Automatic Authorization Analysis

Manuel Lamotte-Schubert

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Saarbrücken
2015

Day of Colloquium	September 18th, 2015
Dean	Prof. Dr. Markus Bläser
Chairman	Prof. em. Dr. Dr. h.c. Reinhard Wilhelm
Supervisor and First Reviewer	Prof. Dr. Christoph Weidenbach
Second Reviewer	Dr. habil. Peter Baumgartner
Scientific Assistant	Dr. Swen Jakobs

Abstract

Today many companies use an ERP (Enterprise Resource Planning) system such as the SAP system to run their daily business ranging from financial issues down to the actual control of a production line. These systems are very complex from the view of administration of authorizations and include a high potential for errors. The administrators need support to verify their decisions on changes in the authorization setup of such systems and also assistance to implement planned changes error-free.

First-order theorem proving is a reliable and correct method to offer this support to administrators at work. But it needs on the one hand a corresponding formalization of an SAP ERP system instance in first-order logic, and on the other hand, a sound and terminating calculus that can deal with the complexity of such systems. Since first-order logic is well-known to be undecidable in general, current research deals with the challenge of finding suitable and decidable sub-classes of first-order logic which are then usable for the mapping of such systems.

This thesis presents a (general) new decidable first-order clause class, named $\mathcal{BDI}$ (Bounded Depth Increase), which naturally arose from the requirement to assist administrators at work with real-world authorization structures in the SAP system and the automatic proof of properties in these systems. The new class strictly includes known classes such as $\mathcal{PVD}$. The arity of function and predicate symbols as well as the shape of atoms is not restricted in $\mathcal{BDI}$ as it is for many other classes. Instead the shape of "cycles" in resolution inferences is restricted such that the depth of generated clauses may increase but is still finitely bound. This thesis shows that the Hyper-resolution calculus modulo redundancy elimination terminates on $\mathcal{BDI}$ clause sets. Further, it employs this result to the Ordered Resolution calculus which is also proved to terminate on $\mathcal{BDI}$, and thus yielding a more efficient decision procedure which is able to solve real-world SAP authorization instances. The test of conditions of $\mathcal{BDI}$ have been implemented into the state-of-the art theorem prover SPASS in order to be able to detect decidability for any given problem automatically. The implementation has been applied to the problems of the TPTP Library in order to detect potential new decidable problems satisfying the $\mathcal{BDI}$ class properties and further to find non-terminating problems "close" to the $\mathcal{BDI}$ class having only a few clauses which are responsible for the undecidability of the problem.

Zusammenfassung

Viele Unternehmen verwenden heutzutage ERP (Enterprise Resource Planning) Systeme wie das SAP System zur Unterstützung des täglichen Geschäfts angefangen vom Rechnungswesen bis hin zur Steuerung einer Fertigungslinie. Diese Systeme sind hinsichtlich der Verwaltung von Berechtigungen sehr komplex und besitzen deswegen eine hohe Fehleranfälligkeit. Administratoren benötigen deswegen sowohl Unterstützung, um ihre Entscheidungen bei Änderungen im Berechtigungs-Setup zu verifizieren und auch Hilfe, um geplante Änderungen an Berechtigungen fehlerfrei umsetzen zu können.

Theorembeweisen in Prädikatenlogik ist eine zuverlässige und korrekte Methode, um Administratoren die notwendige Unterstützung bei der Arbeit zu bieten. Sie benötigt jedoch auf der einen Seite eine entsprechende Formalisierung einer SAP ERP System Instanz in Prädikatenlogik, und auf der anderen Seite einen korrekten und terminierenden Kalkül der mit der Komplexität solcher Systeme umgehen kann. Da Prädikatenlogik bekanntlicherweise im Allgemeinen unentscheidbar ist, beschäftigt sich die aktuelle Forschung mit der Herausforderung, geeignete und entscheidbare Teilklassen der Prädikatenlogik zu finden, die dann für die Abbildung solche Systeme benutzbar sind.

Diese Arbeit stellt eine (allgemeine) neue entscheidbare Klauselklasse in Prädikatenlogik mit dem Namen $\mathcal{BDI}$ (Bounded Depth Increase) vor, die auf natürlichem Wege aus der Anforderung zur Unterstützung der Administratoren bei der praxisorientierten Arbeit mit Berechtigungsstrukturen im SAP System und dem automatischen Nachweis von Eigenschaften in solchen Systemen entstand. Die neue Klasse enthält bereits bekannte Klassen wie $\mathcal{PVD}$. Die Stelligkeit von Funktions- und Prädikatssymbolen wie auch die Gestalt der Atome ist in $\mathcal{BDI}$ nicht beschränkt, wie es bei vielen anderen Klassen der Fall ist. Stattdessen wird die Gestalt von Zyklen in Resolutionsinferenzen derart beschränkt, dass die Tiefe von generierten Klauseln zwar ansteigen kann, aber letztendlich begrenzt ist. Diese Arbeit zeigt, dass der Hyperresolutions-Kalkül zusammen mit der Eliminierung von Redundanzen auf $\mathcal{BDI}$ Klauselmengen terminiert. Zusätzlich wird das Ergebnis übertragen auf den Kalkül der geordneten Resolution, für die ebenfalls die Terminierung auf $\mathcal{BDI}$ bewiesen wird und damit eine effizientere Enscheidungsprozedur liefert, die für praxisorientierte SAP Instanzen anwendbar ist. Der Test der Bedingungen für $\mathcal{BDI}$ wurden im aktuellen Theorembeweiser SPASS implementiert, um die Entscheidbarkeit für ein beliebiges Problem automatisch feststellen zu können. Die Implementierung wurde auf die aktuelle TPTP-Problem-Bibliothek angewendet, um neue entscheidbare Probleme zu finden, die den Anforderungen der $\mathcal{BDI}$

Klasse genügen und weiterhin nicht terminierende “beinahe”-$\mathcal{BDI}$ Probleme zu ermitteln, bei denen nur wenige Klauseln verantwortlich für die Unentscheidbarkeit des Problems sind.

Acknowledgements

This work would not have been possible without the persistent support and continuous motivation of my supervisor Prof. Dr. Christoph Weidenbach. I would like to thank him for his scientific guidance, patience and cooperation in all aspects accompanying this thesis. I'm also grateful to Dr. habil. Peter Baumgartner, who invited me for a visit at NICTA Canberra, Australia, where I spent 4 months doing research on business rules. Furthermore, I would like to thank my colleagues at the Max Planck Institute for Informatics with whom I often had supporting discussions.

I thank the anonymous reviewers of the publications that are incorporated in this thesis. Their comments have helped me considerably in obtaining a fresh view on some aspects and in improving both the quality of my work and its presentation.

Especially, I would like to thank my wife Agata Schubert and my family who supported me over the whole time at the Max Planck Institute for Informatics and encouraged me to finish this thesis.

Finally, I thank the (yet to come) readers of this thesis for their interest in my work. I sincerely hope that you can benefit from it for your own work.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Computer systems have been invented to help people in many areas of everyday life. Especially, there are powerful software systems aiming to support the operation of a business. One typical software of this area is called Enterprise Resource Planning (ERP) system. Typically, ERP systems are built to integrate almost 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 well-known SAP ERP system (formerly known as SAP R/3) offered by SAP SE[1].

The typical scenario of a company using an ERP system like SAP ERP is depicted in Figure 1.1. When a company decides to use a system like SAP ERP, it first formulates its business as processes. A typical business process is the purchase process that is used as a case study in this thesis. It starts with the creation of a purchase requisition out of the requirement for an asset, 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 who is then responsible for the delivery of the asset. 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. On the process layer, the company also decides on regulations and constraints which are typically called business policies. The development of processes and the authorization concept is guided by business policies. For example, a business 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 ERP instance leading to a corresponding process and authorization setup. The authorization setup exactly defines for a user whether or not he/she is authorized to execute certain functions in the system.

The initial setup of a new SAP system instance typically requires a so-called

[1]SAP®, SAP® R/1®, SAP® R/2®, SAP® R/3®, and SAP Netweaver™ are the trademarks or registered trademarks of SAP SE in Germany and in several other countries.

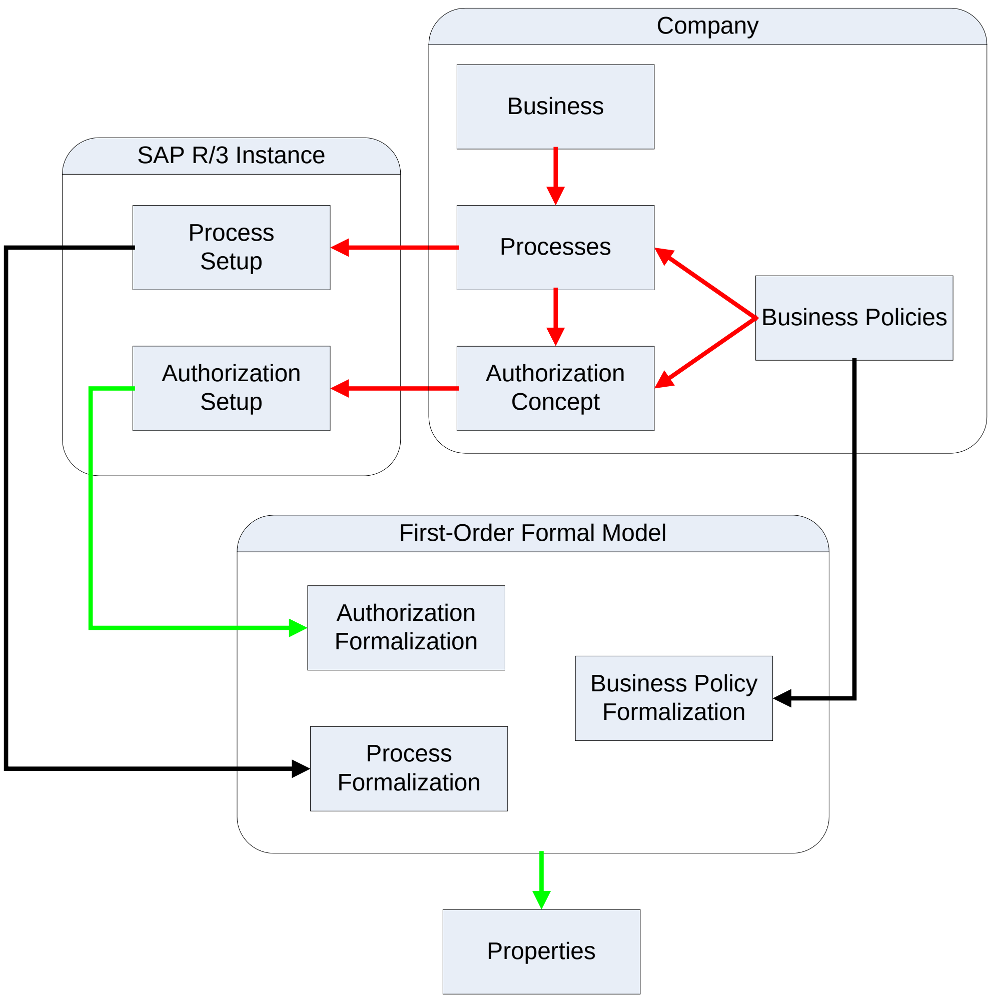


Figure 1.1: Analysis of authorizations in SAP ERP.

Customizing to meet the business needs of the company. Therefore, two SAP ERP implementations as used by two different companies will usually never look the same. The Customizing is mostly carried out by separate companies whose daily business is to configure all the customizing settings, and to setup the corresponding authorization concept. Correspondingly, these companies usually have appropriate routines for these tasks which try to ensure that no errors occur in the initial setup. However, as soon as the system is productive, it requires permanent and individual maintenance. 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 or introduction projects. However, for example, changes might be necessary because of changes in the organizational structure of the company or the employees change. Such changes result in changes of the authorizations and, because of the complex structure and the sheer size of most of today's SAP systems, it is practically impossible to guarantee the compliance of the business policies with the process and authorization setup afterwards by hand without the support of rigid tools and programs. 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. In practice, especially changes to the authorization setup, for example, caused by organizational changes in a company, cause the most headache to SAP authorization administrators. Who has currently access or permission to retrieve or change certain sensitive data? Does the system (still) comply with the company's business policies initially and especially after the change? Such questions are a daily occurrence for SAP system administrators who have to manage a vast number of users, roles, and authorizations and it's not trivial to give correct answers to these critical questions. This is the point where automatic tools come into play which support the administrators by verifying their decisions and assisting them to implement planned changes error-free.

A suitable and particularly reliable solution to correctly answer the previous questions is automatic theorem proving (ATP) and its corresponding tools. ATP is located in the area of automated reasoning and is used to mathematically prove theorems by means of computer programs. One part in this area – which is used in this thesis – is first-order theorem proving. It is characterized by a restricted logic (in contrast to higher logic) that is still expressive enough to allow the specification of arbitrary computable problems, often in a reasonably natural and intuitive way [38]. This formality is the underlying strength of ATP: There is no ambiguity in the statements of the problem, as it is often the case when using a natural language such as English. On the other hand, the problem of validity of formulas in first-order logic is undecidable [9] but semi-decidable, and a number of sound and complete calculi have been developed, to enable fully automated systems for the verification of first-order problems. One of today's current challenges in ATP is the research to obtain sound and terminating calculi that are able to deal with first-order formalizations representing new and complex systems like the SAP ERP system.

In order to use ATP for answering questions and proving correctness properties in the SAP authorization context, it is necessary to transform the relevant authorization parts of a given SAP system (instance) into a formalized first-order logic representation. This transformation is done manually for the business policies and the process setup because these parts are rather individual for each system (indicated by black arrows in Figure 1.1). The formalization of the remaining part – the authorization setup – is not trivial but automatable (green arrow in Figure 1.1) and almost the same for all SAP system instances because of the fixed underlying (authorization) structure. Once the first-order representation is at hand, verification and questioning tasks can be started by means of an automated theorem prover tool.

1.2 Contribution

The verification of formulas or a set of clauses representing an SAP authorization scenario instance by means of automatic theorem proving is only useful and feasible, if the input (often called “problem”) given to any theorem prover, i.e., the formulas or set of clauses, is decidable. The satisfiability problem can be decided for clause sets representing the previously mentioned SAP authorization layer if it exists a known decidable class and the problem satisfies the conditions of the respective clause class. However, if no such class exists as in the case for the SAP authorization problems, a characteristics clause class with a defined structure has to be determined and proved to be terminating. Termination is ensured if the length (number of literals in a clause) and depth (maximal depth of a literal in a clause) of newly generated clauses can be finitely bound [20].

This thesis continues previous work that has been published in my Masters Thesis [23]. It consists of the analysis of real-word authorization structures as they occur, e.g., in enterprise relationship systems like the SAP system. From this work naturally arose the main contribution of this thesis: To determine the characteristics and common structure on clause sets representing (first-order) SAP authorization instances which are stated in the form of a general new first-order clause class named *Bounded Depth Increase* ($\mathcal{BDI}$).

Usually, the term depth of newly generated clauses by the respective resolution (superposition) strategy does not grow for most of the so far studied decidable clause classes (for example, [6, 35, 12, 17, 19, 1]). For the new clause class $\mathcal{BDI}$ defined in this thesis, the term structure of clauses belonging to the class is not restricted at all. Further, predicates may have an arbitrary number of arguments. An overall bounded term depth is guaranteed by restricting the form of recursive definitions for predicates that occur in the clause set. For the $\mathcal{BDI}$ class any considered resolvent has a depth of at most $2n$ where n is the maximal depth of a clause in the initial set (Theorem 5.11). By requiring that all variables occurring in a positive literal of a clause also occur in a negative one of that clause, (positive) Hyper-resolution generates only ground clauses (Lemma 5.9), implying together the depth bound termination and therefore decidability of the $\mathcal{BDI}$ class (together with Factoring as a reduction rule). This result is then generalized from Hyper-resolution to Ordered Resolution to obtain a more efficient decision procedure. The main results, the class definition, termination proof and generalization, have been published in several forms in [24, 26, 25].

Because the $\mathcal{BDI}$ class characteristics have been constructed out of the first-order representation of SAP authorization instances, these problems clearly satisfy the $\mathcal{BDI}$ class requirements and can be decided by Hyper-resolution or even more efficient by Ordered Resolution (together with Factoring as a reduction rule).

For a better understanding of the $\mathcal{BDI}$ properties (Section 5.2), consider the following $\mathcal{BDI}$ clause set

$$\begin{array}{lrcl}
(1) & & \rightarrow & P(f(a), h(a), a) \\
(2) & P(x, y, z) & \rightarrow & Q(x, y, f(g(x))), S(x, y) \\
(3) & Q(x, y, f(z)) & \rightarrow & R(f(g(x)), x, h(y)) \\
(4) & R(f(g(x)), y, h(z)) & \rightarrow & P(x, y, z) \\
(5) & P(a, b, c) & \rightarrow &
\end{array}$$

where clauses are written in implication form. The clauses (2)-(4) recursively define the predicate P. By resolving clauses (1) and (2) via resolution, the clause

$$\rightarrow Q(f(a), h(a), f(g(f(a)))), S(f(a), h(a))$$

is generated causing an overall depth increase by the term $f(g(f(a)))$, the third argument of Q through the first argument of P. The deeper term is a result of x occurring at depth 0 in $P(x, y, z)$ in clause (2) and at depth 2 in the third argument of $Q(x, y, f(g(x)))$. In this case, we require that the third argument of Q cannot show up by resolving along the cycle (2)-(4) as a first argument of P. We ensure this by the concept of a watched argument (Definition 5.3). The terms at watched arguments of an atom are assumed to never increase during any derivation and argument positions holding terms with increased variable depth only depend on watched argument positions. For the example, the argument positions $1, 2$ of the predicates Q and P are watched and all atoms with predicates P, Q, satisfy this requirement (Definition 5.7-(iii)). A second increase in depth is potentially produced by clause (3), at the first argument of the R atom, where the clauses (2)-(4) also recursively define R. This clause does also not eventually generate terms of increasing depth, because all occurrences of R atoms in the clause set are similar (Definition 2.14), i.e., they have the same tree shape, and thus can only generate a bounded increase in depth (Definition 5.6). Finally, for the clauses (1), (4), (5) the depth of occurrences of variables in positive literals is smaller than their respective depth in negative literals. As a result, positive Hyper-resolution terminates on the above clause set.

In addition to the decidability result, this thesis describes the implementation of the $\mathcal{BDI}$ criteria into the state-of-the-art automated theorem prover SPASS version 3.8 in order to automatically classify whether a given problem satisfies the conditions of the new class $\mathcal{BDI}$. For a positive result, the problem is terminating and decidable. To this end, the version of SPASS including the implementation of the $\mathcal{BDI}$ class properties is applied to the problems of the TPTP Library (Version 6.1.0) [33] in order to detect potentially new decidable problems and to find problems "close" to the $\mathcal{BDI}$ class having only a few clauses which are responsible for the overall non-termination of the problem.

1.3 Related Work

The identification of decidable fragments of first-order logic has a long tradition in automated reasoning research. It started with the specification of decidable quantor prefix classes at the beginning of the 20th century: Bernays-Schönfinkel, Ramsey, Ackermann, Gödel, Kalmár, Schütte (see [8] for an overview). After the invention of automated reasoning calculi, in particular resolution-based calculi, it moved to the identification of decidable clause classes (e.g., see [6, 35, 12, 17, 19, 1]) which then serve, e.g., as (background) fragments for effective reasoning on tree automata properties, reachability problems in security, knowledge representation formalisms, or data structures.

The $\mathcal{BDI}$ class presented in this thesis is not included in any known decidable clause class. It generalizes the well-known class $\mathcal{PVD}$ [12]. The class of guarded formulas, originally proposed by Andréka et al. [27], was shown to be decidable through an effectively bounded finite model property. The first resolution decision procedure for the guarded fragment has been described by de Nivelle [28]. It has been further studied by Georgieva et al. [14] resulting in the fragment GF1$^-$, for which Hyper-resolution is a decision procedure. The class GF1$^-$ includes function symbols but does not support non-guarded formulas. For example, a transitivity clause is not included in this fragment but contained in our class $\mathcal{BDI}$. Further classes that can be decided by resolution (superposition) without generating clauses with a term depth increase are the monadic class [6], the class of shallow sort theories [35], or classes related to tree automata [19].

Another related class is $\mathcal{BU}$ [15], which generalizes the set of all clause sets one can obtain from GF1$^-$, includes function symbols, and is also decidable by Hyper-resolution. The class definition of $\mathcal{BU}$ takes special care of variables, for example, every non-positive functional clause must contain a covering negative literal which contains all the variables of the clause. Eventually this limits the depth of clauses generated by Hyper-resolution. In $\mathcal{BDI}$, we don't require such conditions but instead limit the form of recursive definitions.

