Analysis of Authorizations in SAP R/3*

Manuel Lamotte-Schubert and Christoph Weidenbach

Max Planck Institute for Informatics, Campus E1 4,
D-66123 Saarbrücken
{lamotte,weidenbach}@mpi-inf.mpg.de

Abstract. Today many companies use an ERP (Enterprise Resource Planning) system such as SAP R/3 to run their daily business ranging from financial issues down to the actual control of a production line. Already due to their sheer size, these systems are very complex. In particular, developing and maintaining the authorization setup is a challenge. The goal of our effort is to automatically analyze the authorization setup of an SAP R/3 system against business policies. To this end we formalize the processes, authorization setup as well as the business policies in first-order logic. Then, properties can be (dis)proven fully automatically with our theorem prover SPASS. We exemplify our approach on the purchase process, a typical constituent of any SAP R/3 installation.

1 Introduction

Enterprise Resource Planning (ERP) systems are built to integrate all facets of the business across a company including areas like finance, planning, manufacturing, sales, or marketing. The broader the functionality of such a system, the larger the number of users, the greater the dynamics of a company, the more complex is the administration of the authorizations. In particular, this applies to the SAP R/3 system offered by SAP [1]. In this paper we investigate the authorization setup of SAP R/3. Although SAP R/3 is not the newest release, the most recent release SAP ERP 6.0 actually shares the same authorization subsystem.

Our approach is depicted in Fig. 1. When a company decides to use an ERP system like SAP R/3, it first formulates its business as processes. For example, a typical purchase process starts with the creation of a purchase requisition out of a purchase request, followed by the release of such a requisition, and finally the transformation of the released requisition into the purchase that is eventually sent to a supplier. The processes directly induce an authorization concept. Very often each step of a process corresponds to a particular role of a company employee. For our example, the transformation of the released requisition into a purchase is a typical buyer activity. The development of processes and the authorization concept is guided by business policies. For our example, a business

* SAP, SAP R/3 and SAP ERP 6.0 are registered trademarks of SAP AG in Germany and in several other countries.

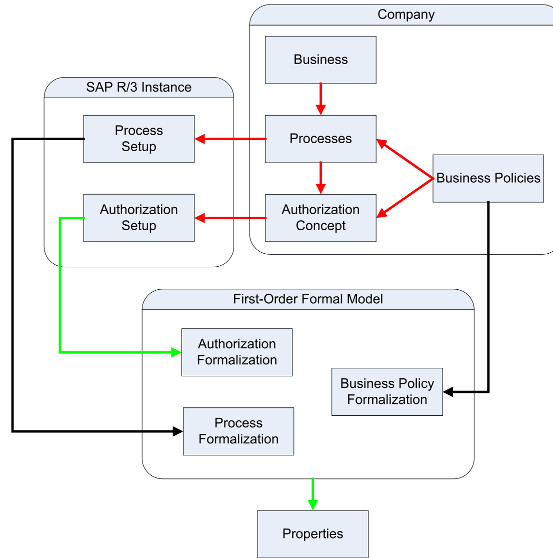

Fig. 1. Analysis of authorizations in SAP R/3

policy might require that the activity of creating a requisition and creating a purchase must always be separated, performed by different persons, and therefore must not be contained in one authorization role. This is a typical rule out of the *Segregation/Separation of Duties* (SoD) approach. Once the processes and authorization concept are defined, the configuration is implemented into an SAP R/3 instance leading to a corresponding process and authorization setup. Due to the sheer size of an SAP project, the number of processes, different employee roles and the highly dynamic development of such a system over time, it is practically impossible to guarantee the compliance of the business policies with the process and authorization setup. Furthermore, it is non-trivial to set up new authorization roles for employees following organizational changes in the business without destroying the overall compliance between the authorization setup and the business policies.

We suggest to solve this problem by first-order logic theorem proving. We model the process and authorization setup in first-order logic and automatically analyze it with respect to a first-order formulation of the business policies. SPASS always terminates for provable (it ends with a proof) and non-provable cases (it ends with a saturated set of clauses). The termination of SPASS enables the use of an abduction principle deriving missing facts. Then, defining new authorization roles can be solved by saturating abductive queries (Sect. 5). Formulating the processes and business policies has to be done by hand (Sect. 4). However, the authorization setup can be formulated automatically and we suggest a tool pipeline (Sect. 4.3). In practice, the changes to the authorization setup, e.g., caused by organizational changes in a company, cause the most headache to SAP autho-

rization administrators. Business policies and processes are less likely to change and if they change this is not done on a daily basis but by additional smaller SAP change/introduction projects. Therefore, our approach offers a reasonable amount of automatization. As an example SAP R/3 instance for studying the SAP internal process and authorization setup we used the SAP R/3 system run by the Max Planck society.

