers,
         ConstrRoles,InitRoles,ActivateRoles,DeactivateRoles);
ACTIONS
  Event (callee : ComponentIds; caller : ComponentIds;
         if : InterfaceIds; ev : EventIds;
         arg : Args; activeRoles : SUBSET(Roles)) ≜
    p.Event(callee,caller,if,ev,arg,activeRoles);
END
```

Fig. 6. Process instance `ActiveBuyingRoles`

(cf. Fig. 7). This process defines a state transition system modelling that the e-requisitioner may only call the directory-of-sellers for seller addresses if some goods in the counting stock are short. The state transition system has the four states ``no", ``food", ``beverage", ``foodAndBeverage" modelling which types of goods are short. The proof is performed by means of the theorem listed in Fig. 5. Thus, we can reduce the proof to the recognition of a suitable refinement mapping which fulfills the two side conditions specified by the conjuncts in the theorem condition `Pars`. A straightforward refinement mapping is

```
rm ≜ [ s ↦ IF (s = "food") THEN {"foodReq"}
           ELSE IF (s = "beverage") THEN {"beverageReq"}
                ELSE IF (s = "foodAndBeverage")
                     THEN {"foodReq", "beverageReq"}
                     ELSE {} ]
```

The first conjunct of `Pars` is true for this refinement mapping since the equation chain `rm[InitState] = rm["no"] = {} = InitRoles` holds. To verify the second conjunct, we reduce the proof to cases guided by the conditions of the `IF THEN ELSE`-constructs in the parameter `Trans` of `AskDoSIfBuyerOnly`. For

```
PROCESS AskDOSIfBuyerOnly
CONSTANTS
  ...
  States ≜ {"no","food","beverage","foodAndBeverage"};
  InitState ≜ "no";
  Trans ≜
    [(s,edt,a) ↦
      IF (edt = ("eReq","stock","main","RecvGetStock") ∧
          a.no < threshold[a.good])
      THEN IF (a.type = "food")
           THEN IF (s ∈ {"no","food"}) THEN "food" ELSE "foodAndBeverage"
           ELSE IF (s ∈ {"no","beverage"})
                THEN "beverage" ELSE "foodAndBeverage"
      ELSE IF (edt = ("adap","eReq","main","CallSendOrder"))
           THEN IF (a.type = "food")
                THEN IF (s ∈ {"no","food"})
                     THEN "no" ELSE "beverage"
                ELSE IF (a.type = "beverage")
                     THEN IF (s ∈ {"no","beverage"})
                          THEN "no" ELSE "food"
                     ELSE "no"
           ELSE s];
...
END
```

Fig. 7. Process `AskDoSIfBuyerOnly`

instance, in the case that the state transition system is in the state ‘‘`no`" or ‘‘`food`" and the e-requisitioner receives an event from the counting stock indicating a shortage of food, we can prove the conjunct by means of the two equation chains `rm[Trans[(s,edt,a)]] = rm["food"] =` {‘‘`foodReq`"} and `(rm[s] ∪ ActivateRoles[(edt,a)]) \ DeactivateRoles[(edt,a)] =` {‘‘`foodReq`"}\{} = { ‘‘`foodReq`"}. By applying similar framework theorems, we prove that C fulfills the other processes of the RBAC-based system security specification S, too.

8 Concluding Remarks

We reported on an approach making formal verifications possible that software components fulfill RBAC-based system security properties. The corresponding cTLA framework facilitates the specification of component contract-based descriptions and of RBAC models while the framework theorems make the refinement proofs easier. Due to the compositionality of cTLA, however, not only the application but also the creation of the framework is supported. The cTLA process types of the component contract patterns and RBAC models were developed by a person within four weeks, while the theorems were set up and proven within two weeks. We intend to enhance the framework-based approach in order to support also other system security policy models. In particular, information

flow systems are of interest since they enable the detection of confidentiality attacks due to flaws in the — often complex — flow of data between components (cf. [26, 27]). This work will complement a less formal but highly automated approach [28] which is based on object-oriented modelling and on the application of graph rewrite rules on object models. Moreover, we plan to extend the global formal model, the specification patterns, and the theorems in order to facilitate also the specification and verification of flexible component-structured systems where components are added to or removed from an application during runtime.