A completely different way to deal with the problem of termination is the method proposed by Baumgartner et al. [7]. It reduces model search first to a sequence of satisfiability problems made of function-free first-order clause sets, and then applies a theorem prover to the transformed (decidable) problem. However, this method also comes with several drawbacks. One is that the transformation is not unsatisfiability preserving unless left-totality is forced. Consider the following (unsatisfiable) clause set (in implication form) as an example:

$$\begin{array}{lrcl} (1) & & \rightarrow & P(g(x),a) \\ (2) & P(g(b),y) & \rightarrow & \end{array}$$

Obviously, simple resolution between these two clauses yields the empty clause. The proposed transformation (without the left-totality) leads to the new clause set

$$\begin{array}{lrcl} (1') & R_a(z), R_g(x,y) & \rightarrow & P(y,z) \\ (2') & R_b(z), R_g(z,x), P(x,y) & \rightarrow & \end{array}$$

which is not unsatisfiable anymore. The left-totality for restoring unsatisfiability consists of a skolemized version of the axiom $\forall x_1, \ldots, x_n \exists y R_f(x_1, \ldots, x_n, y)$ for all function symbols f, which causes an expansion of the signature of the problem in size of the number of domain elements. In the context of SAP authorizations with large domains, this would cause a massive increase of the generation of ground facts during saturation. This obvious problem of scalability towards larger domain sizes is known by the authors and has also been mentioned as a main area of further research in their paper. A second main disadvantage is the missing (value) symmetry breaking mechanism for function symbols of arity greater than one. A constraint satisfaction problem exhibits value symmetry if permuting the values of a partial solution for the problem gives another partial solution. For $\mathcal{BDI}$ classes, this is not a problem because there is no transformation applied to a given set of clauses and the termination is guaranteed mainly by limiting the form of recursive definitions.

2 Foundations

2.1 Mathematical Foundations

This chapter recalls basic definitions of first-order logic as well as the fundamentals of first-order theorem proving which are used in this thesis. The definitions are basically taken from [36], and [5], and the relations on terms from [2]. Further, it recaps the definition and properties of graphs.

Definition 2.1 (Natural Numbers)
The set $\mathbb{N}$ of natural numbers contains all non-negative integers, i.e., $\mathbb{N} = \{0, 1, 2, \ldots\}$.

Definition 2.2 (Multisets)
A *multiset* over a set A is a function $M : A \to \mathbb{N}$. Intuitively, $M(a)$ specifies the number of occurrences of a in M. If $M(a) > 0$ then a is an element of M, otherwise M is called *empty*. Like the empty set, the empty multiset is denoted by $\emptyset$. The *union* of two multisets M_1 and M_2 with $x \in A$ is defined by $(M_1 \cup M_2)(x) = M_1(x) + M_2(x)$. Analogously, the *intersection* of M_1 and M_2 with $x \in A$ is defined by $(M_1 \cap M_2)(x) = \min\{M_1(x), M_2(x)\}$.

Multisets are described by using a set-like notation. For example, $\{x, x, x\}$ denotes the multiset M where $M(x) = 3$ and $M(x') = 0$ for all $x' \neq x$ in A.

Definition 2.3 (Orderings)
A *partial ordering* $\succeq$ on a set M is a binary relation on M that is

(i) *reflexive*, i.e. $a \succeq a$ for all $a \in M$,

(ii) *antisymmetric*, i.e., $a \succeq b$ and $b \succeq a$ implies $a = b$ for all $a, b \in M$, and

(iii) *transitive*, i.e. $a \succeq b$ and $b \succeq c$ implies $a \succeq c$ for all $a, b, c \in M$.

A *strict partial ordering* $\succ$ on a set M is a binary relation on M that is

(i) *irreflexive*, i.e., $a \not\succ a$ for all $a \in M$, and

(ii) *transitive*, i.e., $a \succ b$ and $b \succ c$ implies $a \succ c$ for all $a, b, c \in M$.

If $=$ is the identity relation on M and $\succeq$ is a partial ordering on M, then the relation $\succ$ defined as $\succ = (\succeq \setminus =)$ is a strict partial ordering. Conversely, if $\succ$ is a strict partial ordering on M, then the relation $\succeq$ defined as $\succeq = (\succ \cup =)$ is a partial ordering.

A partial ordering $\succeq$ is *total* or an *ordering* on $M' \subseteq M$ if $a \succeq b$ or $b \succeq a$ for all $a, b \in M'$. A strict partial ordering $\succ$ is *total* or a *strict ordering* on $M' \subseteq M$ if $a \succ b$ or $b \succ a$ or $a = b$ for all $a, b \in M'$.

Definition 2.4 (Lexicographic and Multiset Orderings)
Let M, N be sets. The multiset extension $\succ^{mul}$ of a strict partial ordering $\succ$ is defined as follows: $M \succ^{mul} N$ if

(i) $M \neq N$, and

(ii) for all $n \in N \setminus M$ there exists an $m \in M \setminus N$ with $m \succ n$.

The lexicographic extension $\succ^{lex}$ on tuples of length n of some strict order $\succ$ is defined as: $(t_1, \dots, t_n) \succ^{lex} (s_1, \dots, s_n)$ if for some $1 \leq i \leq n$ holds

(i) $t_j = s_j$ for all $1 \leq j < i$, and

(ii) $t_i \succ s_i$.

Definition 2.5 (Minimal and Maximal Elements)
Let $\succ$ be a strict partial ordering on a set M and let N be a subset of M or a multiset over M. With respect to $\succ$ and N, an element $a \in N$ is called

- *maximal* if there is no element $b \in N$ with $b \succ a$,
- *minimal* if there is no element $b \in N$ with $a \succ b$,
- *strictly maximal* if a is maximal and, if N is a multiset, $N(a) = 1$,
- *strictly minimal* if a is minimal and, if N is a multiset, $N(a) = 1$.

Definition 2.6 (Well-Foundedness)
Let M be a set. A (strict) partial ordering $\succ$ is called *well-founded* (Noetherian), if there is no infinite descending chain $a_0 \succ a_1 \succ a_2 \succ \dots$ with $a_i \in M, i \in \mathbb{N}$.

2.2 First-Order Logic

2.2.1 Syntax

This section introduces the syntax of the first-order language without equality used within this thesis, especially terms, clauses, and formulas. We follow the notation of [36] and [29].

Terms and Formulas

Definition 2.7 (Signature)
A first-order language is constructed over a *Signature* $\Sigma = (\mathcal{F}, \mathcal{R})$ with

(i) $\mathcal{F}$ is a non-empty, in general infinite set of *function symbols*, and

(ii) $\mathcal{R}$ is a non-empty, in general infinite set of *predicate symbols*.

Additionally, every function or predicate symbol has some fixed *arity*: $\mathcal{F} \cup \mathcal{R} \to \mathbb{N}$. A function symbol f with *arity*$(f) = 0$ is called a *constant*.

The letters P and Q are typically used as predicate symbols, f, g as function symbols, and u, v, x, y, z as variables.

Definition 2.8 (Terms)
Let $\mathcal{X}$ be an infinite set of variable symbols disjoint from the symbols in the signature Σ. The set of all *terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is recursively defined by:

(i) every function symbol $c \in \mathcal{F}$ with arity zero (constant) is a term,

(ii) every variable $x \in \mathcal{X}$ is a term, and

(iii) whenever $t_1, \ldots, t_n$ are terms and $f \in \mathcal{F}$ is a function symbol with *arity*$(f) = n$, then $f(t_1, \ldots, t_n)$ is also a term.

A term not containing any variable is a *ground term*.

To improve readability, a list $t_1, \ldots, t_n$ of terms is often written as $\vec{t}$ if the counter variable n is obvious from the context or the concrete value of the upper bound is not important. Otherwise, in case we explicitly want to mention that the counter variable runs from 1 to n, we write $\vec{t_n}$. Also, there is the n-fold application $f(\ldots(f(t))\ldots)$ of a unary function symbol f to a term t for which we write in short $f^n(t)$.

Definition 2.9 (Atoms)
Let $\Sigma = (\mathcal{F}, \mathcal{R})$ be a signature, $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, and $P \in \mathcal{R}$ is a predicate symbol with *arity*$(P) = n$. Then, $P(\vec{t})$ is an *atom* over the signature Σ.

Definition 2.10 (Formulas)
Let Σ be a signature. The set of first-order *formulas* is inductively defined in terms of atoms over Σ as follows:

(i) every atom is a formula,

(ii) the two logical constants $\top$ and $\bot$ (true and false) are formulas,

(iii) if ϕ_1, ϕ_2 are formulas, so are $\neg\phi_1$ and $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$, and

(iv) if $x \in \mathcal{X}$ and ϕ is a formula, then $\exists x.\phi$ and $\forall x.\phi$ are formulas.

The formulas occurring in this thesis are written in implication form, $\Gamma \to \Delta$, where Γ contains the negative atoms and Δ the positive atoms. For example, a formula $\forall x.\neg\phi_1 \wedge \phi_2$ is stated as $\forall x.\phi_1 \to \phi_2$.

Definition 2.11 (Literals, Clauses)
An atom A or the negation of an atom $\neg A$ is called *literal*. Disjunctions of literals are *clauses* where all variables are implicitly universally quantified. Clauses are often denoted by their respective multisets of literals where the multisets are written in usual set notation. Alternatively to the multiset notation of clauses, clauses are written in implication form $\Gamma \to \Delta$ where the

multiset Γ is called *antecedent* and the multiset Δ is called *succedent* of the clause. The atoms in Γ denote negative literals while the atoms in Δ denote the positive literals in a clause. A clause is called a *unit clause* if it contains only one literal. The *empty clause* with $\Gamma = \Delta = \emptyset$ is denoted by $\square$. A clause is *Horn* if Δ contains at most one atom. We often abbreviate disjoint set union with sequencing, e.g., we write $\Gamma \rightarrow \Delta, R(t_1, \dots, t_n)$ for $\Gamma \rightarrow \Delta \backslash \{R(t_1, \dots, t_n)\} \cup \{R(t_1, \dots, t_n)\}$.

Definition 2.12 (Variables of Terms and Atoms)
The set of *free* variables of an atom P, term f is denoted by

(i) $\mathit{vars}(P(t_1, \dots, t_n)) = \cup_i \mathit{vars}(t_i)$,

(ii) $\mathit{vars}(f(t_1, \dots, t_n)) = \cup_i \mathit{vars}(t_i)$, and

(iii) $\mathit{vars}(x) = \{x\}$.

The function naturally extends to literals, clauses and (multi)sets of terms (literals, clauses).

Definition 2.13 ((Inner) Positions, Length)
A *position* is a word over the natural numbers. Let $f(t_1, \dots, t_n)$ be a term. The set $\mathit{pos}(f(t_1, \dots, t_n))$ of positions of a term is recursively defined as:

(i) the empty word ϵ is a position in any term t and $t|_\epsilon = t$

(ii) if $t|_p = f(t_1, \dots, t_n)$, then $p.i$ is a position in t for all $i = 1, \dots, n$ and $t|_{p.i} = t_i$.

An alternative notation for $t|_p = s$ is $t[s]_p$. The term (atom) obtained from t by *replacing* $t|_p$ in t with s is denoted by $t[p/s]$ where $p \in \mathit{pos}(t)$.

The *length* of a position p is defined by $\mathit{length}(\epsilon) = 0$ and $\mathit{length}(i.r) = 1 + \mathit{length}(r)$. The notion of a position can be extended to atoms, literals and even formulas in the obvious way.

Let p be an arbitrary position of a term s (atom, literal). We call p an *inner position* if there exists a position q in s such that $q = p.r$, $r \neq \epsilon$.

Definition 2.14 (Similarity)
Let p be an arbitrary position of a term s (respectively, atom or literal). Two atoms $P(t_1, \dots, t_n)$ and $Q(s_1, \dots, s_m)$ are called *similar* if $\mathit{pos}(P(t_1, \dots, t_n)) = \mathit{pos}(Q(s_1, \dots, s_m))$ and for all inner positions p we have $P(t_1, \dots, t_n)|_p = Q(s_1, \dots, s_m)|_p$, implying $P = Q$ and $n = m$.

Definition 2.15 (Depth, Occurrences)
The *depth* of a term is the maximal length of a position in the term, for example, $\mathit{depth}(t) = \mathit{max}(\{\mathit{length}(\pi) \mid \pi \in \mathit{pos}(t)\})$. The depth of a literal $[\neg]P(t_1, \dots, t_n)$ is the maximal depth of its terms: $\mathit{depth}([\neg]P(t_1, \dots, t_n)) = \mathit{max}(\{\mathit{depth}(t_1), \dots, \mathit{depth}(t_n)\})$. The depth of a clause is the maximal depth of its literals, and in the same manner, the depth of a set of literals is the maximal depth of its literals. Additionally, the function *depth* is extended to

variables and clauses (or sequences of literals) such that $\mathit{depth}(x, C)$ returns the maximal depth of an occurrence of the variable $x \in \mathit{vars}(C)$ of a clause C.

Let *occ* be a function returning the number of occurrences of a term s in a term t, defined by

$$\mathit{occ}(s, t) = |\{p \in \mathit{pos}(t) \mid t|_p = s\}|.$$

Substitutions

Definition 2.16 (Substitutions)
A *substitution* σ is a mapping from the set of variables $\mathcal{X}$ to the set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $x\sigma \neq x$ for only finitely many $x \in \mathcal{X}$. The *domain* of a substitution σ is defined as $\mathit{dom}(\sigma) = \{x \mid x\sigma \neq x\}$ and the co-domain of σ as $\mathit{cdom}(\sigma) = \{x\sigma \mid x\sigma \neq x\}$. A *ground substitution* σ has no variable occurrences in its co-domain, i.e., $\mathit{vars}(\mathit{cdom}(\sigma)) = \emptyset$. An injective substitution σ where $\mathit{cdom}(\sigma) \subset \mathcal{X}$ is called a *variable renaming*.
A substitution σ can be lifted to terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ by $f(t_1, \ldots, t_n) = f(t_1\sigma, \ldots, t_n\sigma)$ for all $f \in \mathcal{F}$ with arity n. Likewise, the application of σ to literals and clauses is defined as:

(i) $P(t_1, \ldots, t_n)\sigma = P(t_1\sigma, \ldots, t_n\sigma)$
(ii) $(\neg P(t_1, \ldots, t_n))\sigma = \neg P(t_1\sigma, \ldots, t_n\sigma)$
(iii) $\{A_1, \ldots, A_n\}\sigma = \{A_1\sigma, \ldots, A_n\sigma\}$
(iv) $(\Gamma \to \Delta)\sigma = \Gamma\sigma \to \Delta\sigma$.

Definition 2.17 (Unifiers and Most General Unifiers)
Given two terms (atoms) s, t, a substitution σ is called a *unifier* for s and t if $s\sigma = t\sigma$. It is called a *most general unifier* (mgu) if for any other unifier τ of s, t there exists a substitution λ with $\sigma\lambda = \tau$. A substitution σ is called a *matcher* from s to t if $s\sigma = t$. The notion of an mgu is extended to atoms and literals in the obvious way. We say that σ is a unifier for a sequence of terms (atoms, literals) $t_1, \ldots, t_n$ if $t_i\sigma = t_j\sigma$ for all $1 \leq i, j \leq n$ and σ is an mgu if in addition for any other unifier τ of $t_1, \ldots, t_n$, there exists a substitution λ with $\sigma\lambda = \tau$.

Term and Clause Orderings

Superposition-based calculi typically limit the number of possible inferences by considering only clauses where the involved literals are maximal. This is admissible because the eventual proof of the input clauses still remains correct (model existence problem, [3]), and is realized by using ordering conditions that are stated on the term level, and which are then extended to literal occurrences in a clause, and to clauses (see Definition 2.20).

Popular orderings are the Knuth-Bendix ordering (KBO) [22, 10], the lexicographic path ordering (LPO) [21], and the recursive path ordering with status (RPOS) [10]. In this thesis (and within our usage of the prover Spass), we use the KBO. For a broader introduction to orderings, consider the book by Baader and Nipkow [2].

Definition 2.18 (Precedence, Weight)
Let $\Sigma = (\mathcal{F}, \mathcal{R})$ be a finite signature. The strict partial ordering $>$ on the symbols in Σ is called a *precedence*. Let $\mathit{weight} : \Sigma \to \mathbb{N}$ be a weight function. We call a weight function *admissible* for some precedence if for every unary function symbol f with $\mathit{weight}(f) = 0$, the function f is maximal in the precedence, i.e., $f \geq g$ for all other function symbols g.

The function *weight* is extended to a weight function on terms $\mathit{weight} : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \mathbb{N}$ as follows:

(i) if $t \in \mathcal{X}$ then $\mathit{weight}(t) = k$, where $k = \mathit{min}(\{\mathit{weight}(c) \mid c \in \mathcal{F}, \mathit{arity}(c) = 0\})$, and

(ii) $t = f(t_1, \ldots, t_n)$, then $\mathit{weight}(t) = \mathit{weight}(f) + \sum_i \mathit{weight}(t_i)$.

Definition 2.19 (Kuth-Bendix Ordering)
Let $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ be two terms, then $t \succ_{kbo} s$ if $\mathit{occ}(x, t) \geq \mathit{occ}(x, s)$ for every variable $x \in (\mathit{vars}(t) \cup \mathit{vars}(s))$ and

(i) $\mathit{weight}(t) > \mathit{weight}(s)$, or

(ii) $\mathit{weight}(t) = \mathit{weight}(s)$ and $t = f(t_1, \ldots, t_n)$ and $s = g(s_1, \ldots, s_m)$ and

  (2a) $f > g$ in the precedence, or

  (2b) $f = g$ and

    (2b1) $(t_1, \ldots, t_n) \succ_{kbo}^{lex} (s_1, \ldots, s_m)$ or

    (2b2) $(t_n, t_{n-1}, \ldots, t_1) \succ_{kbo}^{lex} (s_1, s_{m-1}, \ldots, s_1)$

If the precedence $>$ is total on Σ then the KBO is total on ground terms (atoms).

Definition 2.20 (Literal/Clause ordering)
Let $\succ$ be a (reduction) ordering on terms/atoms. It can be extended to clauses as follows: Clauses are considered as multisets of occurrences of atoms. The occurrence of an atom A in the antecedent of a clause is identified with the multiset $\{\{A, \top\}\}$, the occurrence of A in the succedent with $\{\{A\}, \{\top\}\}$. The constant $\top$ is always assumed to be minimal with respect to $\succ$. Additionally, the symbol $\succ$ is overloaded on literal occurrences to be the twofold multiset extension of $\succ$ on terms (atoms) and $\succ$ on clauses to be the multiset extension of $\succ$ on literal occurrences. If $\succ$ is well-founded (total) on terms (atoms), so are the multiset extensions on literals and clauses.

Definition 2.21 (Reductive Clause)
A clause $\Gamma \to \Delta, A$ is called *reductive* for the atom A if the atom A is a strictly maximal occurrence of an atom in the clause.

2.2.2 Semantics

Definition 2.22 (Herbrand Interpretation)
A *Herbrand interpretation* $\mathcal{I}$ is a set of ground atoms. A ground atom A is said to be true in $\mathcal{I}$ if $A \in \mathcal{I}$. It is called false in $\mathcal{I}$ if $A \notin \mathcal{I}$. The constant $\top$ is true in all interpretations, whereas $\bot$ is false in all interpretations. The logical connectives are interpreted as usual: A conjunction $A \wedge B$ is true in $\mathcal{I}$ if both A and B are true in $\mathcal{I}$; a disjunction $A \vee B$ is true if at least one of A and B is true; a negated atom $\neg A$ is true in $\mathcal{I}$ if $A \notin \mathcal{I}$. If F is a formula, the universally quantified formula $\forall x.F$ is true in $\mathcal{I}$ if $F\sigma$ is true in $\mathcal{I}$ for all substitutions σ that assign x to some ground term, and an existentially quantified formula $\exists x.F$ is true in $\mathcal{I}$ if $F\sigma$ is true in $\mathcal{I}$ for some substitution σ that assigns x to some ground term. A ground clause is true in $\mathcal{I}$ if at least one of its literals is true in $\mathcal{I}$.

Definition 2.23 (Model)
An interpretation $\mathcal{I}$ is called a *model* of a formula F if F is true in $\mathcal{I}$ and is written as $\mathcal{I} \models F$. Similarly, for two formulas F_1, and F_2, $F_1 \models F_2$ denotes that whenever F_1 is true in $\mathcal{I}$ then also F_2 is true in $\mathcal{I}$. An alternative notation is $\mathcal{I} \models F_2$ whenever $\mathcal{I} \models F_1$. If $C_1, \dots, C_n$ and D are clauses, we write $C_1, \dots, C_n \models D$ if for all Herbrand interpretations $\mathcal{I}$ whenever $\mathcal{I} \models C_1, \dots, C_n$ then also $\mathcal{I} \models D$. For a set of clauses N, we say $\mathcal{I}$ is a model of N, written $\mathcal{I} \models N$ if $\mathcal{I} \models C$ for all clauses $C \in N$.

Definition 2.24 (Satisfiability)
Let N be a clause set. N is called *satisfiable* if there is a Herbrand interpretation $\mathcal{I}$ with $\mathcal{I} \models N$. Otherwise, N is called *unsatisfiable*.

2.3 First-Order Reasoning

First-order theorem proving deals with the problem of showing whether a given clause (formula) C is a logical consequence of a set of clauses N, written as $N \models C$. The proof is usually done the other way around by showing the inconsistency of the set $N \cup \{\neg C\}$. This inconsistency is established by providing a formal proof of $\bot$ from N by means of appropriate calculi which are described by a collection of inference and reduction rules that determine how new clauses can be derived from the given clauses.