There have been other efforts to address the verification of the authorization setup in SAP R/3. SAP itself offers a tool collection for Governance, Risk and Compliance. The main difference to our approach is that these tools are only able to check compliance with respect to the transactions performed during concrete runs of the system and are not able to prove the overall compliance of the authorization setup with the business policies. Furthermore, there is no tool support for the business policy compliant generation of new authorization roles available up to now. Other efforts include the general verification of role-based access control principle together with constraints like SoD [2] but they are neither connected to the SAP R/3 system nor they do incorporate business processes. To the best of our knowledge, there has been no attempt so far to analyze the authorizations in SAP R/3 together with the business processes and business policies.

The paper is organized as follows. After explaining the basic first-order notation (Sect. 2) the SAP R/3 internal mechanisms are studied with respect to processes and authorizations in Sect. 3, followed by the formalization in first-order logic (Sect. 4). Due to space limitations we only explain important aspects of the developed first-order theory, hiding details that are not needed to understand the main ideas. Nevertheless, the overall formalization can be obtained from the SPASS homepage (`spass-prover.org`) in the "prototype and experiments" section. Our results on experiments are contained in Sect. 5 and the paper ends with a small conclusion and ideas for future work (Sect. 6).

2 Background

The formalization of the process, authorization setup and business policies is accomplished using first-order logic without equality. The following syntax definition as well as the semantics of the used language is taken from [3].

A first-order language is constructed over a signature $\Sigma = (\mathcal{F}, \mathcal{R})$, where $\mathcal{F}$ and $\mathcal{R}$ are non-empty, disjoint, in general infinite sets of function and predicate symbols, respectively. Every function or predicate symbol has some fixed arity. In addition to these sets that are specific for a first-order language, we assume a further, infinite set $\mathcal{X}$ of variable symbols disjoint from the symbols in Σ. Then the set of all *terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is defined as usual. A term not containing a variable is a *ground term*. If $t_1, \ldots, t_n$ are terms and $R \in \mathcal{R}$ is a predicate symbol with arity n, then $R(t_1, \ldots, t_n)$ is an *atom*. An atom or the negation of an atom is called *literal*. Disjunctions of literals are *clauses* where all variables are implicitly universally quantified. *Formulae* are recursively constructed over atoms and the operators $\supset$ (implication), $\equiv$ (equivalence), $\wedge$ (conjunction), $\vee$

(disjunction), $\neg$ (negation) and the quantifiers $\forall$ (universal), $\exists$ (existential) as usual. For convenience, we often write $\forall x_1, \ldots, x_n \,.\, \phi$ instead of $\forall x_1 \ldots . \forall x_n \,.\, \phi$ and analogously for the existential quantifier and assume the descending binding precedence $\neg$, $\wedge$, $\vee$, $\supset$, $\forall$, $\exists$.

The formal model uses predicate symbols whose first letter is always uppercase and the predicate itself is italic, e.g. the predicate *Access* is used to represent the authorization access relation for a user, the atom *Access*(`MUELLER`, `ME51N`) expresses that user `MUELLER` holds all rights to perform the transaction `ME51N`, the creation of a purchase requisition. Constants originating from SAP R/3 are always written in typewriter font, e.g. `MUELLER`. In general, function names start with a lowercase letter different from "*x*", e.g. the function *authObj* is used to represent an authorization (object). Variables are always prefixed with "*x*" and written lowercase, for example, *xu*, *xwrk*.

Although we do not explicitly define sorts, our formulae are actually many-sorted. For the explanation of our predicate and function usage we sometimes refer to these "implicit" sorts by putting them in square brackets. For example, the "declaration"

Access(< *user* >, *authObj*(< *auth object name* >, < *auth field* >, < *value* >))
explains that the first argument of an *Access* atom is a user term and the second argument a term representing an authorization (object).

3 SAP R/3 Setup and Business Policies in Detail

We use the SAP terminology throughout our work in order to describe the relevant aspects of the SAP R/3 system. The definition of terms adopted from SAP are written in italics.

Authorization Setup. The SAP R/3 authorization architecture is a complex structure and consists of several components interacting with each other. The key data structure is an *authorization*, an instance of an *authorization object*, that is eventually assigned via a *profile* to a user and typically grants the access to one particular action inside SAP R/3. In order to align authorizations with process steps, they are grouped in *roles*.

In detail, an authorization object is a named entity that holds one or more named authorization fields, similar to a class structure of a programming language. Together with appropriate field values, the authorization object constitutes the authorization. An authorization is therefore an instance of an authorization object, similar to the instance of a programming language class. The relation is shown in Fig. 2.