References

1. Szyperski, C.: Component Software — Beyond Object Oriented Programming. Addison-Wesley Longman (1997)
2. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making Components Contract Aware. IEEE Computer **32** (1999) 38–45
3. Lindqvist, U., Jonsson, E.: A Map of Security Risks Associated with Using COTS. IEEE Computer **31** (1998) 60–66
4. Herrmann, P.: Trust-Based Procurement Support for Software Components. In: Proceedings of the 4th International Conference on Electronic Commerce Research (ICECR-4), Dallas, ATSMA, IFIP (2001) 505–514
5. Herrmann, P., Krumm, H.: Trust-adapted enforcement of security policies in distributed component-structured applications. In: Proceedings of the 6th IEEE Symposium on Computers and Communications, Hammamet, IEEE Computer Society Press (2001) 2–8
6. Herrmann, P., Wiebusch, L., Krumm, H.: State-Based Security Policy Enforcement in Component-Based E-Commerce Applications. In: Proceedings of the 2nd IFIP Conference on E-Commerce, E-Business & E-Government (I3E), Lisbon, Kluwer Academic Publisher (2002) 195–209
7. Fraser, T., Badger, L., Feldman, M.: Hardening COTS Software with Generic Software Wrappers. In: Proceedings of the 1999 IEEE Symposium on Security and Privacy, IEEE Computer Society Press (1999) 2–16
8. Herrmann, P.: Trust-Based Protection of Software Component Users and Designers. In Nixon, P., Terzis, S., eds.: Proceedings of the 1st International Conference on Trust Management. LNCS 2692, Heraklion, Springer-Verlag (2003) 75–90
9. Khan, K., Han, J., Zheng, Y.: A Framework for an Active Interface to Characterise Compositional Security Contracts of Software Components. In: Proceedings of the Australian Software Engineering Conference (ASWEC'01), Canberra, IEEE Computer Society Press (2001) 117–126
10. ISO/IEC: Common Criteria for Information Technology Security Evaluation. (1998) International Standard ISO/IEC 15408.
11. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST Standard for Role-Based Access Control. ACM Transactions on Information and System Security **4** (2001) 224–274
12. Herrmann, P., Krumm, H.: A Framework for Modeling Transfer Protocols. Computer Networks **34** (2000) 317–337
13. Vissers, C.A., Scollo, G., van Sinderen, M.: Architecture and specification style in formal descriptions of distributed systems. In Agarwal, S., Sabnani, K., eds.: Protocol Specification, Testing and Verification. Volume VIII., Elsevier, IFIP (1988) 189–204

14. Back, R.J.R., Kurkio-Suonio, R.: Decentralization of process nets with a centralized control. Distributed Computing (1989) 73–87
15. Herrmann, P., Krumm, H., Drögehorn, O., Geisselhardt, W.: Framework and Tool Support for Formal Verification of High Speed Transfer Protocol Designs. Telecommunication Systems **20** (2002) 291–310
16. Herrmann, P., Krumm, H.: Modular Specification and Verification of XTP. Telecommunication Systems **9** (1998) 207–221
17. Lamport, L.: The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems **16** (1994) 872–923
18. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters **21** (1985) 181–185
19. Heyl, C., Mester, A., Krumm, H.: ctc — A Tool Supporting the Construction of cTLA-Specifications. In Margaria, T., Steffen, B., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Number 1055 in Lecture Notes in Computer Science, Springer-Verlag (1996) 407–411
20. Graw, G., Herrmann, P., Krumm, H.: Constraint-Oriented Formal Modelling of OO-Systems. In: Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 99), Helsinki, Kluwer Academic Publisher (1999) 345–358
21. Ferraiolo, D.F., Barkley, J.F., Kuhn, D.R.: A Role Based Access Control Model and Reference Implementation within a Corporate Intranet. ACM Transactions on Information Systems Security **1** (1999) 34–64
22. Osborn, S.L., Sandhu, R.S., Munawer, Q.: Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. ACM Transactions on Information and System Security **3** (2000) 85–106
23. Zöllner, J., Federrath, H., Klimant, H., Pfitzmann, A., Piotraschke, R., Westfeld, A., Wicke, G., Wolf, G.: Modeling the security of steganographic systems. In: Proceedings of the 2nd Workshop of Information Hiding. LNCS 1525, Portland, Springer-Verlag (1998) 345–355
24. Schmitz, L.: The SalesPoint Framework — Technical Overview. Available via WWW: ist.unibw-muenchen.de/Lectures/SalesPoint/overview/english/TechDoc.htm (1999)
25. OBI Consortium: OBI Technical Specifications — Open Buying on the Internet. Draft release v2.1 edn. (1999)
26. Ferrari, E., Samarati, P., Bertino, E., Jajodia, S.: Providing flexibility in information flow control for object-oriented systems. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland (1997) 130–140
27. Myers, A.C., Liskov, B.: Complete, Safe Information with Decentralized Labels. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland (1998) 186–197
28. Herrmann, P.: Information Flow Analysis of Component-Structured Applications. In: Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC'2001), New Orleans, ACM SIGSAC, IEEE Computer Society Press (2001) 45–54