Let $Res^{\succ}_{S}$ be the calculus – the inference and reduction rules – used in this thesis which is presented in detail in the following sections. It uses the resolution calculus of Bachmair and Ganzinger [5] that is based on general resolution due to Robinson [31]. The rules are given in a generic way such that each definition covers several variants of the rule. In particular, the rules are parameterized by an admissible ordering $\succ$ on literals and a selection function S. The admissible ordering is a total (well-founded) strict ordering on the ground level for the literals and extended to the non-ground level in a canonical manner. The selection function assigns to each clause a possibly empty set of occurrences of negative literals with the effect that all inference rule applications taking this clause as a parent clause must involve the selected literals [36]. Note that the selection restriction of negative literals does not destroy completeness [4].

2.3.1 Inferences

Given the calculus $\mathit{Res}_S^{\succ}$, one way of altering a given clause set N is to use inferences with premises in N to derive new clauses. The other way is to eliminate so-called *redundant* clauses which is described in Section 2.3.2.

An *inference* is a relation on clauses where the elements of such a relation are written as

$$\frac{C_1 = \Gamma_1 \to \Delta_1 \quad \dots \quad C_n = \Gamma_n \to \Delta_n}{D = \Gamma \to \Delta}$$

The clauses $C_1, \dots, C_n$ are called the *premises*, and D the *conclusion* or *resolvent* of the inference. The conclusion of the inference is eventually added to the given clause set N. A set of inferences is called an inference system.

The closure of clause sets under the calculus $\mathit{Res}_S^{\succ}$ in this thesis is given by

$$\begin{aligned}
\mathit{Res}(N) &= \{C \mid C \text{ is a conclusion of an inference or} \\
&\qquad\qquad \text{reduction rule with premises in } N\}, \\
\mathit{Res}^0(N) &= N, \\
\mathit{Res}^{n+1}(N) &= \mathit{Res}(\mathit{Res}^n(N)) \cup \mathit{Res}^n(N) \text{ for } n \geq 0, \text{ and} \\
\mathit{Res}^*(N) &= \bigcup_{n \geq 0} \mathit{Res}^n(N).
\end{aligned}$$

A set of clauses N is called *saturated* (with respect to the inferences of the calculus *Res*) if $\mathit{Res}^*(N) \subseteq N$. A set N of clauses that has been altered by n steps of inferences (and reductions) is written as $\mathit{Res}^n(N)$. If the exact number n is not known or not important, the altered set N is referred to as N^*.

This thesis makes use of the following inference rules. The termination proof of the $\mathcal{BDI}$ clause class even computes inferences only using the below mentioned ordered hyper-resolution rule. As usual the calculus *Res* is based on a reduction ordering $\succ$ that is total on ground terms.

Definition 2.25 ((Ordered) Hyper-Resolution)
The inference

$$\frac{E_1, \dots, E_n \to \Delta \quad \to \Delta_i, E_i' \quad (1 \leq i \leq n)}{(\to \Delta, \Delta_1, \dots, \Delta_n)\sigma}$$

where

(i) σ is the simultaneous *mgu* of $E_1, \dots, E_n, E'_1, \dots, E'_n$,

(ii) all $E'_i\sigma$ are strictly maximal in $(\rightarrow \Delta_i, E'_i)\sigma$

is called an *ordered hyper-resolution* inference. If condition (ii) is dropped, the inference is called a *hyper-resolution* inference.

Definition 2.26 ((Ordered) Resolution)
The inference

$$\frac{\Gamma_1 \rightarrow \Delta_1, E_1 \quad E_2, \Gamma_2 \rightarrow \Delta_2}{(\Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2)\sigma}$$

where

(i) σ is the *mgu* of E_1 and E_2,

(ii) no literal in Γ_1 is selected,

(iii) $E_1\sigma$ is strictly maximal in $(\Gamma_1 \rightarrow \Delta_1, E_1)\sigma$,

(iv) the atom $E_2\sigma$ is selected or it is maximal in $(E_2, \Gamma_2 \rightarrow \Delta_2)\sigma$, and no literal in Γ_2 is selected

is called an *ordered resolution* inference. If conditions (iii)-(iv) are replaced by E_2 is selected or no literal is selected in Γ_2, the inference is called *resolution*.

Definition 2.27 ((Ordered) Factoring)
The inferences

$$\frac{\Gamma \rightarrow \Delta, E_1, E_2}{(\Gamma \rightarrow \Delta, E_1)\sigma}$$

and

$$\frac{\Gamma, E_1, E_2 \rightarrow \Delta}{(\Gamma, E_1 \rightarrow \Delta)\sigma}$$

where

(i) σ is the *mgu* of E_1 and E_2,

(ii) (E_1, E_2 occur positively, E_1 is maximal and no literal in Γ is selected), or (E_1, E_2 occur negatively, E_1 is maximal and no literal in Γ is selected or E_1 is selected),

is called *ordered factoring right* and *ordered factoring left*, respectively. If condition (ii) is replaced by (E_1, E_2 occur positively and no literal in Γ is selected) or (E_1, E_2 occur negatively, E_1 is selected or no literal in Γ is selected) the inferences are called *factoring right* and *factoring left*, respectively.

2.3.2 Reductions

Along with using inferences, there is the way of using reductions on a given clause set N. If clauses in a set N^* are already implied by smaller clauses in N^*, they can be eliminated using the reduction rules given below. The concept of redundancy is used to reduce the number of clauses that need to be considered as premises for future inference steps. In general, a clause is called *redundant* with respect to a set of clauses N, if it follows from clauses in N that are smaller than C.

Definition 2.28 (Subsumption)
The reduction
$$\frac{\Gamma_1 \rightarrow \Delta_1}{\Gamma_2 \rightarrow \Delta_2}$$
where $\Gamma_2\sigma \subseteq \Gamma_1$ and $\Delta_2\sigma \subseteq \Delta_1$ for some substitution σ is called *subsumption*.

Definition 2.29 (Condensation)
The reduction
$$\frac{\Gamma_1 \rightarrow \Delta_1}{\Gamma_2 \rightarrow \Delta_2}$$
where $\Gamma_2 \rightarrow \Delta_2$ subsumes $\Gamma_1 \rightarrow \Delta_1$ and $\Gamma_2 \rightarrow \Delta_2$ is derived from $\Gamma_1 \rightarrow \Delta_1$ by instantiation and (exhaustive) application of trivial (duplicate) literal elimination is called *condensation*.

For the purpose of this thesis the reduction rules subsumption and condensation suffice. Nevertheless, the calculus could be enhanced by all simplification rules which are compatible with the general notion of redundancy.

Now the (ordered) hyper-resolution calculus consists of the rules (ordered) hyper-resolution, (ordered) factoring, subsumption deletion, and condensation. The (ordered) resolution calculus consists of the rules (ordered) resolution, (ordered) factoring, subsumption deletion, and condensation.

Reduction rules are applied exhaustively and before the application of any inference rule.

2.3.3 Soundness and Completeness

The most important properties of every calculus are soundness and completeness. Soundness means that only inferences can be made that do not change the semantics of the problem while completeness entails that if a resolvent is a logical consequence then it can also be derived by the calculus.

The calculus $Res_S^{\succ}$ used in this thesis by Bachmair and Ganzinger is sound and refutationally complete [5]. The following gives the formal definition:

Definition 2.30
The calculus $Res_S^{\succ}$ is *sound* and *refutationally complete* with respect to a set N of clauses if $N \models \bot \Leftrightarrow \bot \in N$.

2.4 Graphs

A graph is a representation of a set of objects (vertices), where some of them have relations with other objects. The relations are represented by graphical links (edges). The following gives a formal definition of a graph, and is based on [11, 34].

Definition 2.31
A *graph* G consists of a set of *vertices* V and a set of *edges* E. If each edge in E is an ordered pair (v, w) of vertices in V, the graph is *directed*, or, if each edge in E is an unordered pair $\{v, w\}$ of vertices in V, the graph is *undirected*.

In a directed graph, the element v is the *tail* and the element w is the *head* of an edge (v, w) of vertices in V. Alternatively to the notation (v, w), an edge from v to w is denoted by $v \to w$. A *path* $p : v \to^* w$ in G is a sequence of vertices and edges going from v to w. A path $p : v \to *v$ is called *closed*. The *length* of a path corresponds to the number of edges that the path has. A *cycle* is a closed path with length equal or greater than three.

Figure 2.1 depicts a directed graph G with $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $E = \{(v_1, v_5), (v_1, v_3), (v_2, v_5), (v_3, v_4), (v_4, v_6), (v_5, v_4), (v_6, v_3)\}$.

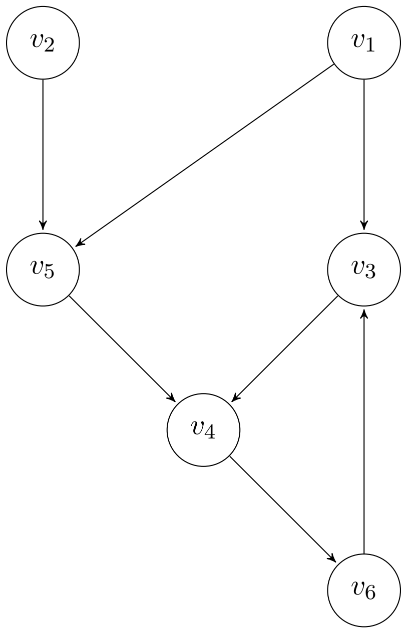


Figure 2.1: Example of a directed graph $G = (V, E)$ with a cycle of length 3 $(v_3 \to v_4 \to v_6 \to v_3)$.

Graphs are used in this thesis to represent the dependencies between clauses and its literals (or predicate symbols) of a given clause set. In order to speak about recursive definitions of predicates or, alternatively, cycles between clauses for a given clause set, the following notion of reachability between predicate symbols of atoms occurring in (possibly different) clauses is established by means of graphs.

Definition 2.32 (Reachability)
Given a clause set N and its directed graph G with $V = \mathcal{R}$ (all predicate symbols) and edges $E = \{(P, Q) \mid C = \Gamma, P(\vec{x}) \rightarrow \Delta, Q(\vec{y})\})$ for all clauses $C \in N$ with corresponding predicate symbols P, Q.

A predicate Q *is reachable from* P *in one step* if it exists an edge (P, Q) in G. Moreover, a predicate R *is reachable from* P if $(P, R) \in E$ or if it exists a path $P \rightarrow^* R$.

3 The SAP (Authorization) System

During the last decades a lot of mid-size and large companies introduced ERP (Enterprise Resource Planning) software like the SAP system.

The SAP systems have a long tradition. It started with a financial Accounting system named R/1® which was then replaced by the SAP R/2® system at the end of the 1970s. The SAP R/2® system was in a mainframe based business application software suite that was very successful in the 1980s and early 1990s. In 1992, with the introduction of distributed client-server computing, SAP brought out a client-server based version, called SAP R/3®. It's new architecture was compatible with multiple platforms and operating systems and allowed SAP to gain a lot of new customers. The current version SAP ERP 6.0 (formerly mySAP ERP 2005) differs from R/3® in the way that it builds on SAP NetWeaver^{TM} as its platform for applications. The components based on SAP NetWeaver^{TM} can be implemented both in ABAP® (SAP's own programming language) or Java. The main component of the SAP ERP 6.0 system – ERP Central Component (SAP ECC) – is the successor of the R/3® system and consists of the following core modules:

- Financials:
  Financial Accounting (FI), Controlling (CO), Strategic Enterprise Management (SEM), Enterprise Controlling (EC), Investment Management (IM), Public Sector Management (PSM), Project System (PS), Real Estate Management (RE), Treasury (TR)

- Human Capital Management (HCM), also known as Human Resources (HR):
  Personnel Management (PA), Personnel Time Management (PT), Payroll (PY), Training and Event Management (PE), Personnel Development (PD), Cost Planning (CP)

- Logistics:
  Materials Management (MM), Production Planning and Control (PP), Plant Maintenance (PM), Sales and Distribution (SD), Logistics Execution (LE), Environment, Health & Safety (EHS), Customer Service (CS), Quality Management (QM), Logistics-General (LO), Product Lifecycle Management (PLM), Warehouse Management (WM)

- Workflow (WF)

Additionally, there are modules which are prefixed with IS (Industry Solution) indicating that they combine the basic modules with additional industry-specific functionality. The following list gives an excerpt of the variety of different industry modules:

- Aerospace and Defense (IS-AD),
- Automotive (IS-A),
- Oil and Gas (IS-OIL),
- Health care (IS-H),
- Media (IS-M)
- ...

Altogether, one can see from the variety of available modules and the long history of SAP that it is a very successful system and the instances itself may be very large. From the organizational point of view, a single instance of an SAP ERP system can map one department of a company (e.g., supply chain management) as well as worldwide corporations with a lot of departments and subsidiaries. Considering such large systems, it is conceivable that especially the administration of authorizations is highly extensive.

Concerning the different versions of SAP, I have already explored in my Masters Thesis [23] that the authorization concepts of the former SAP R/3® system and the current releases seem to be identical. Therefore, we can simply refer to "The SAP System" in this thesis which means the SAP R/3® system or any release after up to now.

During my work, I had access to the SAP installation of the Max-Planck Society which I used to acquire information about the authorization mechanisms and the authorizations. The Max-Planck Society uses the SAP system release *SAP ECC 6.0* with *SAP NetWeaver 7.0 (2004s)*.

This chapter provides the description of the SAP system and the relevant authorization layer that is required for the development of the formalization in Chapter 4. Several paragraphs throughout the whole chapter are adopted literally or in a rephrased form from my Masters Thesis [23]. Besides, Chapter 3 is organized as follows.

In Section 3.1, the reader gets introduced into the different components related to authorization checks and the relationship between these elements in an authorization check procedure will be explained.

Afterwards in Section 3.2, the authorization setup structure representing the users and their authorizations is described and how the authorizations can be assigned to users.

Section 3.3 presents (parts of) the purchase process as a typical example of a business process. It includes some screenshots in order to depict and to ease the understanding of the process implementation in the SAP system.

Eventually, Section 3.4 introduces into the concept of business policies and gives some examples related to the purchase process.

3.1 Authorization Checks

This section explains the main components used for the authorization checks starting with a brief description of the term of a transaction in Section 3.1.1. Transactions often appear in the context of database transactions, and this basic understanding also applies roughly to a transaction in the SAP system.

Sections 3.1.2 and 3.1.3 describe how authorization objects and authorizations are used to protect data, functions and even database tables from unauthorized access, and also dissociate the two terms from each other.

Eventually, a complete authorization check is sketched in Section 3.1.4 which describes how its execution makes use of the single components presented before.

3.1.1 Transactions

Transactions are known from databases where a transaction corresponds to a single logical operation on the data – no matter how many individual changes are required. It must fulfill the so-called ACID criteria: Atomicity, consistency, isolation, and durability.

This concept also applies to transactions in the SAP system. The program (code) where the authorization checks are attached and implemented is called an *SAP transaction*. Usually, the start of a transaction implies the check of multiple individual authorizations. According to the ACID criteria for database transactions, the check of all individual authorizations for a specific user either succeeds or fails as a group (atomicity) and cannot be interfered by authorization checks for other users running the same program (isolation). In other words, if only one individual check fails for the specific user, then the overall transaction fails, too. The exact procedure of authorization checks is described in more detail in Section 3.1.4.

The execution of the transaction in SAP corresponds to the execution of a function in the SAP system. Then, the function starts the associated program. Every program that needs to be protected has its unique identification code which is called *transaction code*. In SAP, the term transaction is typically used to refer the called program itself. Short statements like "the transaction *xy*" denote the program where *xy* is the transaction code for that program.

The concrete number of individual authorization checks often differs between multiple runs of a transaction. This is caused by those checks implemented using a traditional if-else statement and which depend on the data that has been entered.

Furthermore, we need to distinguish between the authorization check itself and the checked data: If the implementation of a number of authorization checks for a concrete transaction doesn't change and the number of checks is also the same for two runs, then the checks themselves are identical (the same access right is checked) in each run of the transaction, but the data is not necessarily equal. For example, a check in a transaction could be the verification of the user's department where the checked data is different because not all employees in a company belong to the same department. In that case, the check for the access right itself – namely the right to access the department – for each run is equal.

3.1.2 Authorization Objects

Authorization objects are used to protect functions or data in the SAP system. Every authorization check occurring when a transaction starts or during the execution of a transaction checks the associated authorization object to be present with the required values. The authorization object can be seen as a container for the authorization values. It is a logical entity that groups between one and ten value fields requiring authority checking within the system. The fields can be eventually filled with different authorization values. This structure is illustrated by Figure 3.1. Already the (old) release 4.6 of

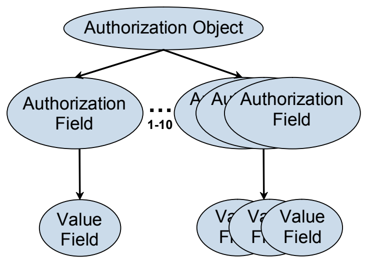


Figure 3.1: Authorization object structure.

SAP includes around 900 pre-defined authorization objects that are classified into ca. 40 object classes corresponding to their application area, for example Finance or Human Resources. If still none of these objects is suitable, there is the possibility to define additional authorization objects which are liable to the structure described before.

3.1.3 Authorizations

The combination of the authorization object with concrete values having been filled into the value fields constitutes the authorization, that is an instance of the authorization object. In this way, many different instances can be created

Authorization Field	**Possible Values**
`ACTVT` (Activity)	`01` - Create
	`02` - Change
	`03` - View

Table 3.1: Authorization values denoting the type of an activity.

using one authorization object by filling in different values. The structure is depicted in Figure 3.2. The term "authorization" often leads to confusion be-

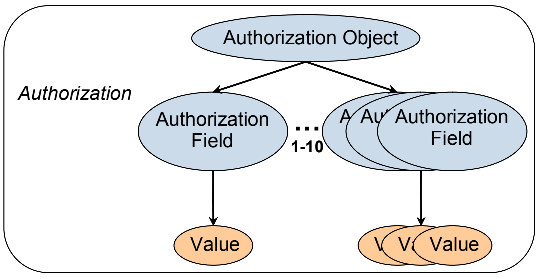


Figure 3.2: Authorization (instance of an authorization object).

tween the everyday language terminology and the terminology used by SAP people. In general, an authorization is the permission of a person to do something, or, translated to our scenario, the permission to execute a function in a system. However, in the SAP system and usually in this thesis (but still depending on the context), we refer to the authorization as the special notion for the previously mentioned combination of the authorization object and its field values.

The insertable values for a value field of an authorization object can be single values or also value ranges. For example, there are a number of numerical values available denoting the type of an activity. Table 3.1 shows some typical activity codes.

In addition to the previously mentioned numerical values and value ranges, one can use the wild-card characters question mark (?) and the asterisk (*). As usual, the question mark represents a single character while the asterisk stands for any combination of characters of any length. As soon as a value contains wild-card characters, it is called a *regular pattern*. However, in this thesis, I have considered only the asterisk as the wild-card symbol and so far no composed values (in order to ease the complexity).

Authorizations are eventually used during the authorization check in the SAP system in which the built-in authorization policy forbids all actions not explicitly permitted by an authorization. Authorizations can only grant permissions but not restrict or forbid them.

3.1.4 Authorization Check Procedure

Authorization checks occur whenever a user requests access to a particular transaction. In such a case, the user's credentials (his/her assigned authorizations) are checked against the requested ones and if the authorizations match (both the authorization object and its values must match), the user is permitted to access the information which is protected by the authorization object.

The actual authorization check consists of two parts: (i) it checks the presence of the required authorization in the authorization profile of the user's master record (for this purpose only the authorization object name is compared), (ii) the comparison of the required value(s) with the value(s) present in the value field(s) of the authorization. The check (i) is successful if the requested authorization object is available in the user's profile. The second check (ii) succeeds if all value fields with the corresponding values of the object match to the required fields and values (AND-combination). Matching means in the context of this thesis that one of the following conditions holds:

- The present value in the value field is the asterisk wild-card character (*) . This character matches any required value.
- The present value in the value field and the required value match exactly, i.e., they are equal.

As mentioned earlier, regular pattern authorization values as well as intervals are not considered in this thesis.

Further, a successful authorization check requires both the authorization object existence check (i) and all single field/value checks (ii) to succeed. If only one of the single checks fails, then the overall check of the authorization fails.

There are two different cases when an authorization check is triggered during the execution of a transaction:

- **Check at the start of a transaction**
  There is a special authorization object, named `S_TCODE`, that is always checked at first for every transaction before the actual program associated with the transaction will be executed. This verification is set up by the system and can't be turned off. It is possible to associate one further authorization object at this place and let the system check these two objects before the actual program starts.
- **Check during the progress of a program**
  This is the default case where all authorization checks are located in

the program code. On the code level, an authorization check can be forced using the *AUTHORITY-CHECK* statement. Whenever the system reads this keyword during the execution of the program code, it will check the associated authorization object with its fields[1]. It is possible to check one object repeatedly in a program or to group several different authorization objects to protect special data or a specific part of the program.

In earlier times, the authorization objects always had to be identified manually in the program code. Today this is a bit easier because SAP provides database tables[2] listing all these checks. Of course, if a customer doesn't follow this convention and decides not to insert the checks into these tables, it is necessary to locate the checks directly in the program code as before.

3.2 The Authorization Setup

The authorization setup is the layer on which authorizations are assigned to users. However, authorizations assigned to users become only effective for a user and can be used in an authorization check if the authorizations are also present in the authorization profile of the user's master record.