There are *single* and *composite roles* for the grouping of authorizations available. A single role groups authorizations whereas composite roles serve as containers for single roles. Single roles have a name and a list of authorizations. For example, Fig. 3 shows the structure of single role with name `ZBANF_WRK_INF_ED` by means of the concrete authorization `S_TCODE` with a field `TCD` and the value `ME51N`. The overall role `ZBANF_WRK_INF_ED` contains all authorizations required to create requisitions.

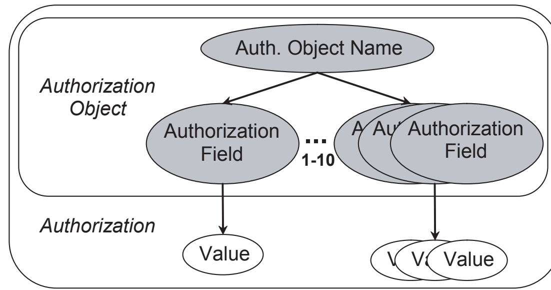

Fig. 2. Authorization object and authorization

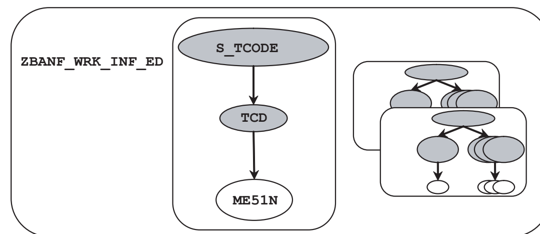

Fig. 3. Single role including authorizations

Single and composite roles generate profiles in SAP R/3 that are then eventually assigned to the user.

In our first-order model authorization objects become terms starting with function *authObj* and ground instances of those are authorizations. For the formulation of roles and profiles we use the respective predicates *SingleRole*, *CompositeRole*, and *UserProfile*. Mappings between objects are represented by functions and concrete values become constants.

Process Setup. A process is a consecutive flow of transactions in SAP R/3. Typically, the different actions are enabled by the creation or change of data inside the system due to preceding transactions. There is a unique identification code for any transaction, for example, `ME51N` stands for the transaction to create a requisition. If a user executes a particular transaction in the system then the effective authorizations from the users' authorization profile are checked. These checks are called *authorization checks*.

Each authorization check consists of two parts: (i) the presence of the required authorization object in the authorization profile associated with the user (for this purpose only the authorization object name is compared) is checked, and if this check succeeds, (ii) the required value(s) for the transaction are compared with the value(s) present in the value field(s) of the authorization assigned to the user. In particular, the second check succeeds if all value fields with the corresponding values of the object match to the required fields and values (AND-combination). A match can mean simple equality, e.g. the right to change data, or comparisons

with respect to some ordering, e.g. the amount of money is below some threshold. If one check fails, then the overall check of the authorization fails.

The first authorization check in every transaction is the check for the transaction code which is triggered by the SAP R/3 system before the actual transaction starts. The name of the corresponding authorization object for this check is `S_TCODE`. This object has only one authorization field `TCD` which serves as a container for the required transaction code. All further authorization checks are implemented in the transaction.

Example: Purchase Process and its Authorization Checks. The purchase process introduced in Sect. 1 is a typical constituent of the SAP R/3 system and is used in this paper as a running example. The creation of the requisition as well as the creation of the order are mapped by exactly one transaction in SAP R/3 whereas the release transaction implicitly additionally calls the transaction to view the requisition. Furthermore, releases of requisitions require release strategy settings done once at the initial configuration of an SAP R/3 system.

An SAP R/3 purchase requisition document is created to request the purchase of goods or services by calling the transaction `ME51N` in the SAP R/3 system. This transaction code is subject to the first authorization check and must be present as an authorization in the users' authorization profile.

The creation of a requisition needs to fill different fields, for example, the plant field for which the item is destined for. Some of these fields are protected by authorization objects, for example, the plant field is protected by the authorization object `M_BANF_WRK` with the two authorization fields `ACTVT` (activity) and `WERKS` (plant). The field activity requires the concrete value `01` (for "create") and the value for the plant field depends on the data entered in the requisition. If the data for the plant has been entered, the users' authorization to perform the action "create" for the entered plant is checked. The other protected fields document type and purchase group are protected in a similar way.

Release procedures for requisitions are used in SAP R/3 to approve requisitions which exceed a certain budget limit before they can be converted to an order. The SAP R/3 system uses so-called release strategies to achieve such approvals. A release strategy is an object that contains conditions for its application as well as a small process definition. This process defines the required actions to eventually release the requisition.

The order is the request to the supplier or another plant of the company to deliver the requisitioned material or service under terms and conditions agreed before. A released requisition is the prerequisite to create an order that is connected to the requisition. The authorization checks in the create order transaction are performed analogously to the checks occurring during the create requisition transaction.

Business Policies. Business policies are constraints on the business. A lot of business constraints follow best-practice approaches, for example, Segregation/Separation of Duties (SoD). This approach is considered in our work and requires that there is no single individual having the control over two or more phases of

a process, so that a deliberate fraud is more difficult to occur. In the purchase process, this means that the requisitioner must be distinct from the releasing person and the buyer.

On the SAP R/3 transactional level, it means that a single user is not allowed to have the authorizations to perform the appropriate transactions to create requisitions, release requisitions and orders for some plant, material group, purchasing group and organization[1].

4 SAP R/3 Formal Authorization Analysis

The SAP R/3 authorization system is formalized using first-order logic without equality. The formalization represents the process (we use the purchase process as example) and authorization setup as well as the formalized business policies. The prerequisite for the construction of the formalization is a snapshot of an SAP R/3 system, i.e. the formalization represents the state of the system at a given time.

For the goal of proving compliance (abduce changes) of the authorization setup with the business policies, we perform a number of abstractions, easing the size and depth of the formalization. We assume that we always have only one item per purchase requisition/order. We do not deeply model numbers, for example, amounts of money. Numbers are formalized as constants, intervals of numbers are also described as constants, for example *GREATER_1000_LESS_10000_EUR*, and the corresponding ordering relations between these constants are established. Within the authorization check procedure and the release strategy appliance checks, the SAP R/3 system uses a comprehensive pattern matching mechanism. For simplification, we formalized only exact matching and the asterisk symbol matching every required value. Composed values of an asterisk and a string are currently not supported by our formalization.

We formalized most of the SAP R/3 system parts in form of monadic predicates because SPASS offers particular reduction support for these predicates via soft typing [4]. The large set of authorization components like roles and profiles is modeled by the monadic predicates, while the assignment of these components to the users is represented by an implication. The set of process states in the SAP R/3 system is modelled by a set of predicates; and the abstraction of its dynamic behavior, which is relevant for authorization, is captured implications. The premise of such an implication represents the conditions for the process step while the conclusion stands for the effects after the execution of the corresponding transactions in this step. The form of business policies is individual and therefore the formalization of the policies depends on the type of the policy.

4.1 Authorization and Process Setup

The authorization setup layer consists of several predicates representing the way where authorizations are arranged and eventually assigned to a user. A single

[1] These are the properties protected by authorization objects.

role is modeled by the unary predicate *SingleRole*. The function *authObj* with arity 3 therein maps the authorization value to the authorization field of the authorization object. The authorization object together with the value represents the authorization that is then assigned to the single role by the binary function *singleRoleEntry*. Each authorization that is contained in some SAP R/3 single role results in a *SingleRole* atom in the formalization.

$$\begin{array}{l} \mathit{SingleRole}(\mathit{singleRoleEntry}(<\mathit{single\ role\ name}>, \\ \qquad \mathit{authObj}(<\mathit{auth\ object\ name}>, <\mathit{auth\ field}>, <\mathit{value}>))) \end{array}$$

A composite role is modeled by the unary predicate *CompositeRole*. The function *compositeRoleEntry* therein associates the single role given by its name with the composite role.

$$\begin{array}{l} \mathit{CompositeRole}(\mathit{compositeRoleEntry}(<\mathit{composite\ role\ name}>, \\ \qquad <\mathit{single\ role\ name}>)) \end{array}$$

The effective authorizations associated with a user are stored in the authorization profile. This profile is modeled by the unary predicate *UserProfile*. The function *userProfileEntry* maps the authorizations given by the function *authObj* to the user; any authorization check looks for the required authorization only in the users authorization profile.

$$\begin{array}{l} \mathit{UserProfile}(\mathit{userProfileEntry}(<\mathit{user}>, \\ \qquad \mathit{authObj}(<\mathit{auth\ object\ name}>, <\mathit{auth\ field}>, <\mathit{value}>))) \end{array}$$