Authorizations can be assigned to users using the following different ways: The administrator creates so-called roles or authorization profiles or uses a combination of both ways. As mentioned, the effective authorizations are always and only stored in authorization profiles. The assignment of any authorization to a user on the base of a role therefore requires the creation of the corresponding authorization profile. All parts are shown in an overview picture in Figure 3.3.

The following sub-sections describe the definition of authorization profiles and roles and also explain the differences between these two terms.

3.2.1 Authorization Profiles

An authorization profile is a group of authorizations. If an authorization profile has been assigned to a user then he/she is authorized to access all transactions/functions/data granted by the authorizations in the profile.

SAP distinguishes between simple (or single-level) profiles and composite profiles.

[1]It is possible to disable the check of an object but this is not important to the general functionality and therefore discarded here.

[2]The table `USOBX` holds the default values for authorizations checks occuring in transactions. The customer table `USOBX_C` is initially (during the overall system setup) filled with the contents of the table `USOBX` and can be adjusted further to fit the customer's personal needs.

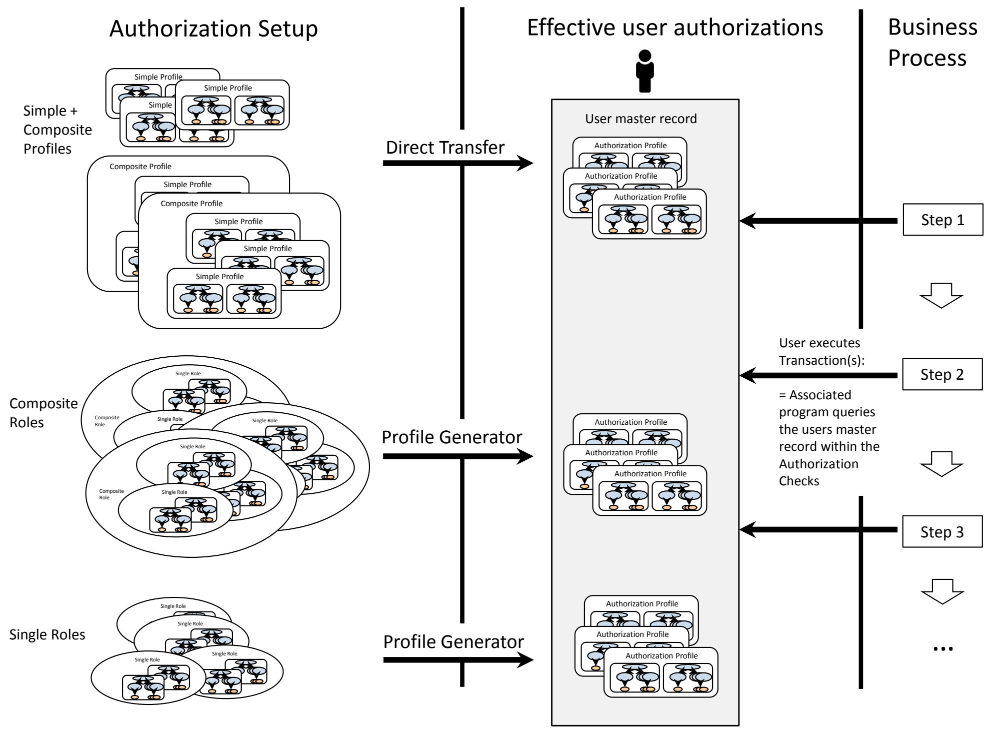


Figure 3.3: Complete picture of SAP authorizations.

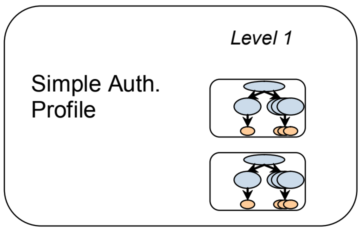


Figure 3.4: Structure of a simple (single-level) authorization profile.

- Simple (single-level) profiles have authorizations on one level. Nested levels are not possible. The following Figure 3.4 shows the structure of a simple profile. The assignment of a simple profile to a user results in the assignment of all authorizations to the user which are contained in the profile.

- Composite profiles can contain single profiles or other composite profiles. The notion is to reduce the maintenance effort with a better structure in the authorizations part. Therefore, a composite profile groups different simple and/or other composite profiles together. The following Figure 3.5 depicts the structure of a composite profile.

  From the technical side there is no difference between the assignment of a simple profile and a composite profile to a user. The assignment of a composite profile just results in assignments of all authorizations which are contained at any level (union). Authorizations can only grant access to transactions/functions/data but they can't forbid the access. Therefore, conflicts resulting from the union cannot occur because the union of access grants implies again access grants and never the denial to access something. If, for example one profile contains only the permission to read some data and another profile contains the permission to write then the result is the permission to read *and* write the data.

  The existence of the composite profiles remains from earlier releases of SAP R/3® without the support for roles. SAP strongly recommends to only use the concept of roles.[32, p. 88] The main reason concerns the ability to structure. Roles offer more structure features than profiles and are the newer concept.

3.2.2 Roles

The so called Role Based Access Control (RBAC – also called role-based security) is an essential concept in SAP systems. Role-based security is a form of user-level security where the application doesn't focus on the individual user's

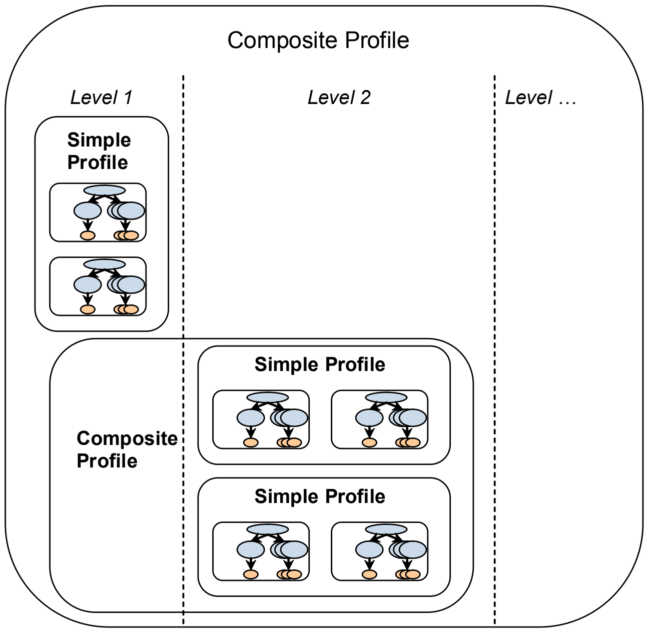


Figure 3.5: Structure of a composite authorization profile.

identity; but rather on a logical role they occupy.

The concept of roles is important in SAP systems because roles offer great possibilities to compose structures. It is possible to create single roles as well as composite roles and even inheritance between roles is supported. A tool called "SAP Profile Generator" is used to create single and composite roles. The name Profile Generator comes from the fact that an authorization profile will be generated for each role and also after every change of the role. The generation is required because authorizations are only effective for a user if he/she holds them in his/her master record. If a user holds authorizations then he/she in fact holds one or more authorization profiles which group the authorizations.

If a role inherits properties from another role then the parent role is called *template role*. For example, a template role can inherit the contained authorizations to the child roles. Just like inheritance in programming languages, changes to the template role will be automatically applied to the derived roles. Template roles are often used to define roles having the same functionality (i.e., allowing to execute the same transactions), but the specific different organizational level authorizations such as company code, purchasing organization, etc. are then defined in the child roles.

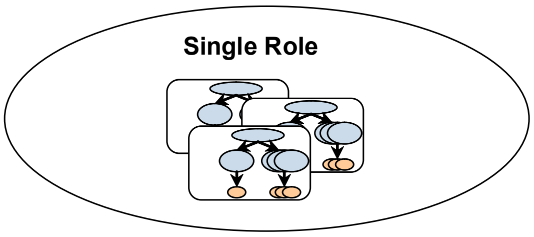


Figure 3.6: Structure of a single role.

Single Roles

The structure of a single role is similar to the authorization profile, i.e., it also contains authorization values. Figure 3.6 depicts the structure of a single role.

Whenever a single role is created using the Profile Generator the appropriate authorization profile should be generated, too. A user effectively holds only the authorizations which are present in the (generated) authorization profile and which in turn has been assigned to the user. Therefore, the assignment of a role to a user without the existence of the generated authorization profile is useless. This is due to the fact that authorization checks always and only compare the required authorizations with the authorizations present in the authorization profile of the user's master record. Therefore, an authorization check will fail if the required authorization is only present in the role which has been assigned to the user and not in the user's authorization profile.

The difference to the (direct) authorization profile is that changes to the authorizations of a user which are grouped in auto-generated authorization profiles must take place through the role definition. It isn't possible to change values directly in (auto-)generated profiles as it is in direct authorization profiles.

Composite Roles

Composite roles are used for structuring and to reduce the maintenance effort. They can bundle an arbitrary number of single roles. The structure of a composite role is shown in Figure 3.7. In contrast to a composite profile, composite roles can make use of the feature of inheritance. Therefore, the structure capabilities are better for roles than for direct profiles.

The typical usage of a composite role is that it reflects the position of a user with its tasks and responsibilities in a company whereas a single role holds all authorization information needed to perform one concrete task.

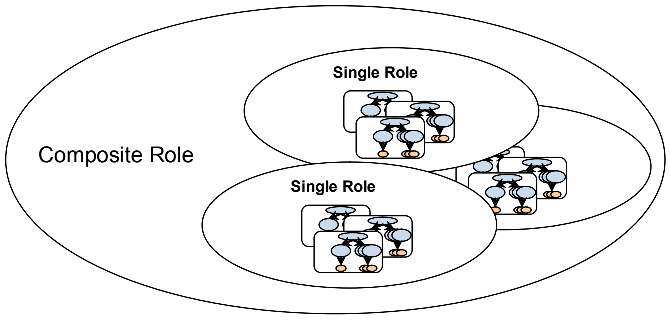


Figure 3.7: Structure of a composite role.

3.3 Business Processes

A business process is a sequence of interrelated steps which solve a particular issue. It can be part of another process and/or can also contain other processes.

Business processes are categorized into three different types:

1. *Management processes* govern the operation of a company. Typical management processes include “Corporate Governance” and “Strategic Management”.

2. *Operational processes* are processes that constitute the core business and create the primary value stream. Typical operational processes are Purchasing, Manufacturing, Marketing, and Sales.

3. *Supporting processes* support the core processes. Examples include Accounting, Recruitment or IT-support.

Management processes are indirectly modeled in SAP systems because they typically partly initiate or consist of other non-management processes. On the other hand, the operational and supporting processes are mapped in SAP systems by sequences of different transactions.

In general, all business processes that are mapped by a number of SAP transactions use the same authorization check mechanism. The analysis of all business processes which are mapped in an SAP system instance would go beyond the scope of this thesis. Therefore, I follow [23] and consider the purchase process as typical constituent of most SAP systems. The following sections introduce (the steps of) the purchase process used in this thesis and describe the process steps and the mandatory fields as well as its associated authorization objects.

The authorization check procedure is basically the concatenation of all authorization checks which occur in the execution of the transaction. The access is only granted if all single authorization checks have been passed successfully.

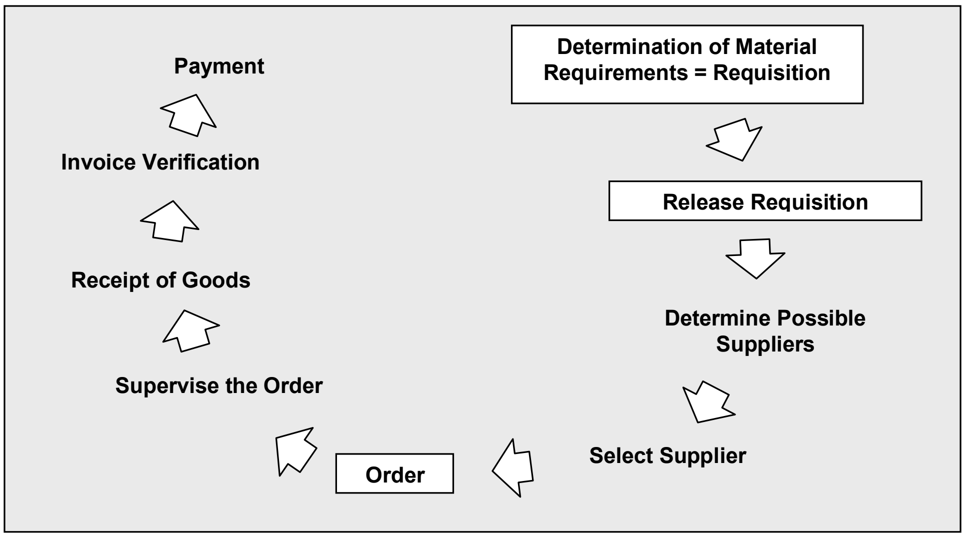


Figure 3.8: The purchase process.

A more comprehensive explanation including an example about the underlying authorizations and authorization check procedure is available in [23].

3.3.1 The Purchase Process

The purchase process consists of several steps which are depicted in Figure 3.8. In order to keep the case study in this thesis straightforward and comprehensible, we deal only with the creation of a requisition, the release and the eventual order (marked by a box in the figure). The creation of the requisition as well as the creation of the order is mapped by exactly one transaction in the SAP instance of the Max Planck Society. The release transaction is mapped by two transactions, one to view the requisition and the second to apply the release.

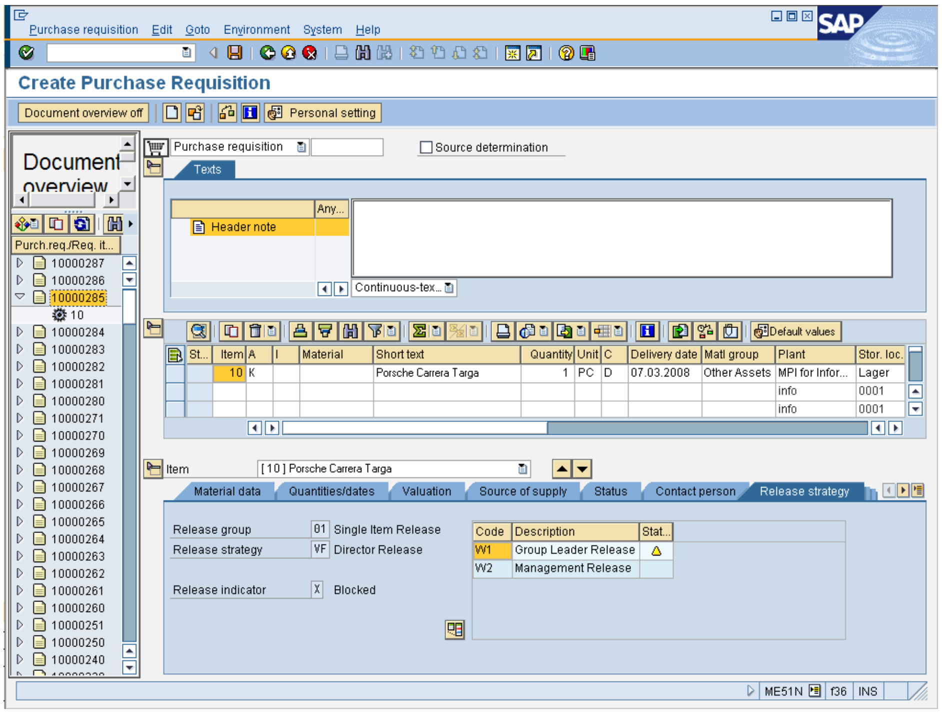
The following sections describe the essential data which must be entered to successfully create the entities. This is accompanied by screenshots of the system to give a rough idea of the appearance of the purchase transaction screens in the SAP system.

Create a Requisition

A purchase requisition document is created by a department and the purchasing group, respectively to request the purchase of goods or services. Such a document has some mandatory fields – the fields which must be filled in order to successfully finish the transaction. Therefore, the values of most of the required fields are subject to authorization checks.

The list of mandatory fields contains the *document type* (related to the authorization object `M_BANF_BSA`) – the type of the requisition object. Typical values for this field are `NB` which stands for regular or "normal business", `RSU` which stands for re-supply, or `SH` for special handling[3].

Another mandatory field is the *item number* which is located in the item area (see Figure 3.9). It prohibits the creation of empty requisitions.



Create Purchase Requisition

Purchase requisition ☐ Source determination

Purch.req./Req. it...
10000287
10000286
10000285
10
10000284
10000283
10000282
10000281
10000280
10000271
10000270
10000269
10000268
10000267
10000266
10000265
10000264
10000263
10000262
10000261
10000260
10000251
10000250
10000240

St...	Item	A	I	Material	Short text	Quantity	Unit	C	Delivery date	Matl group	Plant	Stor. loc.
	10	K			Porsche Carrera Targa	1	PC	D	07.03.2008	Other Assets	MPI for Infor...	Lager
											info	0001
											info	0001

Item [10] Porsche Carrera Targa

Release group	01	Single Item Release
Release strategy	VF	Director Release
Release indicator	X	Blocked

Code	Description	Stat...
W1	Group Leader Release	△
W2	Management Release	

Figure 3.9: Screen to create a requisition.

Each item which is going to be purchased is destined for a concrete *plant* (related to the authorization object `M_BANF_WRK`). Therefore, the plant is a mandatory field, too. The value depends on the structure of the company (and consequently of the customizing settings) and is typically prefilled. An empty value suggests a non-complete customizing.

Furthermore, each item must be assigned to a concrete purchasing group (related to the authorization object `M_BANF_EKG`). The term purchasing group is somewhat misleading because in the Max-Planck Society a purchasing group is often composed of just one person.

Each item requires the input of either a material number or a short description of the item. The material number is used in order to load the material related data from the material master record which is stored in the database.

[3]The available values in a specific SAP system instance depend on the customizing settings. Customizing is the adjustment of a default SAP system to match the business needs of a company.

Further mandatory fields like the material description, material group and the unit of measure are then prefilled automatically. The second option is to use just the material description. Then the fields like material group and unit of measure have to be filled manually.

Other required fields are, of course, the quantity of the items going to be be purchased, the delivery date and a price valuation.

The account assignment category is also mandatory. It determines which account assignment details are further required for an item (for example, the specification of a cost center or an account number).

The screenshot depicted in Figure 3.9 shows most of the previous mentioned fields of the screen on the basis of the requisition for a Porsche sports car.

Release a Requisition

Release procedures for requisitions are used in the SAP system to approve requisitions which exceed a certain budget limit before they can be converted to an order.

The SAP system uses so-called *release strategies* to achieve such approvals. They have to be defined in the customizing of the purchasing before the execution of any purchase activity.

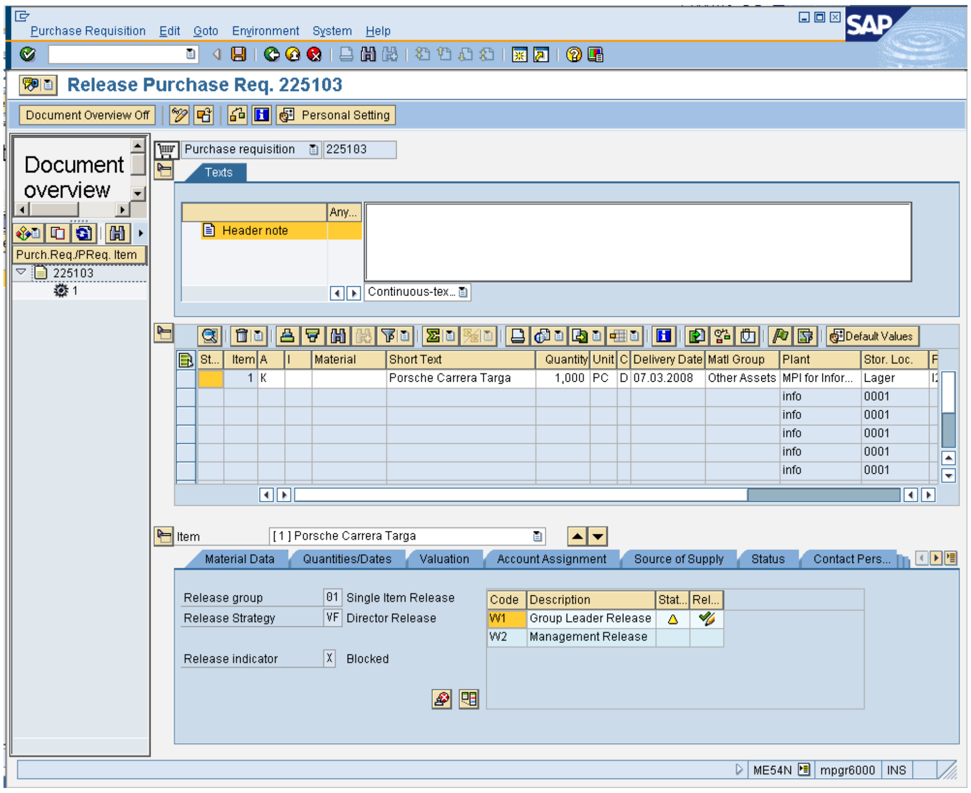
In order to provide a wide variety of possibilities the release strategies make use of the so-called *class system* which is a general component of the SAP system and not limited for use within release strategies. This class system is used to describe and classify any objects, for example, materials or requisitions, through *characteristics* in order to determine whether a certain strategy will be applied or not.

A release typically consists of several single release steps which have to be specified in the release strategy. Each step is defined by its *release code*. The release codes are again grouped in the *release group*. The so-called *release indicators* indicate the approval state of a requisition and define the possible actions after a certain release step.