The following formula exactly models the mechanism of the SAP R/3 user authorization profile creation. The assignment of a role to a user implies the assignment of the appropriate generated authorization profile. Whenever a single role or composite role is going to be assigned to a user via the predicate *Holds*, the authorization part is extracted and the corresponding authorization profile entry (representing the effective authorization) for the user is created:

$$\begin{array}{l} \forall\ \mathit{xu, xpn, xsrn, xcrn, xaon, xaof, xav}\ . \\ \quad (\mathit{SingleRole}(\mathit{singleRoleEntry}(\mathit{xsrn}, \mathit{authObj}(\mathit{xaon, xaof, xav}))) \wedge \\ \quad \mathit{Holds}(\mathit{xu, xsrn})) \vee \\ \quad (\mathit{CompositeRole}(\mathit{compositeRoleEntry}(\mathit{xcrn, xsrn})) \wedge \\ \quad \mathit{SingleRole}(\mathit{singleRoleEntry}(\mathit{xsrn}, \mathit{authObj}(\mathit{xaon, xaof, xav}))) \wedge \\ \quad \mathit{Holds}(\mathit{xu, xcrn})) \\ \supset \\ \quad \mathit{UserProfile}(\mathit{userProfileEntry}(\mathit{xu}, \mathit{authObj}(\mathit{xaon, xaof, xav}))) \end{array}$$

The authorization check result – access or decline – is represented in our first-order formalization by the binary predicate *Access*. If the atom *Access* is valid, the access to the function or data protected by the authorization object is granted. Otherwise, it is not. In other words, a valid instance of the following *Access* atom expresses that the user <*user*> has successfully passed the

authorization check of the appropriate authorization which is denoted by the authorization object with its field and value.

$$Access(< user >, authObj(< auth\ object\ name >, < auth\ field >, < value >))$$

In the authorization check procedure, the required authorization object information together with the appropriate authorization value are compared to the authorization present in the users authorization profile. All properties, namely the authorization object name, the field and the value must be equal in the check in order to succeed. This is modeled by the following implication. If the required authorization is present in the user authorization profile, then the access to this authorization is granted.

$$\begin{array}{l} \forall\ xu,\ xaon,\ xaof,\ xav\ . \\ \quad UserProfile(userProfileEntry(xu, authObj(xaon,\ xaof,\ xav))) \\ \supset \\ \quad Access(xu, authObj(xaon,\ xaof,\ xav)) \end{array}$$

Example: Purchase Process. The formalization of the transaction layer with its authorization checks is done manually in our example. We have introduced an additional layer by overloading the predicate symbol *Access*. The following transition shows the abstraction for the transaction “create a requisition”. It groups all authorization checks occurring during the execution of the transaction.

$$\begin{array}{l} \forall\ xu,\ xwrk,\ xbsa,\ xekg\ . \\ \quad Access(xu, authObj(\texttt{S_TCODE}, \texttt{TCD}, \texttt{ME51N})) \wedge \\ \quad Access(xu, authObj(\texttt{M_BANF_WRK}, \texttt{ACTVT}, \texttt{01})) \wedge \\ \quad Access(xu, authObj(\texttt{M_BANF_WRK}, \texttt{WERKS}, xwrk)) \wedge \\ \quad Access(xu, authObj(\texttt{M_BANF_BSA}, \texttt{ACTVT}, \texttt{01})) \wedge \\ \quad Access(xu, authObj(\texttt{M_BANF_BSA}, \texttt{BSART}, xbsa)) \wedge \\ \quad Access(xu, authObj(\texttt{M_BANF_EKG}, \texttt{ACTVT}, \texttt{01})) \wedge \\ \quad Access(xu, authObj(\texttt{M_BANF_EKG}, \texttt{EKGRP}, xekg)) \\ \supset \\ \quad Access(xu, \texttt{ME51N}) \end{array}$$

The first check represents the check of the transaction code (ME51N) carried out by the SAP R/3 system at the start of every transaction. The authorization objects M_BANF_WRK, M_BANF_BSA, and M_BANF_EKG check the plant, document type and purchase group, respectively. They have constants in the first field (ACTVT) checking the type of action (01 stands for “create”, 02 stands for “change”, 03 stands for “view”) and variables in their second field standing for the values of the corresponding fields: plant, document type and purchase group. If the atom $Access(xu, \texttt{ME51N})$ holds, then the user is allowed to execute the transaction in at least one instance, for example for one plant (variable *xwrk*), one document type (variable *xbsa*) and one purchase group (variable *xekg*). Later in the context of the requisition creation, when the exact values are known from the requisition, the values of the variables have to be evaluated and checked again.