The release strategy (entity) – denoted by its release strategy name – combines a characteristics class, the release codes, release group, and the release indicators. A more detailed explanation of the release strategy and its constituents including some examples can be found in [23].

The screenshot depicted in Figure 3.10 shows the release screen for a requisition. It continues the previous example of the Porsche sports car and shows the active release strategy `VF`, which stands for a management release. It consists of two single release steps: (i) the group leader (release code `W1`) and (ii) a person of the management, for example the director (owning the release code `W2`). The release of the group leader has already been passed but the director release is still open and needs to be completed to fully release the requstion.

The fields that are subject to authorization checks for the release step are the ones mentioned in Section 3.3.1, create a requisition, as well as the release



Release Purchase Req. 225103

225103
1

Purchase requisition 225103

St...	Item	A	I	Material	Short Text	Quantity	Unit	C	Delivery Date	Matl Group	Plant	Stor. Loc.	F
	1	K			Porsche Carrera Targa	1,000	PC	D	07.03.2008	Other Assets	MPI for Infor...	Lager	I:
											info	0001	
											info	0001	
											info	0001	
											info	0001	
											info	0001	

Item [1] Porsche Carrera Targa

Release group	01	Single Item Release
Release Strategy	VF	Director Release
Release indicator	X	Blocked

Code	Description	Stat...	Rel...
W1	Group Leader Release	△	✓
W2	Management Release		

ME54N mpgr6000 INS

Figure 3.10: Screen to release a requisition.

group and code (together in the authorization object `M_EINK_FRG`).

Create an Order

The order is the request to the supplier or another plant of the company to deliver the requisitioned (and released) material or services under the terms and conditions agreed before.

In practice there are often negotiations with possible suppliers about the exact prices, conditions and perhaps possible discounts after the release of a requisition. If the price or the condition changes, it is typically a minimal change. A big change results in the re-appliance of the release strategy, i.e., either the changed item or the overall requisition has to be released again, depending on the release settings.

Assume that the release has succeeded and the conditions have not changed. A released requisition can be transformed to an order (which is connected to the requisition) where, in contrast to the requisition, the order requires additionally at least accounting settings (for booking on a specific cost center, project, etc.), and a vendor.

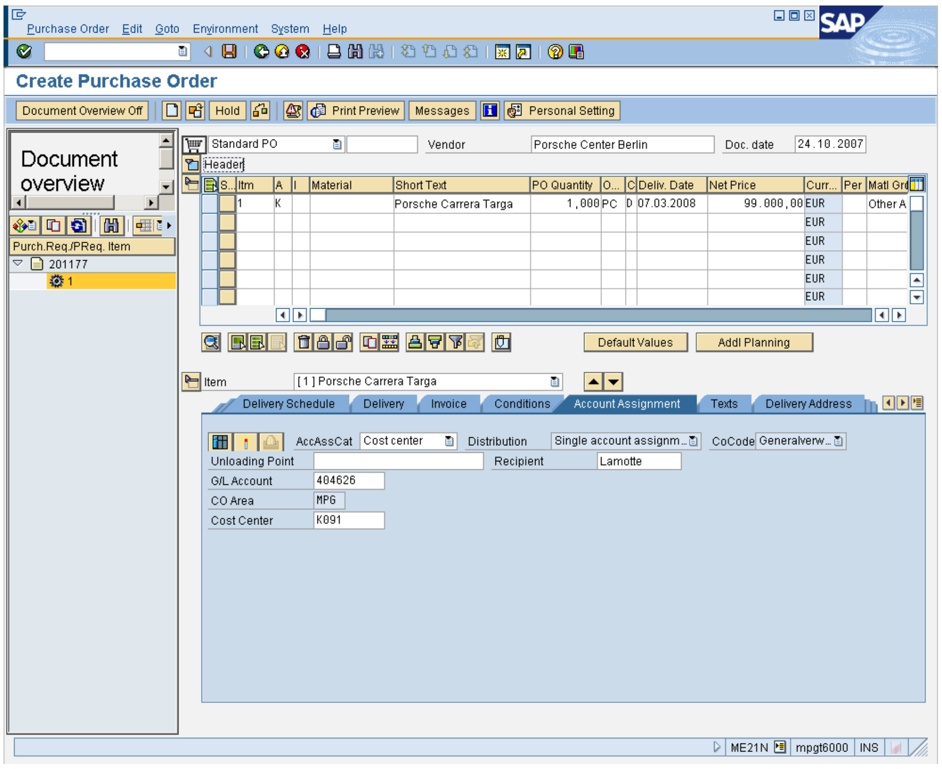
Most of the other required fields correspond to the ones available in the requisition: The document type (related to the authorization object `M_BEST_BSA`), plant (related to the authorization object `M_BEST_WRK`), and purchasing group (related to the authorization object `M_BEST_EKG`). In addition, there is the field purchasing organization (related to the authorization object `M_BEST_EKO`) that is subject to authorization checks.

Figure 3.11 depicts the order step for the Porsche sports car which has been previously requested and then released. After the order creation, the next step would be to print out the order and send it to the supplier.

The previously presented way to create orders is the default method. However, companies often need additional ways to create orders because the run through all steps of the complete purchase process often takes a lot of time. In such cases, the process is altered to allow alternative ways to create orders. For example, an extension could allow direct orders which do not require a released requisition but have restrictions to a certain type of material and/or a limit on the total amount of money.

3.4 Business Policies

Business policies are constraints on the business. According to the Business Rules Group [16] a business policy or business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business. Much of the industry's understanding of business policies has been historically shaped



Create Purchase Order

Standard PO | Vendor: Porsche Center Berlin | Doc. date: 24.10.2007

Header

S...	Itm	A	I	Material	Short Text	PO Quantity	O...	C	Deliv. Date	Net Price	Curr...	Per	Matl Grd
	1	K			Porsche Carrera Targa	1,000	PC	D	07.03.2008	99.000,00	EUR		Other A
											EUR		
											EUR		
											EUR		
											EUR		
											EUR		

Purch.Req./PReq. Item
201177
1

Item: [1] Porsche Carrera Targa

AccAssCat	Cost center	Distribution	Single account assignm...
CoCode	Generalverw...		
Unloading Point		Recipient	Lamotte
G/L Account	404626		
CO Area	MPG		
Cost Center	K091		

ME21N | mpgt6000 | INS

Figure 3.11: Screen to create an order.

by the Business Rules Group, which grouped business rules in one of four categories:

- Definitions of business terms
  The most basic element of a business rule is the language used to express it. The very definition of a term is itself a business rule which describes how people think and talk about things. Thus, defining a term is establishing a category of business rule.

- Facts relating terms to each other
  The nature or operating structure of an organization can be described in terms of the facts which relate terms to each other. To say that a customer can place an order is a business rule. Facts can be documented as natural language sentences or as relationships, attributes, and generalization structures in a graphical model.

- Constraints
  Every enterprise constrains behavior in a non-formal way, e.g., by saying that a complete process cannot be executed by just one person, is closely related to constraints on what data may or may not be updated. To prevent a record from being made is, in many cases, to prevent an action from taking place.

- Derivations
  Business rules (including laws of nature) define how knowledge in one form may be transformed into other knowledge, possibly in a different form.

The semantic essence of a business policy expresses a logical definition of some facet of the organization's way of doing business. An important feature of business policies is that they are usually specified by business people – the people who have responsibility for the business activities to which the rules apply.

The business policies considered in this thesis are related to definitions, facts, and constraints. A lot of the business policy concepts are general and applicable to many processes. However, in the context of this thesis they only impose restrictions in the area of the previously presented purchase process.

Today, one best-practice approach is the *Segregation of Duties* or *Separation of Duties* (SoD). It is basically the concept of having more than one person required to complete a task. The use of SoD leads to individual statements specifying conditions and/or limitations. Each statement is a business policy. In more detail, SoD means that there is no single individual having the control over two or more phases of a process, transaction or operation, respectively, so that a deliberate fraud is more difficult to occur because it requires collusion of two or more individuals or parties. There is no industry standard for the separation of duties, but the Information Systems Audit and Control Association (ISACA) has provided a segregation of duties control matrix [18], which is a general guideline suggesting which positions should be separated and which ones require compensating controls when combined.

A further approach – which is connected to SoD – is the *four-eyes principle*. It should be applied to critical activities or processes which should be performed by different persons to ensure the correctness of the transactions. As in the SoD concept, the application of the four eyes principle results in conditions and limitations – the business policy statements. In this thesis, we use a simplified form of this principle stating that the steps of a process cannot be execute by a single user. In other words, at least two different users are required to execute the standard purchase process.

As mentioned above, the business policies generally impact the execution of a business process. However, only the processes which are mapped in the SAP system as a sequence of transactions can be regarded. Therefore, a business policy which has been specified by business people in a business terminology has to be implemented in the SAP system by permissions and restrictions to a set of transactions.

4 Formalization of the SAP Authorization Layer

This chapter describes the main parts of the development of a formal model representing the authorization setup layer, the authorization check layer, and the business process layer from an SAP system by first-order formulas. It starts by taking some abstractions into consideration in Section 4.1 which affect the way of construction and the structure of the resulting formulas. The construction itself is presented in detail in Section 4.2, and is based on the work described in my Masters Thesis [23], therefore, several paragraphs throughout the whole chapter are adopted literally or in a rephrased form from my Masters Thesis. To the end of the formalization of the SAP authorization layer, the purchase process together with some typical business policies (see Sections 3.3.1 and 3.4) is reviewed under the aspects of the present formulas and the underlying structure of the resulting set of clauses.

4.1 Abstractions

The formal model presented in this thesis is based on a snapshot of an SAP system, i.e., the extracted authorization information (c.f. Chapter 3 for information about SAP authorizations). Correspondingly, the formal model represents the system at the given time the snapshot was taken and any change of the authorizations requires an adjustment of the formulas.

However, the formalization presented in this thesis makes some abstractions. The first one refers to time. It would be possible to model the time when an authorization check happens during the execution of a transaction in first-order logic but this would lead to an immense increase of the run-time of theorem provers because of the obvious state explosion and is, at least for our purpose of proving termination and correctness of the authorizations and the related business policies, not (yet) necessary.

Another assumption for the formalization are unique data. Dynamics in the data or the authorizations are not considered. A typical example of dynamics is a change of user authorizations while the user is performing a transaction.

The next abstraction is related to our example purchase process. We use only one item per requisition/order which means that a new requisition/order object is created for each item to be purchased.

Eventually, we have not modeled pattern matching of strings with wild-card symbols which is used during the authorization checks and the release strategy appliance checks. We have modeled only the asterisk wild-card symbol * (but no composed values containing this symbol) matching every required value.

A more detailed description of the abstractions can be found in [23].

4.2 Construction

The formalization of the authorization setup layer considered in this thesis includes the authorization roles and the generation of authorization profiles, and is described in Section 4.2.1. Afterwards, the formalizations of the authorization check and the purchase process are presented in Sections 4.2.2, and 4.2.3, respectively. The formulas are given in implication form, and if not explicitly stated, they are universally quantified.

4.2.1 Authorization Setup

Roles

A role contains either different authorizations (single role), or in turn further roles with authorizations (composite role), but it isn't possible to mix single roles and authorizations in one composite role level.

A single role is modeled by the unary atom *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 in turn represents the authorization which is mapped to the single role by the binary function *singleRoleEntry()*.

SingleRole(singleRoleEntry(<single role name>,
authObj(<auth object name>, <auth field>, <value>)))

Consider the following instance as an example:

SingleRole(singleRoleEntry(`ZLAMOTTE_BANF_INFO`*,*
authObj(`S_TCODE`*,* `TCD`*,* `ME51N`*)))*

It's part of the single role `ZLAMOTTE_BANF_INFO` granting the permission to create requisitions and contains an authorization object `S_TCODE` with the field `TCD` and the value `ME51N`.

A composite role is modeled using the unary atom *CompositeRole*. The function *compositeRoleEntry()* therein associates the specified single role with the composite role.

CompositeRole(compositeRoleEntry(<composite role name>,
<single role name>))

Consider as an example the following two instances where the single roles `ZLAMOTTE_ORDER_INFO_BASE` and `ZLAMOTTE_ORDER_INFO_INFO` are both contained in the (composite) role `ZLAMOTTE_ORDER_INFO`.

CompositeRole(compositeRoleEntry(`ZLAMOTTE_ORDER_INFO`*,*
`ZLAMOTTE_ORDER_INFO_BASE`*))*

CompositeRole(compositeRoleEntry(`ZLAMOTTE_ORDER_INFO`*,*
`ZLAMOTTE_ORDER_INFO_INFO`*))*

Profile Generation and Authorization Profiles

During an authorization check the required value of an authorization is compared with the value in the user's authorization profile. There are autogenerated and direct authorization profiles. Both types of profiles represent authorizations that a user holds.

The general authorization profile of a user which provides his/her effective authorizations is modeled using the unary atom *UserProfile*. The function *userProfileEntry()* maps the different authorizations to the user:

UserProfile(userProfileEntry(<user>,
authObj(<auth object name>, <auth field>, <value>)))

A concrete instance of this atom is similar to a single role instance; it also contains the authorization (the authorization object with its value). The difference to the single role instance is that it is directly mapped to the user by the function *userProfileEntry()*. Of course, a complete user authorization profile typically consists of many single instances of this atom. The following is an example for one instance, where the user `LAMOTTE` owns an authorization object `M_BANF_WRK` with the field `WERKS` and value `INFO`:

UserProfile(userProfileEntry(`LAMOTTE`*,*
authObj(`M_BANF_WRK`*,* `WERKS`*,* `INFO`*)))*

The value `INFO` in this example enables all actions of the user `LAMOTTE` concerning purchase requisitions to the plant `INFO`.

The previous mentioned atom *UserProfile()* models the effective authorizations of a user; any authorization check looks for the required authorization in these authorization profile instances. However, authorizations cannot be assigned directly because they are contained in structures like roles (or also direct profiles[1]). Therefore, the assignment of authorizations is accomplished by the assignment of the single or composite role to a user. This is modeled by the following predicate *Holds* denoting the fact that a user holds the authorizations of a single or composite role.

Holds(<user>, <single role name/composite role name>)

[1]Using roles rather than direct profiles is best practice, hence, we concentrate on the roles here. See [23] for more details on direct profiles.

The effective authorization instances (represented by instances of the atom *UserProfile()*) are then generated automatically by a transition which first checks the type of the item assigned to the user via the predicate *Holds*, extracts the authorization part and eventually creates the user profile instance (representing the effective authorization) for the user.

4.2.2 Authorization Checks

The authorization check result – access or decline – is represented in our first-order model by the binary atom *Access()*. If the atom is valid, the access to the checked authorization object is granted, otherwise it is not.

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

The function *authObj()* with arity 3 maps the authorization value to the authorization field of the authorization object. The resulting authorization is then associated with the user.

To ease specification of authorizations on a transaction base, the predicate *Access* has been overloaded to model the access to an overall transaction, too. Valid instances of the following form have the meaning that all individual authorization checks of the entire transaction have been successful.

Access(<user>, <transaction code>)

The authorization check itself compares the value of the authorization present in the (generated) user authorization profile with the required authorization. The result of the check is indicated by validity of the predicate *Access* which has been introduced before. The formalization presented in this thesis provides two basic ways to pass such an authorization check. The first way is that the user has the exact required authorization in its authorization profile. The authorization object name, the field and the value must match. This is achieved by the following formula:

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

The second way is that the authorization value of an authorization in the user's authorization profile is the wild-card symbol *. In this case, if the authorization objects match then the access is always granted, no matter what the required value is. The authorization in the user authorization profile must contain the constant `STAR` as a value which is depicted by the following transition.[2]

$\forall$ *xu, xaon, xaof, xav* .
  UserProfile(userProfileEntry(xu, authObj(xaon, xaof, `STAR`*)))*
$\rightarrow$ *Access(xu, authObj(xaon, xaof, xav))*

4.2.3 Purchase Process

The following sections describe the formalization of the purchase process as a business process example with its authorization checks.

Create a Requisition

The following transition shows the overloaded predicate symbol *Access* (introduced in Section 4.2.2) which models the single authorization checks that are needed for the execution of the transaction `ME51N` (create a requisition). The first check represents the check of the transaction code to create a requisition. The authorizations `M_BANF_WRK`, `M_BANF_BSA` and `M_BANF_EKG` have a constant in the first field (the authorization object field name like `ACTVT` or `WERKS`) and a variable value in their second field (that contains the authorization value). The atom *Access(xu,* `ME51N`*)* only becomes valid if the user *xu* has the exact constants in his authorization profile and some value for every variable value. This means that the user is allowed to execute the transaction in at least one instance, for example for one plant, one document type and one purchase group. The concrete value of the variables will and has to be evaluated and checked at run-time when a requisition is going to be created and the exact values are known.

$\forall$ *xu, xwrk, xbsa, xekg* .
  Access(xu, authObj(`S_TCODE`*,* `TCD`*,* `ME51N`*))* $\wedge$
  Access(xu, authObj(`M_BANF_WRK`*,* `ACTVT`*,* `01`*))* $\wedge$
  Access(xu, authObj(`M_BANF_WRK`*,* `WERKS`*, xwrk))* $\wedge$
  Access(xu, authObj(`M_BANF_BSA`*,* `ACTVT`*,* `01`*))* $\wedge$
  Access(xu, authObj(`M_BANF_BSA`*,* `BSART`*, xbsa))* $\wedge$
  Access(xu, authObj(`M_BANF_EKG`*,* `ACTVT`*,* `01`*))* $\wedge$
  Access(xu, authObj(`M_BANF_EKG`*,* `EKGRP`*, xekg))*
$\rightarrow$ *Access(xu,* `ME51N`*)*

Using the grouped authorizations (denoted by the transaction code), we model the business process steps which typically consist of several transactions. The purchase request for an arbitrary asset is modeled by the atom *Requisition*

[2] An additional monadic atom $\neg Authorization(xav)$ still has to be added to the formula in order to satisfy the conditions of the new class $\mathcal{BDI}$, defined in Section 5.2.

with arity 8. The structure is described as follows:

Requisition(<document type>, <position>, <material>, <plant>,
<purchasing group>, <purchasing organization>,
<material group>, <price>)

Similar to the request there is the state *RequisitionCreated* which indicates that the requisition object has been created by a user in the SAP system. The difference to the *Request* is that the user who has created the requisition is now also connected to it. This leads to an instance of the following atom with arity 9:

RequisitionCreated(<user>, <document type>, <position>, <material>,
<plant>,<purchasing group>, <purchasing organization>,
<material group>, <price>)

Using the previous two structures, we are now able to formally express the business step which represents the creation of a requisition:

$$
\begin{aligned}
&\forall\ xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt\ .\\
&\quad Requisition(xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xp) \land\\
&\quad Access(xu, \texttt{ME51N}) \land\\
&\quad Access(xu, authObj(\texttt{M\_BANF\_WRK}, \texttt{WERKS}, xwrk)) \land\\
&\quad Access(xu, authObj(\texttt{M\_BANF\_BSA}, \texttt{BSART}, xbsa)) \land\\
&\quad Access(xu, authObj(\texttt{M\_BANF\_EKG}, \texttt{EKGRP}, xekg)) \land\\
&\quad Access(xu, \texttt{ME53N}) \land\\
&\quad Access(xu, \texttt{ME52N})\\
&\rightarrow RequisitionCreated(xu, bsa, xpos, xmat, xwrk, xekg, xekorg,\\
&\qquad\quad xmatkl, xgswrt)
\end{aligned}
$$

The process starts with the existence of a purchase request modeled by the atom *Requisition.* In order to create a purchase requisition object in the SAP system, a user needs access to the corresponding transaction `ME51N`. The (overloaded) atom *Access(xu,* `ME51N`*)* stands for the group of single authorization checks in this transaction. As mentioned, the variables *xwrk* (plant), *xbsa* (document type), and *xekg* (purchasing group) must be checked again in the subsequent lines because only at this point the value of the variables is known (namely the value from the requisition that is going to be created). Typically, a user authorized to create a requisition is also authorized to view it or to change it in case of a mistake. This is modeled by the lines below checking the access to the transactions `ME53N` (View) and `ME52N` (Change). If all conditions are satisfied, then the state *RequisitionCreated* is implied. The result represents the state in the SAP system where the requisition object has been created.

Release a Requisition

The release of requisitions requires different settings in the Customizing of purchases. At first, these settings have to be formalized in order to apply a release strategy to a requisition.

The first step is the definition of the release strategy. It is modeled by the atom *ReleaseStrategy* with arity 3.

ReleaseStrategy(<release strategy name>, <release group>,
class(<characteristics class name>,
property(<property name>, <value>)))

The nested structure is required in order to obtain a formalization which is close to the structure of the real SAP system wherein the release strategy combines a characteristics class (containing single characteristics, each given by a property and its value(s)), the release group and code and the release strategy name. The binary function *property()* maps the value to its property which is in turn mapped to the class name by the binary function *class()*. At the end, the complete characteristic is assigned to the release strategy.

Consider the release strategy `VF` as an example. This strategy stands for a strategy called "director release strategy". The strategy `VF` in our example applies when the following conditions are satisfied:

i) the plant entered in the requisition is equal to the value `INFO`,

ii) the specified purchasing group is equal to `I26` and

iii) the total money amount of the item which is subject to the release strategy application check is greater than 10.000 EUR.

In the following formalization of this strategy, the different properties matching the fields of a requisition (`FRG_CEBAN_WERKS`=plant, `FRG_CEBAN_EKGRP`=purchasing group, and `FRG_CEBAN_GSWRT`=total amount of money) are grouped by the characteristics class (`FRG_EBAN`). Further details on the release strategy setup can be found in [23].

ReleaseStrategy(`VF`*,* `01`*, class(*`FRG_EBAN`*,*
property(`FRG_CEBAN_WERKS`*,* `INFO`*)))*

ReleaseStrategy(`VF`*,* `01`*, class(*`FRG_EBAN`*,*
property(`FRG_CEBAN_EKGRP`*,* `I26`*)))*

ReleaseStrategy(`VF`*,* `01`*, class(*`FRG_EBAN`*,*
property(`FRG_CEBAN_GSWRT`*,* `GREATER_10000_EUR`*)))*

Of course, we need to execute one or more single release steps depending on the applied strategy. Each of the steps again requires a release based on a release group and the release code. This requirement is modeled with the atom *ReleaseRequirement* with arity 3.

ReleaseRequirement(<release strategy name>, <release group>, <release code>)

The following two formulas again refer to the release strategy `VF` – the director release. This release strategy consists of two steps, the cost center release in the beginning (denoted by the constant `W1`) and the director release at the end (denoted by the constant `W2`). The constants `W1` and `W2` represent the respective release code in the SAP system. The release strategy and the release codes are grouped in the release group denoted by the constant/number `01`. In order to perform the first release step, the releasing person needs the right release group and release code in his authorization profile.

$$\mathit{ReleaseRequirement}(\texttt{VF}, \texttt{01}, \texttt{W1})$$

$$\mathit{ReleaseRequirement}(\texttt{VF}, \texttt{01}, \texttt{W2})$$

Using the previously defined atoms it is now possible to model a release step. The atom *RequisitionReleasedStep* with arity 11 in the following formula denotes a requisition item that has been released with a certain release group and code. In the following transition, the requisition released step of the release strategy *xfrgstrat* is valid if the requisition has been created by a user *xu1*, the release strategy *xfrgstrat* applies, the release requirement has been defined and the authorization checks succeed for a user *xu2*.

$$\begin{aligned}
&\forall\ \mathit{xu1, xu2, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt,}\\
&\qquad \mathit{xfrgstrat, xfrggr, xfrgco, xcl}\ .\\
&\quad \mathit{RequisitionCreated(xu1, xbsa, xpos, xmat, xwrk, xekg, xekorg,}\\
&\qquad\qquad \mathit{xmatkl, xgswrt)} \land\\
&\quad \mathit{ReleaseStrategy(xfrgstrat, xfrggr, class(xcl,}\\
&\qquad\qquad \mathit{property(}\texttt{FRG\_CEBAN\_EKGRP}\mathit{,xekg)))} \land\\
&\quad \mathit{ReleaseStrategy(xfrgstrat, xfrggr, class(xcl,}\\
&\qquad\qquad \mathit{property(}\texttt{FRG\_CEBAN\_WERKS}\mathit{,xwrk)))} \land\\
&\quad \mathit{ReleaseStrategy(xfrgstrat, xfrggr, class(xcl,}\\
&\qquad\qquad \mathit{property(}\texttt{FRG\_CEBAN\_GSWRT}\mathit{,xgswrt)))} \land\\
&\quad \mathit{ReleaseRequirement(xfrgstrat, xfrggr, xfrgco)} \land\\
&\quad \mathit{Access(xu2, authObj(}\texttt{M\_EINK\_FRG}, \texttt{FRGGR}, \mathit{xfrggr))} \land\\
&\quad \mathit{Access(xu2, authObj(}\texttt{M\_EINK\_FRG}, \texttt{FRGCO}, \mathit{xfrgco))} \land\\
&\quad \mathit{Access(xu2,}\ \texttt{ME54N}) \land\\
&\quad \mathit{Access(xu2, authObj(}\texttt{M\_BANF\_WRK}, \texttt{WERKS}, \mathit{xwrk))} \land\\
&\quad \mathit{Access(xu2, authObj(}\texttt{M\_BANF\_BSA}, \texttt{BSART}, \mathit{xbsa))} \land\\
&\quad \mathit{Access(xu2, authObj(}\texttt{M\_BANF\_EKG}, \texttt{EKGRP}, \mathit{xekg))}\\
&\to \mathit{RequisitionReleasedStep(xu2, xfrggr, xfrgstrat, xfrgco, xbsa, xpos, xmat,}\\
&\qquad \mathit{xwrk, xekg, xekorg, xmatkl, xgswrt)}
\end{aligned}$$

As mentioned, a release of a requisition typically consists of one ore more release steps. In the SAP test system of the Max-Planck Society there are two release strategies `KF` and `VF`. The former strategy represents the cost center release and consists of one step (with the code `W1`). The latter strategy denotes the already mentioned director release which consists of two steps (codes `W1` and `W2`). Both strategies belong to the release group `01`. The steps required for each execution variant of the release strategy are modeled by the following transition. The resulting state *RequisitionReleased* is valid, if all required release steps have been passed.

$$\begin{aligned}
&\forall\ \mathit{xu1, xu2, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt,}\\
&\qquad \mathit{xfrgstrat, xfrggr, xfrgco, xcl}\ .\\
&\quad \mathit{RequisitionReleasedStep(xu2,}\ \texttt{01}\mathit{,}\ \texttt{KF}\mathit{,}\ \texttt{W1}\mathit{, xbsa, xpos, xmat,}\\
&\qquad \mathit{xwrk, xekg, xekorg, xmatkl, xgswrt)} \lor\\
&\quad \mathit{(RequisitionReleasedStep(xu1,}\ \texttt{01}\mathit{,}\ \texttt{VF}\mathit{,}\ \texttt{W1}\mathit{, xbsa, xpos, xmat,}\\
&\qquad \mathit{xwrk, xekg, xekorg, xmatkl, xgswrt)} \land\\
&\quad \mathit{RequisitionReleasedStep(xu2,}\ \texttt{01}\mathit{,}\ \texttt{VF}\mathit{,}\ \texttt{W2}\mathit{, xbsa, xpos, xmat,}\\
&\qquad \mathit{xwrk, xekg, xekorg, xmatkl, xgswrt))}\\
&\to \mathit{RequisitionReleased(xu2, xbsa, xpos, xmat, xwrk, xekg,}\\
&\qquad \mathit{xekorg, xmatkl, xgswrt)}
\end{aligned}$$

If none of the available release strategies applies to a requisition, then it is immediately released and ready to order. A release strategy doesn't apply if at least one of the properties defined in the strategy doesn't match. So far, the theory only contains assertions that say that a certain property matches. In this case, however, it is required to verify whether a property does not match. For that reason, the theory must be extended with formulas for each possible value that defines when a release strategy does not apply. The following transition shows the transition for the plant property and its value `INFO`. It means, that if the value `INFO` matches the plant property, then, of course, other (possible) values like `SOFS` and `MPG` do not match. Similar formulas are required for each element of the Cartesian product of property and possible value.

$$\begin{aligned}
&\forall\ \mathit{xfrgstrat, xfrggr}\ .\\
&\mathit{ReleaseStrategy(xfrgstrat, xfrggr, class(}\texttt{FRG\_EBAN}\mathit{,}\\
&\qquad \mathit{property(}\texttt{FRG\_CEBAN\_WERKS}\mathit{,}\ \texttt{INFO}\mathit{)))}\\
&\to \neg \mathit{ReleaseStrategy(xfrgstrat, xfrggr, class(}\texttt{FRG\_EBAN}\mathit{,}\\
&\qquad \mathit{property(}\texttt{FRG\_CEBAN\_WERKS}\mathit{,}\ \texttt{SOFS}\mathit{)))} \land\\
&\neg \mathit{ReleaseStrategy(xfrgstrat, xfrggr, class(}\texttt{FRG\_EBAN}\mathit{,}\\
&\qquad \mathit{property(}\texttt{FRG\_CEBAN\_WERKS}\mathit{,}\ \texttt{MPG}\mathit{)))}
\end{aligned}$$

The ability to decide when a property does not match leads to the following transition. It models, that no release strategy (in our example neither `KF` nor `VF`) applies. Therefore, the requisition is automatically released and ready for an order.

$$\begin{aligned}
&\forall\ \mathit{xu, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt,}\\
&\qquad\quad \mathit{xfrgstrat, xfrggr, xfrgco, xcl}\ .\\
&\quad \mathit{RequisitionCreated(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg,}\\
&\qquad\qquad\qquad\qquad \mathit{xmatkl, xgswrt)} \wedge\\
&(\neg \mathit{ReleaseStrategy(}\texttt{KF}, \mathit{xfrggr, class(}\texttt{FRG\_EBAN},\\
&\qquad\qquad\qquad\qquad \mathit{property(}\texttt{FRG\_CEBAN\_EKGRP}, \mathit{xekg))))} \vee\\
&\neg \mathit{ReleaseStrategy(}\texttt{KF}, \mathit{xfrggr, class(}\texttt{FRG\_EBAN},\\
&\qquad\qquad\qquad\qquad \mathit{property(}\texttt{FRG\_CEBAN\_WERKS}, \mathit{xwrk)))} \vee\\
&\neg \mathit{ReleaseStrategy(}\texttt{KF}, \mathit{xfrggr, class(}\texttt{FRG\_EBAN},\\
&\qquad\qquad\qquad\qquad \mathit{property(}\texttt{FRG\_CEBAN\_GSWRT}, \mathit{xgswrt))))} \wedge\\
&(\neg \mathit{ReleaseStrategy(}\texttt{VF}, \mathit{xfrggr, class(}\texttt{FRG\_EBAN},\\
&\qquad\qquad\qquad\qquad \mathit{property(}\texttt{FRG\_CEBAN\_EKGRP}, \mathit{xekg)))} \vee\\
&\neg \mathit{ReleaseStrategy(}\texttt{VF}, \mathit{xfrggr, class(}\texttt{FRG\_EBAN},\\
&\qquad\qquad\qquad\qquad \mathit{property(}\texttt{FRG\_CEBAN\_WERKS}, \mathit{xwrk)))} \vee\\
&\neg \mathit{ReleaseStrategy(}\texttt{VF}, \mathit{xfrggr, class(}\texttt{FRG\_EBAN},\\
&\qquad\qquad\qquad\qquad \mathit{property(}\texttt{FRG\_CEBAN\_GSWRT}, \mathit{xgswrt))))}\\
&\rightarrow \mathit{RequisitionReleased(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg,}\\
&\qquad\qquad\qquad\qquad \mathit{xmatkl, xgswrt)}
\end{aligned}$$

Create an Order

A successful released requisition is the precondition to create an order object which is connected to the requisition in the SAP system. The following transition models this business step. At first, the existence of the released requisition is checked. It has been released by the user *xu1*. Then the access to the transaction `ME21N` is checked because it is required to create order objects (`ME22N`, `ME23N` to edit and view orders, respectively). The last checks ensure that the user is authorized to create orders for the plant, purchase document type, purchasing group and purchasing organization, respectively, which is specified by the released requisition. All checks must succeed to enter the state *OrderCreated*. Note, that the user who has released the requisition and the user who creates the order object can be the same person in this transition.

$$
\begin{array}{l}
\forall\ \mathit{xu1, xu2, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt}\ .\\
\quad \mathit{RequisitionReleased(xu1, xbsa, xpos, xmat, xwrk, xekg, xekorg,}\\
\qquad\qquad \mathit{xmatkl, xgswrt)} \wedge\\
\quad \mathit{Access(xu2,}\ \mathtt{ME21N}\mathit{)} \wedge\\
\quad \mathit{Access(xu2,}\ \mathtt{ME22N}\mathit{)} \wedge\\
\quad \mathit{Access(xu2,}\ \mathtt{ME23N}\mathit{)} \wedge\\
\quad \mathit{Access(xu2, authObj(}\mathtt{M\_BEST\_WRK}, \mathtt{WERKS}, \mathit{xwrk))} \wedge\\
\quad \mathit{Access(xu2, authObj(}\mathtt{M\_BANF\_BSA}, \mathtt{BSART}, \mathit{xbsa))} \wedge\\
\quad \mathit{Access(xu2, authObj(}\mathtt{M\_BEST\_BSA}, \mathtt{BSART}, \mathit{xbsa))} \wedge\\
\quad \mathit{Access(xu2, authObj(}\mathtt{M\_BEST\_EKG}, \mathtt{EKGRP}, \mathit{xekg))} \wedge\\
\quad \mathit{Access(xu2, authObj(}\mathtt{M\_BEST\_EKO}, \mathtt{EKORG}, \mathit{xekorg))}\\
\rightarrow \mathit{OrderCreated(xu2, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt)}
\end{array}
$$

4.2.4 Business Policies

The business policies are expressed by business people in a business terminology. They impact the way of performing the business processes. For example, they prohibit the possibility for a user to perform all process steps alone. The connection between the business process (steps) and the SAP system is quite simple – there is a mapping which maps a number of transactions to each process step.

Our formalization already contains the mapping to the individual transactions, so it is easy to also model business policies.

The general but for this thesis simplified business policy that has been introduced in Section 3.4 prohibits the execution of the complete process consisting of the steps "create a requisition", "release a requisition" and "create the order" for this requisition by one single user for one concrete plant, material group, purchasing group and organization. The following formula represents this policy: There is no requisition such that one user *xu* can reach all states *RequisitionCreated*, *RequisitionReleased* and *OrderCreated*. Of course, the assumption for such a policy are unique data in the SAP system.

$$
\begin{array}{l}
\neg\exists\ \mathit{xu, xbsa, xwrk, xekg, xekorg, xmatkl, xgswrt}\ .\\
\quad \forall\ \mathit{xpos, xmat}\ .\\
\qquad \mathit{Requisition(xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt)}\\
\rightarrow \mathit{RequisitionCreated(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg,}\\
\qquad\qquad \mathit{xmatkl, xgswrt)} \wedge\\
\qquad \mathit{RequisitionReleased(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg,}\\
\qquad\qquad \mathit{xmatkl, xgswrt)} \wedge\\
\qquad \mathit{OrderCreated(xu, xbsa, xpos, xmat, xwrk, xekg, xekorg,}\\
\qquad\qquad \mathit{xmatkl, xgswrt)}
\end{array}
$$

Another policy is the requirement that a corresponding requisition has to be created before any order. This is stated by the following formula which expresses the fact that if we have an order created, there must exist a corresponding requisition.

$$
\begin{aligned}
&\forall\ \mathit{xu1, xbsa, xpos, xmat, xwrk, xekg, xekorg, xmatkl, xgswrt}\ .\\
&\exists\ \mathit{xu2}\ .\\
&\quad \mathit{OrderCreated(xu, xbsa, xpos, xmat, xwrk, xekg,xekorg,}\\
&\qquad\qquad \mathit{xmatkl, xgswrt)}\\
&\rightarrow \mathit{RequisitionCreated(xu2, xbsa, xpos, xmat, xwrk, xekg, xekorg,}\\
&\qquad\qquad \mathit{xmatkl, xgswrt)}
\end{aligned}
$$

4.2.5 Purchase Process + Business Policies Reviewed

In order to automatically prove properties by means of an automated theorem prover for clause sets resulting from authorization layer formalizations of SAP systems, it is a good idea to show decidability for the given inputs first. This is done by proving the input clause sets to be terminating which is only feasible if their structure is determined and necessary (structural) properties can be established. Having this idea in mind, I have created a clause dependency graph for analysis of the clauses' structure.

Let N be the set of clauses representing a formalized SAP authorization layer instance. The graph is a directed graph with $V = \mathcal{R}$ (all predicate symbols) and edges $E = \{(P,Q)\,|\,C = \Gamma, P(\vec{x}) \rightarrow \Delta, Q(\vec{y}\})$ for all clauses $C \in N$ with corresponding predicate symbols P, Q. This graph is depicted in figure 4.1 for such a clause set.

The vertices of the graph having only outgoing edges represent the (usually ground) facts of the authorization setup (i.e., the set of roles, authorizations, and the assignment to users), and the requirement(s) to enter the process(es) (in our example the vertices *Requisition*, *StandardPurchase*, and *DirectPurchase*[3] for the purchase process).

Further inspection of the graph shows that it contains a cycle with the vertices *RequisitionCreated* $\rightarrow$ *ReleaseStrategy* $\rightarrow$ *RequisitionReleasedStep* $\rightarrow$ *RequisitionReleased* $\rightarrow$ *OrderCreated*. The vertex *RequisitionCreated* is the entry point to the cycle, and *OrderCreated* the last vertex before we go back to the entry point. Cycles become a problem when we can execute a cycle more than once and always derive new (and larger) clauses that have never been derived before in previous loops. If this happens here, theorem provers won't terminate being applied to SAP authorization instances occupying this structure.

Therefore, I have explored the clause(s) causing the edge *OrderCreated* $\rightarrow$

[3]This is a special variant of the purchase process where no release step is necessary but is underlying restrictions regarding the amount of money and the type of material. See [23] for details.

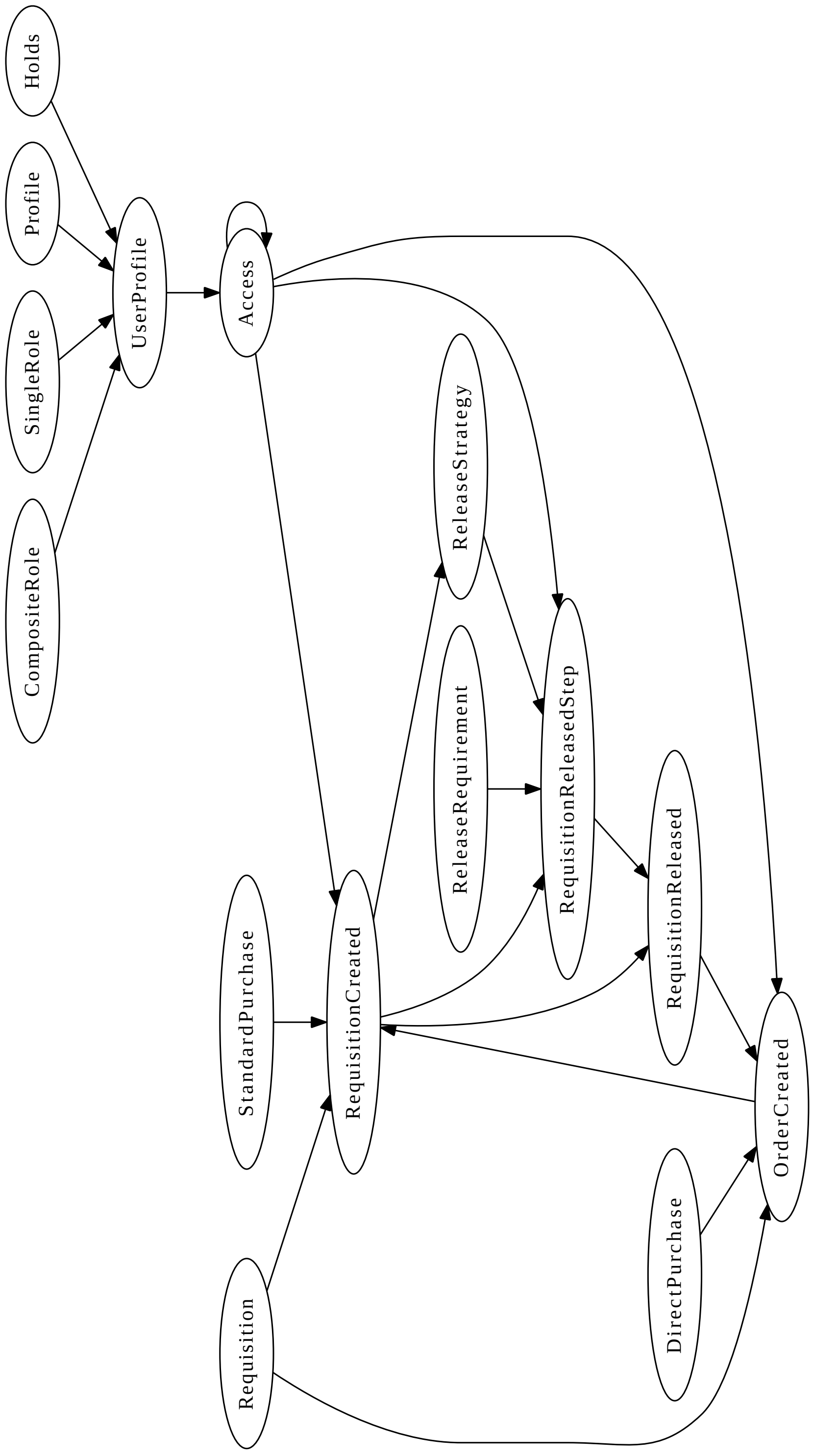


Figure 4.1: Dependency graph.