The previously mentioned additional transactional layer for authorization checks makes it more comfortable to model the formalization of the purchase process steps. The purchase process starts with the existence of a purchase request whose data is then entered into the SAP R/3 system by the purchase requisitioner. After the data has been entered, the request has become an SAP R/3 requisition object that is represented by the atom *RequisitionCreated*. An instance of this atom contains all needed details.

$$\begin{array}{l} \textit{RequisitionCreated}(\langle\textit{user}\rangle, \langle\textit{document type}\rangle, \langle\textit{position}\rangle, \\ \qquad \langle\textit{material}\rangle, \langle\textit{plant}\rangle, \langle\textit{purchasing group}\rangle, \\ \qquad \langle\textit{purchasing organization}\rangle, \langle\textit{material group}\rangle, \langle\textit{price}\rangle, \langle\textit{id}\rangle) \end{array}$$

The following implication represents the creation of the corresponding SAP R/3 requisition object. The predicate *Requisition* is an arbitrary purchase request for an item and the predicate *RequisitionCreated* is represents this item in the SAP R/3 system, created by the user denoted by the variable *xu*. This user *xu* needs access to the create transaction (`ME51N`). As mentioned, the values of the variables *xwrk*, *xbsa*, *xekg* are again subject to authorization checks because at this point the values of the variables are known (namely the values from the requisition that is going to be created). This results in the following formula.

$$\begin{array}{l} \forall\ xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt, xid\ . \\ \quad \textit{Requisition}(xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt, xid) \wedge \\ \quad \textit{Access}(xu, \texttt{ME51N}) \wedge \\ \quad \textit{Access}(xu, \textit{authObj}(\texttt{M_BANF_WRK}, \texttt{WERKS}, xwrk)) \wedge \\ \quad \textit{Access}(xu, \textit{authObj}(\texttt{M_BANF_BSA}, \texttt{BSART}, xbsa)) \wedge \\ \quad \textit{Access}(xu, \textit{authObj}(\texttt{M_BANF_EKG}, \texttt{EKGRP}, xekg)) \\ \supset \\ \quad \textit{RequisitionCreated}(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, \\ \qquad\quad xgswrt, xid) \end{array}$$

A more complex and interesting step in the purchase process is the release of a requisition where a release strategy has applied. The function *property* relates a value to a property name and represents a condition property for the application of some release strategy. The class construction is eventually used to group several properties belonging to a release strategy.

$$\begin{array}{l} \textit{ReleaseStrategy}(<\textit{release strategy name}>, \\ \qquad \textit{class}(<\textit{characteristics class name}>, \\ \qquad\qquad \textit{property}(<\textit{property name}>, <\textit{value}>))) \end{array}$$

Release strategies consist of one or more single release steps which are declared by the atom *ReleaseRequirement*. This atom groups the strategy name and the required code for each step.

$$\textit{ReleaseRequirement}(<\textit{release strategy name}>, <\textit{release code}>)$$

Figure 4 shows the formalization of one release step for an existing requisition object. The existence of the requisition is checked by the first atom *RequisitionCreated* in the premise. Subsequent atoms address the application checks of the

release strategy *xfrgstrat* for which the characteristics denoted by the variables *xekg* (purchasing group), *xwrk* (plant), and *xgswrt* (total amount of money of the requisition) are used. The predicate *ReleaseRequirement* retrieves the release code for the release step in the release strategy and is then subject to an authorization check. In order to proceed with the release step, the user, denoted by the variable *xu2*, needs authorizations for the release (with the code *xfrgco*) as well as for the transaction (`ME54N`) in order to perform the release step. Please note that the user performing the release step (*xu2*) is different from the user who has created the requisition (*xu1*) which is enforced by the business policies (see Sect. 4.2). The conclusion expresses the fact that the user *xu2* has performed the release step with the code *xfrgco* in the overall release of the requisition.

$$
\begin{array}{l}
\forall\ \mathit{xu1, xu2, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt,} \\
\qquad\quad \mathit{xfrgstrat, xfrgco, xcl, xid}\ . \\
\quad \mathit{RequisitionCreated}(\mathit{xu1, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl,} \\
\qquad\quad \mathit{xgswrt, xid}) \wedge \\
\quad \mathit{ReleaseStrategy}(\mathit{xfrgstrat}, \mathit{class}(\mathit{xcl}, \mathit{property}(\mathtt{FRG_CEBAN_EKGRP}, \mathit{xekg}))) \wedge \\
\quad \mathit{ReleaseStrategy}(\mathit{xfrgstrat}, \mathit{class}(\mathit{xcl}, \mathit{property}(\mathtt{FRG_CEBAN_WERKS}, \mathit{xwrk}))) \wedge \\
\quad \mathit{ReleaseStrategy}(\mathit{xfrgstrat}, \mathit{class}(\mathit{xcl}, \mathit{property}(\mathtt{FRG_CEBAN_GSWRT}, \mathit{xgswrt}))) \wedge \\
\quad \mathit{ReleaseRequirement}(\mathit{xfrgstrat, xfrgco}) \wedge \\
\quad \mathit{Access}(\mathit{xu2}, \mathit{authObj}(\mathtt{M_EINK_FRG}, \mathtt{FRGCO}, \mathit{xfrgco})) \wedge \\
\quad \mathit{Access}(\mathit{xu2}, \mathtt{ME54N}) \wedge \\
\quad \mathit{Access}(\mathit{xu2}, \mathit{authObj}(\mathtt{M_BANF_WRK}, \mathtt{WERKS}, \mathit{xwrk})) \wedge \\
\quad \mathit{Access}(\mathit{xu2}, \mathit{authObj}(\mathtt{M_BANF_BSA}, \mathtt{BSART}, \mathit{xbsa})) \wedge \\
\quad \mathit{Access}(\mathit{xu2}, \mathit{authObj}(\mathtt{M_BANF_EKG}, \mathtt{EKGRP}, \mathit{xekg})) \\
\supset \\
\quad \mathit{RequisitionReleasedStep}(\mathit{xu2, xfrggr, xfrgstrat, xfrgco, xbsa, xpos, xmat,} \\
\qquad\quad \mathit{xwrk, xekg, xekorg, xmatkl, xgswrt, xid})
\end{array}
$$