RequisitionCreated and thus closing the cycle. The culprit is

$$\begin{gathered}\mathit{OrderCreated(xu,\ xbsa,\ xpos,\ xmat,\ xwrk,\ xekg,\ xekorg,xmatkl,\ xgswrt)} \rightarrow \\ \mathit{RequisitionCreated(f(xgswrt,\ xmatkl,\ xekorg,\ xekg,\ xwrk,\ xmat,\ xpos,\ xbsa),} \\ \mathit{xbsa,\ xpos,\ xmat,\ xwrk,\ xekg,\ xekorg,xmatkl,\ xgswrt)}\end{gathered}$$

expressing the business policy that an order cannot be created without a previous requisition (c.f. Section 4.2.4). The requisition has to be created by an arbitrary user, and the item to order depends on that user. This is reflected by the function *f(xgswrt, xmatkl, xekorg, xekg, xwrk, xmat, xpos, xbsa)* in the atom $\mathit{RequisitionCreated}(\vec{x})$. Unfortunately, the overall term depth of the succedent in the clause above is larger than the depth of the antecedent (we have one more function symbol, please see the definition of a depth increasing clause in Section 5.2). During the saturation process, we will derive a clause that is larger than the parent clause (because of the function symbol f) and enter the loop again.

Further inspection shows that we have other clauses in the cycle where the term depth of the succedent also increases compared to the antecedent part. One such candidate is the clause

$$\begin{aligned}&\mathit{RequisitionCreated(xu,\ xbsa,\ xpos,\ xmat,\ xwrk,\ xekg,\ xekorg,\ xmatkl,\ xgswrt)},\\ &\mathit{ReleaseGroup(xfrggr)}, \mathit{ReleaseClass(xcl)}\\ &\rightarrow \mathit{ReleaseStrategy(}\texttt{KF}\mathit{,\ xfrggr,\ class(xcl,property(}\texttt{FRG\_CEBAN\_WERKS}\mathit{,xwrk)))},\\ &\mathit{ReleaseStrategy(}\texttt{VF}\mathit{,\ xfrggr,\ class(xcl,property(}\texttt{FRG\_CEBAN\_WERKS}\mathit{,xwrk)))},\\ &\mathit{RequisitionReleased(xu,\ xbsa,\ xpos,\ xmat,\ xwrk,\ xekg,\ xekorg,xmatkl,\ xgswrt)}\end{aligned}$$

representing the fact that the release strategy `KF` or `VF` is applied to a created requisition, or the requisition is immediately released if no release strategy is applicable. Variants of this clause with same structure exist for all possible combinations of release class properties. The clause shown here employs the `FRG_CEBAN_WERKS` property matching the entered plant in the requisition. One can easily see that the maximum term depth in both *ReleaseStrategy* atoms is larger than the maximum term depth in the antecedent part. The reason here is the structure of the release strategy which owns a release class and properties that are mapped to the strategy via functions.

The following Chapter lifts the mentioned structure and its inherent problems to a more general theoretical level and describes, how termination can be achieved even though there are cycles and depth increasing clauses.

5 The Clause Class $\mathcal{BDI}$

Showing decidability for restricted subclasses of first-order logic is one of the current research topics. The application of automated theorem proving in first-order logic for verifying and proving properties in the SAP authorization system considered in this thesis is a real-world application. For a practical use, general decidability of input problems given to a theorem prover, identified by a defined structure is of bigger importance than decidability only for specific (SAP) instances. Therefore, this thesis presents a class with such a defined structure for the clauses representing the SAP authorization layer and proves it to be terminating. The typical and used way in this thesis to prove termination is to show that both the clause size (i.e., the number of literals in a clause) and the term depth are bounded during the saturation of the input clauses.

The following clause set is a good and general example for non-terminating resolution:

$$\begin{array}{lrcl} (1) & & \rightarrow & P(a, f(a)) \\ (2) & P(x,y) & \rightarrow & Q(f(g(y)), y) \\ (3) & Q(x,y) & \rightarrow & R(f(x), x) \\ (4) & R(f(x), x) & \rightarrow & P(f(x), x) \end{array}$$

A quick inspection reveals the recursive definition of the predicate P through the clauses (2)-(4). The interested reader may notice two places of depth increase: (a) the variable y in clause (2) in the first argument of $Q(f(g(y)), y)$, and (b) the variable x in clause (3) in the first argument of $R(f(x), x)$. A clause is generally depth increasing, if there is a variable occurring in an atom on the succedent of a clause and this occurrence is at a deeper position than the deepest occurrence of the same variable in the antecendent of the (same) clause. As a consequence of the depth increase, the application of (Hyper-)resolution between the clauses (1) and (2) yields a depth increased clause

$$(1') \quad \rightarrow \quad Q(f(g(f(a))), f(a)).$$

Further resolution on $(1')$ and clause (3) produces the clause

$$(2') \quad \rightarrow \quad R(f(f(g(f(a)))), f(g(f(a)))).$$

The resolvent $(2')$ in turn resolved with clause (4) then yields

$$(3') \quad \rightarrow \quad P(f(f(g(f(a)))), f(g(f(a)))),$$

which restarts the cycle with an increased second argument $f(g(f(a)))$ of the atom $P(\ldots)$ in clause $(3')$, compared to the atom $P(a, f(a))$ and its second argument $f(a)$ in clause (1).

In order to change this example to be terminating (without regard of the semantics here because we're just interested in termination), there are several possibilities: One solution is to generally prevent depth increases for all clauses, and the other one – used by the presented new class Bounded Depth Increasing ($\mathcal{BDI}$) in this thesis – is to restrict the form of recursive definitions for predicates. The idea is to "watch" certain argument terms of an atom which are assumed to never increase during any derivation and argument positions holding terms with increased variable depth only depend on such watched arguments. In this way, a recursion is only possible with the same term/argument than in the first loop and thus not generating larger resolvents.

The following clause set is one possible modification of the previous example, but matching the $\mathcal{BDI}$ conditions (Section 5.2), and therefore terminating:

$$\begin{array}{lrcl}
(1) & & \rightarrow & P(a, f(a)) \\
(2) & P(x,y) & \rightarrow & Q(f(g(y)), y) \\
(3) & Q(x,y) & \rightarrow & R(f(y), y) \\
(4) & R(f(x), x) & \rightarrow & P(x,x)
\end{array}$$

There are still two depth increasing clauses (2) and (3) but in a non-critical form (different than before). This is because the second argument position is watched for all the predicates being part of the cycle ($P \rightarrow Q \rightarrow R \rightarrow P$). During recursion, only the arguments at these watched positions are considered and thus termination is guaranteed.

This chapter is organized as follows: It starts by constituting some additional notions that are needed to define the properties of the class $\mathcal{BDI}$. Section 5.2 contains the actual definition of the $\mathcal{BDI}$ class, including some examples to gain a better understanding of the conditions. The full termination proof is given in Section 5.3 and uses Hyper-resolution as the solely inference rule, and Factoring (Factorization) as the reduction rule. Afterwards, the termination result is generalized to Ordered Resolution in Section 5.4 because Hyper-resolution enumerates all ground facts from a given clause set which is not feasible for practical applications.

5.1 Prerequisites

A lot of clauses (especially all the ground facts) from the input of an SAP authorization formalization already belong to the class $\mathcal{PVD}$ (Positive Variable Dominated) [12].

Definition 5.1 ($\mathcal{PVD}$)
A clause $\Gamma \rightarrow \Delta$ is $\mathcal{PVD}$ (Positive Variable Dominated) [12] if

(i) $\mathit{vars}(\Delta) \subseteq \mathit{vars}(\Gamma)$ (Δ is ground for $\Gamma = \emptyset$),

(ii) $\mathit{depth}(x, \Delta) \leq \mathit{depth}(x, \Gamma)$ for all $x \in \mathit{vars}(\Delta)$.

The class $\mathcal{PVD}$ has already been proven to be decidable by Hyper-resolution in [13]. It is the starting point for the new class definition of $\mathcal{BDI}$. In contrast to the clause class $\mathcal{PVD}$ where the maximal depth of any derived clause by

Hyper-resolution does not exceed the maximal depth of its parent clauses, the class $\mathcal{BDI}$ permits to have such a growth of the term depth for a derived clause. Clearly, this relaxation requires additional restrictions in order to guarantee that Hyper-resolution still remains a decision procedure for $\mathcal{BDI}$.

Definition 5.2 (Depth Increasing)
We call a clause $C = \Gamma \to P(t_1, \ldots, t_n), \Delta$ *depth increasing* if there is a variable $x \in \mathit{vars}(C)$ and $\mathit{depth}(x, t_i) > \mathit{depth}(x, \Gamma)$ for some t_i where $1 \leq i \leq n$. The variable x is called a *depth increasing variable* in C, $P(t_1, \ldots, t_n)$ a *depth increasing atom* in C, P a *depth increasing predicate* in C, and i a *depth increasing argument position* of P.

A clause $C = \Gamma \to P(t_1, \ldots, t_n), \Delta$ is called *uniquely depth increasing* if C is depth increasing, and there is exactly one depth increasing argument position i of $P(t_1, \ldots, t_n)$ such that for all depth increasing variables $x \in \mathit{vars}(C)$ and $x \notin \mathit{vars}(t_i)$ it holds $\mathit{depth}(x, \{P(t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n), \Delta\}) \leq \mathit{depth}(x, \Gamma)$. Given a clause set N, a depth increasing clause $C \in N$ is called a *uniquely depth increasing clause in N for the predicate P at argument position i* if there is no different depth increasing clause for the same predicate P in N with depth increasing argument position $j \neq i$.

The later definitions of $\mathcal{BDI}$ speak in addition to the so far established term of reachability of predicates (Definition 2.32) about a general reachability from a depth increasing clause: A predicate symbol P occurring in a clause of a clause set N is called *reachable from a depth increasing clause* if there is a clause $(\Gamma \to Q(\vec{t}), \Delta) \in N$, with depth increasing predicate Q, and P is reachable from Q.

Consider the following set N as a motivating example for the below definition 5.3 of watched arguments:

$$\begin{array}{lrcl}
(1) & & \to & P(f(a), b, c) \\
(2) & P(x, y, z) & \to & Q(f(x), y, z) \\
(3) & Q(x, y, z) & \to & P(x, y, z)
\end{array}$$

(Hyper-)resolution applied on N computes infinitely many clauses of the form $Q(f^i(a), b, c)$. The reason is, in particular, the second clause, where the depth of the occurrence of x in the succedent (term $f(x)$) is strictly larger than its depth in the antecedent (term x). In order to exclude such a situation, the non-increasing arguments of $Q(f(x), y, z)$ are "watched", which are y, z (the second and third argument). Due to the existing cycle between the second and third clause, the second and third argument in the atoms with predicate symbol P are also watched. In the case of a depth increase comparing the maximal term depth of the atoms on the right hand side and the maximal term depth occurring in the atoms on the left hand side (as it is the case for the second clause), we require for the second clause that only variables from the watched arguments occur inside the depth growing terms. This means for the example, that if only the variables y or z are arguments of the function f in the second clause, the infinite nesting does not occur.

Definition 5.3 (Watched arguments)
Let *warg* be a function from predicate symbols to sequences of direct argument positions such that if $\mathit{warg}(P) = [i_1, \ldots, i_n]$ then $1 \leq i_j \leq m$, $0 \leq n \leq m$, and $i_j < i_k$ for $j < k$ where m is the arity of P. In case $\mathit{warg}(P) = [i_1, \ldots, i_n]$ then any i_j is called a *watched argument* of P. The function *warg* is extended to atoms by:

$$\mathit{warg}(P(t_1, \ldots, t_m)) = [P(t_1, \ldots, t_m)|_{i_1}, \ldots, P(t_1, \ldots, t_m)|_{i_n}] \,.$$

Example 5.4
Let $P(f(x, y), x, y, z)$ be an atom. Then, $\mathit{arity}(P) = m = 4$.

- If no arguments for P are watched, then $\mathit{warg}(P) = [\,]$.
- If we assume 2 arguments of P to be watched, for example, the second and fourth argument, then $\mathit{warg}(P) = [i_1, i_2]$ with $i_1 = 2$ and $i_2 = 4$. The watched arguments are represented as an ordered list where the arguments are read from the left to the right. Any other order is not permitted. This requirement is ensured by the condition $i_j < i_k$ for $j < k$ implying that the first watched argument must occur before the second watched argument in the atom.

Continuing the example, the extension of the *warg* function to atoms yields for $\mathit{warg}(P(f(x, y), x, y, z)) = [x, z]$.

Especially for the termination proof, it is necessary to define the notion of expressing the fact which clause has first derived a certain (ground) literal, i.e., from which clause a literal is originally coming from.

Definition 5.5 (Origination)
Let N be a set of clauses. *Origination* is defined inductively by:

(i) For all input clauses $C \in N$ each of their literals $L \in C$ *originates* from C.

(ii) For all hyper-resolution derived clauses $\rightarrow \Delta\sigma, \Delta_1, \ldots, \Delta_n$ from parent clauses $C = Q_1(s_{1,1}, \ldots, s_{1,m_1}), \ldots, Q_n(s_{n,1}, \ldots, s_{n,m_n}) \rightarrow \Delta$, $D_i = \rightarrow Q_i(u_{i,1}, \ldots, u_{i,m_i}), \Delta_i$, the literals $L\sigma \in \Delta\sigma$ *originate* from the clause C, and each $L'\sigma \in \Delta_i$ *originates* from the clause D_i.

5.2 Definition of $\mathcal{BDI}$ & Examples

The definition of the class $\mathcal{BDI}$ makes use of two sub-definitions BDI-1 and BDI-2 which take two different structural depth increases into account. The full definition of $\mathcal{BDI}$ uses – along with PVD – the two sub-definitions and a further condition restricting the interaction of two depth increasing clauses.

Definition 5.6 (BDI-1)
Let N be a set of clauses and *warg* a watched argument function. A clause $C = \Gamma \to P_1(t_{1,1}, \dots, t_{1,n_1}), \dots, P_m(t_{m,1}, \dots, t_{m,n_m}), \Delta$ from N with $1 \leq i \leq m$ satisfies BDI-1 if C is depth increasing, and

(i) $\mathit{vars}(\{P_1(t_{1,1}, \dots, t_{1,n_1}), \dots, P_m(t_{m,1}, \dots, t_{m,n_m}), \Delta\}) \subseteq \mathit{vars}(\Gamma)$, and $\mathit{depth}(x, \Delta) \leq \mathit{depth}(x, \Gamma)$ for all $x \in \mathit{vars}(\Delta)$

(ii) for all $C' = P_i(s_1, \dots, s_n), \Gamma' \to \Delta' \in N$ where $P_i(s_1, \dots, s_n)\sigma = P_i(t_{i,1}, \dots, t_{i,n_i})\sigma$ for some unifier σ, the atoms $P_i(s_1, \dots, s_n)$ and $P_i(t_{i,1}, \dots, t_{i,n_i})$ are similar, and for all depth increasing variables x, positions p, variables y, argument positions j where $t_{i,j}|_p = x$, $s_j|_p = y$ with $y \in (\mathit{vars}(P_i(s_1, \dots, s_n)) \cap \mathit{vars}(\Delta'))$ it holds $\mathit{depth}(y, \Delta') = 0$

(iii) for all $P_i(t_{i,1}, \dots, t_{i,n_i})$ holds $\mathit{warg}(P_i(t_{i,1}, \dots, t_{i,n_i})) = [\,]$

(iv) for all atoms $Q_k(\vec{r}_k), R_l(\vec{v}_l) \in \Gamma$ where Q_k is reachable from a depth increasing clause in N and R_l is not reachable from a depth increasing clause holds

$$\mathit{vars}(P_i(t_{i,1}, \dots, t_{i,n_i})) \subseteq \bigcup_k \mathit{vars}(\mathit{warg}(Q_k(\vec{r}_k))) \cup \bigcup_l \mathit{vars}(R_l(\vec{v}_l))$$

(v) for all atoms $Q(\vec{r}) \in \Gamma$ holds

$$(\mathit{warg}(Q(\vec{r})) = [\,] \text{ or for all } R(\vec{v}) \in \Delta \text{ it holds } \mathit{warg}(Q(\vec{r})) = \mathit{warg}(R(\vec{v})))$$

BDI-1-(ii) ensures that any derived atom from a clause satisfying BDI-1 with increased depth (compared to its parent clauses) cannot further contribute to the growth in depth in the next hyper-resolution step where the atom with the increased depth is considered as a parent clause. The R atoms in the clause set presented in Section 1.2 are an example.

BDI-1-(iv) prevents to have two consecutive depth increases in an argument when two consecutive hyper-resolution inference steps with depth increasing clauses take place.

BDI-1-(v) prevents position swapping of previously increased arguments in the non-depth increasing arguments of a clause satisfying BDI-1. The idea is to require for any atom in the succedent with a non-empty watched argument list that the watched argument list for all atoms in the antecedent is either identical or empty.

Definition 5.7 (BDI-2)
Let N be a set of clauses and *warg* a watched argument function. A clause $C = \Gamma \to P(t_1, \dots, t_j, \dots, t_n), \Delta$ from N satisfies BDI-2 if C is a uniquely depth increasing clause in N for the predicate P at argument position j, and

(i) $\mathit{vars}(\{P(t_1, \dots, t_j, \dots, t_n), \Delta\}) \subseteq \mathit{vars}(\Gamma)$

(ii) for all $i \neq j$ holds $t_j \notin \mathit{warg}(P(t_1, \ldots, t_n))$ and $t_i \in \mathit{warg}(P(t_1, \ldots, t_n))$

(iii) for all atoms $Q(s_1, \ldots, s_n) \in \Gamma$ where Q is reachable from P and $\mathit{vars}(Q(s_1, \ldots, s_n)) \cap \mathit{vars}(P(t_1, \ldots, t_n)) \neq \emptyset$:

  (1) $\mathit{arity}(Q) = \mathit{arity}(P)$

  (2) $\mathit{warg}(Q(s_1, \ldots, s_n)) = \mathit{warg}(P(t_1, \ldots, t_n))$

  (3) $\mathit{vars}(s_j) \cap \mathit{vars}(P(t_1, \ldots, t_n)) = \emptyset$

(iv) for all clauses $C' \in N$ with $C' = \Gamma' \to \Delta'$ which have an atom whose predicate is reachable from P, it holds for all atoms $Q(\vec{r}) \in \Gamma'$ that

$$(\mathit{warg}(Q(\vec{r})) = [\,] \text{ or for all } R(\vec{v}) \in \Delta' \text{ it holds } \mathit{warg}(Q(\vec{r})) = \mathit{warg}(R(\vec{v})))$$

(v) for all atoms $S(v_1, \ldots, v_m) \in \Delta$ and $Q_k(\vec{r}_k), R_l(\vec{v}_l) \in \Gamma$ where Q_k is reachable from a depth increasing clause and R_l is not reachable from a depth increasing clause holds

$$\mathit{vars}(S(v_1, \ldots, v_m)) \subseteq \bigcup_k \mathit{vars}(\mathit{warg}(Q_k(\vec{r}_k))) \cup \bigcup_l \mathit{vars}(R_l(\vec{v}_l))$$

Please note that condition BDI-2-(iii) implies that the depth increasing atom has at least two arguments. BDI-2-(iii) takes care of the depth inside the depth increasing atom of a clause satisfying BDI-2. In a clause set N with recursive predicate definitions, this condition restricts the way of increasing the depth in order to prohibit an unbounded growth of depth. BDI-2-(iv) prevents the "transfer" of a term with increased depth in a literal to another literal inside a different clause.

BDI-2-(iv) and BDI-2-(v) guarantee that depth increasing cycles cannot be used several times with the same depth increasing term, analogous to the corresponding conditions in BDI-1-(iv) and BDI-1-(v).

Consider the following set of clauses as a contradicting example for BDI-2:

$$\begin{array}{lrcl} (1) & P(x,y), Q(z,y) & \to & P(f(z),y) \\ (2) & P(x,y) & \to & Q(x,y) \\ (3) & & \to & P(a,b) \end{array}$$

In this example, the clause (1) does not satisfy BDI-1, nor BDI-2, nor $\mathcal{PVD}$. It does not satisfy $\mathcal{PVD}$ because it is depth increasing, nor does it satisfy BDI-1 because the occurrence of the atom $P(f(z), y)$ is not similar to $P(x, y)$ occurring in clause (2) which is required by BDI-1-(ii). And eventually, it also does not satisfy the conditions of BDI-2, because there is the atom $Q(z, y)$, Q is reachable from P but BDI-2-(iii)-(3) is violated by the variable z in $Q(z, y)$. The clauses (2) and (3) both satisfy $\mathcal{PVD}$.

Definition 5.8 ($\mathcal{BDI}$)
Let N be a set of clauses and warg a watched argument function. The set N belongs to $\mathcal{BDI}$ (Bounded Depth Increasing) if for all $C \in N$:

(i) C satisfies $\mathcal{PVD}$, or

(ii) C satisfies BDI-1, or

(iii) C satisfies BDI-2,

and, additionally, for two depth increasing clauses $\Gamma \rightarrow P(t_1, \ldots, t_n), \Delta$ and $\Gamma' \rightarrow Q(t'_1, \ldots, t'_{n'}), \Delta'$ with depth increasing predicates P and Q satisfying BDI-2,

(iv) the predicate Q is not reachable from P and vice versa.

In the context of a clause set N satisfying $\mathcal{BDI}$, condition BDI-2-(iv) can be relaxed to apply only to clauses satisfying $\mathcal{PVD}$. Please note that there are clauses which satisfy the conditions of both BDI-1 and BDI-2.