Fig. 4. Single release step in the SAP R/3 purchase process

Concerning the overall release of a requisition, there are further formulae which define the required single release steps for a complete release of the requisition.

The released requisition is eventually the precondition to create an order object in SAP R/3 that is connected to the requisition. The formalization of this step is analogous to the creation of a requisition.

4.2 Business Policies

The SoD business policy for the purchase process expresses that there should be no user having the control over two or more phases of a process. Very often in smaller companies, this is relaxed into a less strict requirement stating that there should be no user who is allowed to perform the complete purchase process in one instance. The relaxed version of SoD is formalized by the following formula. Starting from the purchase request, there are no values for which the user *xu* can perform the three steps of the purchase process.

$$\neg\exists\ xu, xbsa, xwrk, xekg, xekorg, xmatkl, xgswrt, xpos, xmat, xid\ .\\ RequisitionCreated(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt, xid) \wedge\\ RequisitionReleased(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt, xid) \wedge\\ OrderCreated(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt, xid)$$

4.3 Automatic Authorization Extraction

The authorization checks occurring in the SAP R/3 system can be extracted by looking directly into the source code and by exploring the connection between the transaction and the associated program code. The authorization check at the beginning of a transaction can be read from an internal data dictionary. This is done by the transaction `SE93` which is used to manage the association between the transaction code, the program code and the authorization check. All further checks are implemented into the program code using the *AUTHORITY-CHECK* statement. For convenience, we used the SAP R/3 System Trace tool which monitors, among other things, the authorization checks taking place during the execution of a transaction.

The extraction of the SAP R/3 system users and its authorizations is achieved using the User Information System[2]. It is able to report information about users, their roles and profiles as well as information about authorizations, authorization objects or transactions. The result of a query to this information system can be easily stored in text-format (see Fig. 5) and lists, for example, all authorizations present in a profile.

5 Results

We used the theorem prover SPASS, Version *3.0* [5] for our experiments. The theory containing the SAP R/3 general authorization structure and its instantiation to the purchase process consists of 156 formulae with a size of 41 KByte resulting in 177 clauses. The experiments ran on a Dell PowerEdge 1950 server

[2] Transaction `SUIM`

```
Profile
  |
  ---   Z:EK1_INFO   <PRO>
      |
      |--   M_BEST_WRK <OBJ> Plant in Purchase Order
      |   |
      |   ---   M_BEST_WRKAL <AUT> Plant in Purchase Order
      |       |
      |       |--   ACTVT      <FLD> Activity
      |       |   |
      |       |   ------*
      |       |
      |       ---   WERKS      <FLD> Plant
      |           |
      |           ------INFO
      ...
```

Fig. 5. Export of authorizations from SAP R/3

running at 3.14 GHz equipped with 16 GB RAM, 64-bit Debian Linux, Kernel 2.6.24.2.1.amd64-smp.

One of the key results is that the overall formalization can be finitely saturated. This is mainly due to the fact that there is no recursion over the business processes, and consequently, ordering mechanisms are sufficient for saturation. Our experiments also include the user authorization data. SPASS always terminated within the experiments, ending with either a saturated set of clauses or a proof. A finite saturation means that the given conjecture could not be proven and therefore doesn't hold. If no conjecture was given, it states there is no contradiction in the input formulae and consequently the authorization setup and the business policies are compatible. The fact that SPASS always terminates is also an important prerequisite for the actual development of the theory as it enables inspection of models and detection of accidental inconsistencies.

Every experiment run took less than 20 seconds. The saturation of the input theory, including the user data and business policy, took 13 seconds with SPASS run with default settings. It can be tweaked by predefining a particular selection strategy to less than 1 second.

The ability to run a variety of different queries in addition to the general inspection of the setup was also one of the original motivations to do this work. Having SPASS terminating on queries further enables the use of an abduction principle [6, 7]. We give SPASS the query to be proven and then the saturated clauses out of the query represent a set of abductive answers. This is complete for the propositional case as stated in [6]. Completeness is open for the full first-order case. For example, it is interesting whether a particular user, e.g., `MUELLER`

in our running example, is able to perform the step to create a requisition, maybe for the given plant INF. Such a conjecture is formalized and fed to SPASS as the conjecture:

$$\exists\ \textit{xbsa, xekg, xekorg, xmatkl, xgswrt, xpos, xmat, xid}\ .\ \textit{Requisition}(\textit{xbsa, xpos, xmat}, \texttt{INF}, \textit{xekg, xekorg, xmatkl, xgswrt, xid}) \supset \textit{RequisitionCreated}(\texttt{MUELLER}, \textit{xbsa, xpos, xmat}, \texttt{INF}, \textit{xekg, xekorg, xmatkl, xgswrt, xid})$$

In our example setup the conjecture holds and can be proved in less than 1 second.

Removing MUELLER's access rights to the corresponding transaction ME51N from the theory and rerunning the above conjecture results in a saturation without proof in 8 seconds. Now the purely negative clauses resulting from the query can be interpreted as abductive answers to the query. For example, the generated clause

$$\neg \textit{Access}(\texttt{MUELLER}, \texttt{ME51N})$$

expresses that the right to execute transaction ME51N is missing in order to successfully create a requisition.

6 Conclusion and Future Work

This paper has presented an effort to the automatic analysis of an SAP R/3 process and authorization set up with respect to given business policies using the purchase process as a case study.

To accomplish automatic verification, the SAP R/3 process setup, the authorization setup and the business policies have been formalized in first-order logic. The formalization decisions were taken from a detailed analysis of the SAP R/3 system instance run by the Max Planck society. We could show that the developed formalization can be automatically analyzed by SPASS. Any proof attempt with SPASS we have done in this context terminated. We can automatically check compliance of business policies, properties with respect to specific user authorization configurations as well as automatically abduce compliant changes to the authorization set up.

There are a number of open questions left for future work. Our model of numbers by first-order constants could be overcome by using SPASS(LA) [8], our currently experimental prover for the hierarchic extension of linear arithmetic by first-order logic. For the first-order formula class presented in this paper as well as for such an extended first-order formula class over linear arithmetic decidability is open.

Eventually, it is an open question how our model scales with respect to a more integrated formalization of an SAP R/3 instance. In our example, we analyzed only the purchase process and up to ten users while a typical instance has about 50–200 processes and up to several thousand users. We are optimistic that this

is not out of range to first-order theorem proving because processes as well as users can be analyzed almost independently.

The concrete formalization of authorizations differs for individual (non-SAP) software systems. However, theoretic aspects of our approach like termination, scalability or completeness in verification tasks remain similarly and can be transferred to other systems.

References

1. SAP Press Release: SAP Holds Top Rankings in Worldwide Market Share (July 2008)
   http://www.sap.com/about/newsroom/news-releases/press.epx?pressid=9913.
2. Yuan, C., He, Y., He, J., Zhou, Z.: A verifiable formal specification for rbac model with constraints of separation of duty. In: Information Security and Cryptology. Volume 4318 of LNCS., Springer (2006) 196–210
3. Nonnengart, A., Weidenbach, C.: 6. In: Computing small clause normal forms. Volume 1. Elsevier, Amsterdam, the Netherlands (January 2001) 335–367
4. Ganzinger, H., Meyer, C., Weidenbach, C.: Soft typing for ordered resolution. In McCune, W., ed.: Proceedings of the 14th International Conference on Automated Deduction (CADE-14). Volume 1249 of Lecture Notes in Computer Science., Townsville, Australia, Springer (1997) 321–335
5. Weidenbach, C., Schmidt, R., Hillenbrand, T., Rusev, R., Topic, D.: System description: Spass version 3.0. In Pfenning, F., ed.: CADE-21 : 21st International Conference on Automated Deduction. Volume 4603 of LNCS., Springer (2007) 514–520
6. Dimova, D.: Propositional abduction. Bachelor's thesis, Universität des Saarlandes (September 2007)
7. Eiter, T., Makino, K.: On computing all abductive explanations. In: Eighteenth national conference on Artificial intelligence, Menlo Park, CA, USA, American Association for Artificial Intelligence (2002) 62–67
8. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition modulo linear arithmetic: SUP(LA). In Ghilardi, S., Sebastiani, R., eds.: Frontiers of Combining Systems. 7th International Symposium FroCos 2009, Proceedings. LNCS, Springer (2009) Accepted for publication.