Consider the following set of clauses as an example to demonstrate and discuss the different syntactical conditions of the class $\mathcal{BDI}$:

(1)		$\rightarrow$	$P(f(a), h(a), a)$
(2)	$P(x, y, z)$	$\rightarrow$	$Q(x, y, f(g(x))),\ S(x, y)$
(3)	$Q(x, y, f(z))$	$\rightarrow$	$R(f(g(x)), x, h(y))$
(4)	$R(f(g(x)), y, h(z))$	$\rightarrow$	$P(x, y, z)$
(5)	$P(a, b, c)$	$\rightarrow$	
(6)	$P(x, y, z)$	$\rightarrow$	$T(y, z)$
(7)	$T(x, y)$	$\rightarrow$	$R(x, y, g(z))$

A common requirement for all clauses is that the set of variables of the succedent of each clause is a subset of the set of variables of the antecedent of the same clause. Clause (7) violates this condition and is therefore not in $\mathcal{BDI}$. For the rest only the clauses (1) to (6) are considered. The ground clauses (1) and (5) trivially satisfy $\mathcal{PVD}$, as well as clause (4). The clause (2) is depth increasing and satisfies BDI-2: The variables occurring in atoms different than $Q(x, y, f(g(x)))$ do not increase the term depth. Further, the predicate P of the atom $P(x, y, z)$ is reachable from Q through the clauses (2)-(3)-(4). P has the same arity than Q, the lists of watched arguments (i.e., all arguments except the depth increasing argument) can be defined identical, and the variable z does not occur inside the third argument of $Q(x, y, f(g(x)))$ (BDI-2-(iii)). Clause (3) satisfies BDI-1 because the occurrence of the atom $R(f(g(x)), x, h(y))$ in clause (3) is similar to the atom $R(f(g(x)), y, h(z))$ in clause (4) (BDI-1-(ii)). Furthermore, the variable x whose depth has been increased in clause (3) occurs with depth 0 in the atom $P(x, y, z)$ in the succedent of clause (4). In addition, the atom in the succedent of clause (3) satisfies $\mathit{vars}(R(f(g(x)), x, h(y))) \subseteq \mathit{vars}(\mathit{warg}(Q(x, y, f(z))))$ (BDI-1-(iv)).

5.3 Termination of Hyper-Resolution on $\mathcal{BDI}$

The Hyper-resolution calculus is used in order to decide $\mathcal{BDI}$. The aim is to show that any derivation from a given finite $\mathcal{BDI}$ clause set N terminates. It is well known that this is the case if the depth of terms in clauses as well as the number of different variables in clauses can be finitely bound. For the new class $\mathcal{BDI}$, Hyper-resolution will only generate ground clauses, implying that for termination it is sufficient to provide an overall depth bound.

Lemma 5.9
Any clause derived by a hyper-resolution inference from an initial clause set N satisfying $\mathcal{BDI}$ is positive ground.

Proof. The proof follows from the variable condition $\mathit{vars}(\Delta) \subseteq \mathit{vars}(\Gamma)$ that holds for all clauses satisfying $\mathcal{BDI}$. We prove by induction over the length of the derivation k with $k \in \mathbb{N}$. Consider the first hyper-resolution step ($k = 1$)

$$\begin{aligned} C = P_1, \ldots, P_n &\rightarrow \Delta_C \\ D_1 = &\rightarrow Q_1, \Delta_{D_1} \\ &\vdots \\ D_n = &\rightarrow Q_n, \Delta_{D_n} \end{aligned}$$

with $C, D_i \in N$. Because of $\mathit{vars}(Q_i \cup \Delta_{D_i}) \subseteq \mathit{vars}(\Gamma_{D_i})$ and $\Gamma_{D_i} = \emptyset$, all D_i are ground. Let $R = \rightarrow \Delta_C\sigma, \Delta_{D_1}, \ldots, \Delta_{D_n}$ be the hyper-resolution resolvent with $P_i\sigma = Q_i$ for all i. Because all D_i are ground, it follows that σ is a ground substitution. Additionally, because $\mathit{vars}(\Delta) \subseteq \mathit{vars}(\Gamma)$ it follows that $\Delta_C\sigma$ is ground. Hence, R is ground, too.

Assume that only positive ground clauses have been produced for k steps. Then this holds also for the $(k + 1)$-th step. Consider the hyper-resolution inference with C, D_i as above. D_i is positive ground, either by Definition 5.8, or because only positive ground clauses have been produced in k steps. The clause $C \in N$ because C is not ground. Again, $R = \rightarrow \Delta_C\sigma, \Delta_{D_1}, \ldots, \Delta_{D_n}$ with $P_i\sigma = Q_i$ for all i. As in the base case, σ is a ground substitution and, with $\mathit{vars}(\Delta) \subseteq \mathit{vars}(\Gamma)$, it follows that R is ground, too.

Thus, all clauses C from N including the clauses that have been derived by hyper-resolution inferences are ground. By construction, positive ground clauses trivially satisfy condition (i) of the class $\mathcal{PVD}$ (Definition 5.1). $\diamondsuit$

Because Factoring is applied only to positive clauses, and positive clauses derived by hyper-resolution inferences are always ground as stated in Lemma 5.9, the application of the Factoring rule corresponds to Condensation which amounts to the elimination of duplicate literals. So for $\mathcal{BDI}$ actually no Factoring rule is needed for completeness.

In order to derive clauses with an increased depth compared to the maximum depth of the initial clause set, a depth increasing clause is needed as stated by the following Lemma 5.10.

Lemma 5.10
Let N be a set of clauses all satisfying the conditions of $\mathcal{PVD}$ and

$$N' = N \cup \{C = \Gamma \to P(t_1, \dots, t_n), \Delta\}$$

be a set of clauses satisfying $\mathcal{BDI}$ where C is a depth increasing clause with some argument t_j and a variable $x \in \mathit{vars}(t_j)$ such that $\mathit{depth}(x, t_j) > \mathit{depth}(x, \Gamma)$ for $1 \leq j \leq n$. Further, let $d_N, d_{N'}$ be the respective initial maximal depth values for the sets N, N'.

Hyper-resolution applied to N' can compute a clause E with $\mathit{depth}(E) > d_{N'}$ only by using the depth increasing clause C.

Proof. Hyper-resolution applied to N only produces smaller clauses with respect to the maximum depth d_N [13]. Consequently, as N' distinguishes from N only in the additional clause C', the only possible resolution candidate to derive a clause with a term depth larger than $d_{N'}$ is C'. Consider a hyper-resolution inference between the clause C' and ground partner clauses $D_i \in N'$ for all atoms $Q_i \in \Gamma$. Then, the maximal depth of the atom $P(t_1, \dots, t_n)\sigma$ in the resolvent E can be approximated as

$$\mathit{depth}(x\sigma, t_j\sigma) = \underbrace{\mathit{depth}(x, t_j)}_{\leq d_{N'}} + \underbrace{\mathit{depth}(x\sigma)}_{\leq d_{N'}} > d_{N'}.$$

$\Diamond$

In the following, termination of the class $\mathcal{BDI}$ is proven by showing that the inference rules Hyper-resolution together with Factoring provide a decision procedure for the class. The proof is carried out by induction over the length of the derivation. Figure 5.1 shows the proof levels used in the induction step in order to assist you understanding the different case distinctions.

Theorem 5.11 (Termination of $\mathcal{BDI}$)
Let N be a finite set of clauses (instance) of the class $\mathcal{BDI}$. Then Hyper-resolution together with Factoring (Factorization) provides a decision procedure for the class $\mathcal{BDI}$ and the depth bound is $d = 2 \cdot \max\{\mathit{depth}(\Delta_C) \mid C \in N\}$.

Proof. All clauses $C \in N$ satisfy $\mathcal{BDI}$ and, therefore, it follows from Lemma 5.9 that Hyper-resolution produces only ground clauses.

We prove by induction over the length of the derivation k with $k \in \mathbb{N}$ and show for all clauses $E \in N^*$ ($N \subseteq N^*$, N^* additionally contains all the derived clauses) that the following invariant holds:

(i) $\mathit{depth}(E) \leq d$

(ii) for all atoms $A(\vec{t}) \in E$ with $\mathit{depth}(A(\vec{t})) > \frac{d}{2}$ holds:

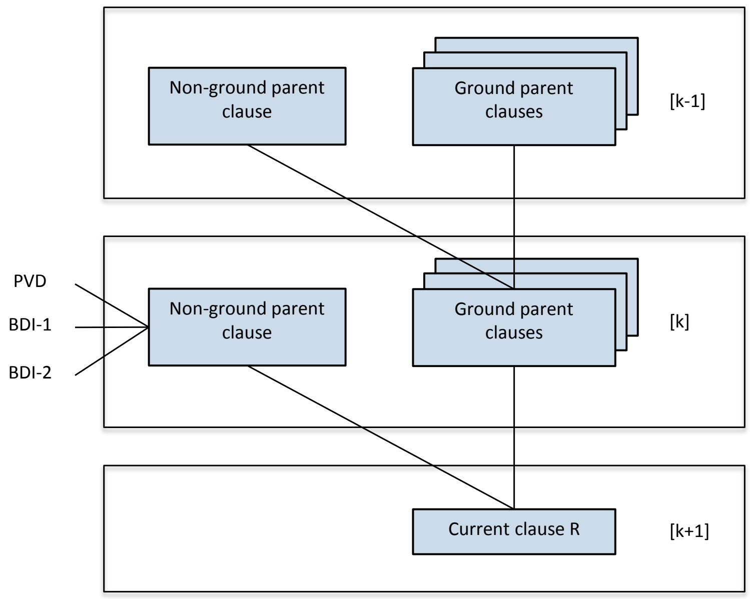


Figure 5.1: Proof levels ($[k+1]$, $[k]$ and $[k-1]$) regarding current, and parent clauses used in the induction step of Theorem 5.11.

(iia) $warg(A(\vec{t})) \neq [\ ]$, A is reachable from a depth increasing clause satisfying BDI-2, and for all arguments $t_p \in warg(A(\vec{t}))$ holds that $depth(t_p) \leq \frac{d}{2}$, or

(iib) $warg(A(\vec{t})) = [\ ]$ and $A(\vec{t})$ originates (and A is therefore also reachable) from a depth increasing clause satisfying BDI-1.

Induction start

We distinguish the cases Hyper-resolution and Factoring to compute inferences.

Case: Hyper-resolution

For Hyper-resolution inferences, we have to consider the 3 different possible clause types BDI-2, BDI-1, and $\mathcal{PVD}$ occurring in N as a (parent) clause C.

Clause C satisfies BDI-2 Let's assume the derivation of inferences starts with a depth increasing non-ground clause C satisfying the conditions of BDI-2 (the other cases BDI-1 and PVD are considered later), such that we have the following situation for the first hyper-resolution step ($k = 1$)

$$\begin{array}{lll} C = \overbrace{Q_1(s_{1,1}, \ldots, s_{1,m_1}), \ldots, Q_n(s_{n,1}, \ldots, s_{n,m_n})}^{\Gamma_C} & \rightarrow & P(t_1, \ldots, t_m), \Delta_C \\ D_1 = & \rightarrow & Q_1(u_{1,1}, \ldots, u_{1,m_1}), \Delta_{D_1} \\ & \vdots & \\ D_n = & \rightarrow & Q_n(u_{n,1}, \ldots, u_{n,m_n}), \Delta_{D_n} \end{array}$$

with the clauses $C, D_i \in N$, $1 \leq i \leq n$, all D_i are ground, and j is the depth increasing argument position of P where $\mathit{depth}(x, t_j) > \mathit{depth}(x, \Gamma_C)$ for some variables $x \in \mathit{vars}(t_j)$. Let

$$E = \rightarrow P(t_1\sigma, \ldots, t_m\sigma), \Delta_C\sigma, \Delta_{D_1}, \ldots, \Delta_{D_n}$$

be the hyper-resolution resolvent of C, D_i with

$$Q_i(s_{i,1}, \ldots, s_{i,m_i})\sigma = Q_i(u_{i,1}, \ldots, u_{i,m_i}).$$

In the following, we prove the properties (i) and (ii) separately for the atoms in Δ_{D_i} (1), $\Delta_C\sigma$ (2) and $P(t_1, \ldots, t_m)\sigma$ (3) of the resolvent E.

(1) For the atoms in Δ_{D_i} holds $\mathit{depth}(\Delta_{D_i}) \leq \frac{d}{2} \leq d$ by definition for all i and therefore they immediately satisfy both (i) and (ii).

(2) The next part are the atoms in Δ_C, where we have for all variables $x \in \mathit{vars}(\Delta_C)$

$$\mathit{depth}(x\sigma, \Delta_C\sigma) = \mathit{depth}(x, \Delta_C) + \mathit{depth}(x\sigma).$$

Because the atoms in Δ_C satisfy $\mathcal{PVD}$ and are therefore not depth increasing, we have

$$\mathit{depth}(x, \Delta_C) \leq \mathit{depth}(x, \Gamma_C)$$

for all $x \in \mathit{vars}(\Delta_C)$. Consequently, we can approximate $\mathit{depth}(x, \Delta_C)$ with $\mathit{depth}(x, \Gamma_C)$ and get

$$\begin{aligned} \mathit{depth}(x\sigma, \Delta_C\sigma) &\leq \mathit{depth}(x, \Gamma_C) + \mathit{depth}(x\sigma) \\ &= \mathit{depth}(x\sigma, \Gamma_C\sigma) \end{aligned}$$

and $\Gamma_C = \bigcup_i Q_i(s_{i,1}, \ldots, s_{i,m_i})$. With

$$Q_i(s_{i,1}, \ldots, s_{i,m_i})\sigma = Q_i(u_{i,1}, \ldots, u_{i,m_i})$$

follows that $\mathit{depth}(Q_i(s_{i,1}, \ldots, s_{i,m_i})\sigma) \leq \frac{d}{2}$ for all i and, therefore,

$$\mathit{depth}(x\sigma, \Delta_C\sigma) \leq \frac{d}{2}.$$

Thus, it holds $\mathit{depth}(\Delta_C\sigma) \leq \frac{d}{2}$ which proves the invariant conditions (i) and (ii) for the atoms in $\Delta_C\sigma$.

(3) The remaining element is the atom $P(t_1,\ldots,t_m)$ with its depth increasing argument position j for which we first prove (i), followed by (ii).
The argument $t_j \in P(t_1,\ldots,t_m)$ is not ground by definition (otherwise, C would not be depth increasing). But it holds that $\mathit{depth}(P(t_1,\ldots,t_m)) \leq \frac{d}{2}$ because $C \in N$, and likewise, for $1 \leq i \leq n$, we have

$$\max\{\mathit{depth}(Q_i(u_{i,1},\ldots,u_{i,m_i}))\} = \max\{\mathit{depth}(u_{i,p_i}) \mid 1 \leq p_i \leq m_i\} \leq \frac{d}{2}$$

where all u_{i,p_i} are ground. With

$$Q_i(s_{i,1},\ldots,s_{i,m_i})\sigma = Q_i(u_{i,1},\ldots,u_{i,m_i})$$

we can approximate the depth of $t_j\sigma$ as follows:

$$\mathit{depth}(t_j\sigma) = \underbrace{\mathit{depth}(x,t_j)}_{\leq\frac{d}{2}} + \underbrace{\max\{\mathit{depth}(u_{i,p_i})\}}_{\leq\frac{d}{2}} \leq d.$$

Consequently, $\mathit{depth}(P(t_1,\ldots,t_m)\sigma) \leq d$ which satisfies (i).
It remains to show (ii) where it is sufficient to prove (iia) because C satisfies BDI-2. We prove (iia) by contradiction and assume $\mathit{depth}(t_i\sigma) > \frac{d}{2}$ for some argument $t_i\sigma \in \mathit{warg}(P(t_1,\ldots,t_m)\sigma)$.
Then $\mathit{arity}(P) > 1$ and there exists a $t_i \neq t_j$, otherwise we would have $\mathit{warg}(P(t_1,\ldots,t_m)\sigma) = [\,]$ according to Definition 5.7 (ii). By the definition of a depth increasing clause, it holds that $\mathit{depth}(x,t_i) \leq \mathit{depth}(x,\Gamma_C)$ for all $i \neq j$. Thus, it follows with $\mathit{depth}(Q_i(u_{i,1},\ldots,u_{i,m_i})) \leq \frac{d}{2}$ and

$$Q_i(s_{i,1},\ldots,s_{i,m_i})\sigma = Q_i(u_{i,1},\ldots,u_{i,m_i})$$

analogously to the part $\Delta_C\sigma$ that $\mathit{depth}(t_i\sigma) \leq \frac{d}{2}$ for all $i \neq j$. This contradicts our assumption and therefore proves (iia).

Clause C satisfies BDI-1 If we start with a depth increasing clause C satisfying BDI-1, we have the following situation:

$$\begin{aligned}
C = &\overbrace{Q_1(s_{1,1},\ldots,s_{1,m_1}),\ldots,Q_n(s_{n,1},\ldots,s_{n,m_n})}^{\Gamma_C} \rightarrow \\
&\qquad\qquad P_1(t_{1,1},\ldots,t_{1,k_1}),\ldots,P_l(t_{l,1},\ldots,t_{l,k_l}),\Delta_C \\
D_i = &\qquad\qquad\qquad\qquad\qquad \rightarrow Q_i(u_{i,1},\ldots,u_{i,m_i}),\Delta_{D_i}
\end{aligned}$$

for all $1 \leq i \leq n$. All D_i are ground and therefore satisfy PVD. Let

$$E = \rightarrow P_1(t_{1,1},\ldots,t_{1,k_1})\sigma,\ldots,P_l(t_{l,1},\ldots,t_{l,k_l})\sigma,\Delta_C\sigma,\Delta_{D_1},\ldots,\Delta_{D_n}$$

be the resolvent of C, D_i with $Q_i(s_{i,1},\ldots,s_{i,m_i})\sigma = Q_i(u_{i,1},\ldots,u_{i,m_i})$. We again prove the conditions (i) and (ii) separately for the atoms in Δ_{D_i} (1), $\Delta_C\sigma$ (2), and the remaining atoms (3) in the part of the resolvent E.

(1) We start with the atoms in Δ_{D_i} which satisfy the invariant conditions by definition analogously as in the previous case BDI-2.

(2) The next part are the atoms in $\Delta_C\sigma$ where we have

$$\mathit{depth}(x, A) \leq \mathit{depth}(x, \Gamma_C)$$

for all non-depth increasing atoms $A \in \Delta_C$. Therefore, it follows together with

$$Q_i(s_{i,1}, \ldots, s_{i,m_i})\sigma = Q_i(u_{i,1}, \ldots, u_{i,m_i})$$

analogously as for $\Delta_C\sigma$ in the previous case BDI-2 that $\mathit{depth}(A\sigma) \leq \frac{d}{2}$ for all $A\sigma \in \Delta_C\sigma$ which satisfies the invariant conditions (i) and (ii) for the atoms in $\Delta_C\sigma$.

(3) The remaining elements are the atoms $P_o(t_{o,1}, \ldots, t_{o,k_o})$ with $1 \leq o \leq l$ where it exists one argument t_{o,p_j} in each atom where

$$x \in (\mathit{vars}(t_{o,p_j}) \cap \mathit{vars}(\Gamma_C))$$

and $\mathit{depth}(x, t_{o,p_j}) > \mathit{depth}(x, \Gamma_C)$. We first prove (i), followed by (ii).

Because we are in the first step (induction start), it holds

$$\mathit{depth}(Q_i(u_{i,1}, \ldots, u_{i,m_i})) \leq \frac{d}{2}$$

for all i. Using $Q_i(s_{i,1}, \ldots, s_{i,m_i})\sigma = Q_i(u_{i,1}, \ldots, u_{i,m_i})$ we can calculate for all variables $x \in \mathit{vars}(P_o(t_{o,1}, \ldots, t_{o,k_o}))$ in the respective atom ($1 \leq o \leq l$):

$$\mathit{depth}(x\sigma, P_o(t_{o,1}, \ldots, t_{o,k_o})\sigma) \leq \underbrace{\mathit{depth}(x, P_o(t_{o,1}, \ldots, t_{o,k_o}))}_{\frac{d}{2}} +$$
$$\underbrace{\max\{\mathit{depth}(Q_i(u_{i,1}, \ldots, u_{i,m_i}))\}}_{\leq \frac{d}{2}} \leq d.$$

This proves (i) for $P_o(t_{o,1}, \ldots, t_{o,k_o})$.

For (ii) it is sufficient to prove the case (iib) because C satisfies BDI-1. Fortunately, property (iib) is immediately satisfied because of Definition 5.6 (iii), stating that $\mathit{warg}(P_o(t_{o,1}, \ldots, t_{o,k_o})) = [\,]$. Moreover, if

$$\frac{d}{2} < \mathit{depth}(P_o(t_{o,1}, \ldots, t_{o,k_o})\sigma) \leq d,$$

$P_o(t_{o,1}, \ldots, t_{o,k_o})\sigma$ obviously originates from C which is a depth increasing clause satisfying BDI-1 and therefore satisfies (iib).

Clause C satisfies $\mathcal{PVD}$ If the clause C is not depth increasing then it satisfies $\mathcal{PVD}$. The partner clauses D_i are all positive ground and therefore satisfy $\mathcal{PVD}$, too. Consequently, $\mathit{depth}(C) \leq \frac{d}{2}$, and $\mathit{depth}(D_i) \leq \frac{d}{2}$ for all $1 \leq i \leq n$. Fermüller et al. [12] have already shown that $\mathit{depth}(E) \leq \frac{d}{2} < d$ for the resolvent E in that case which satisfies both invariant conditions (i) and (ii).

Case: Factoring
Because Factoring is applied only to positive clauses, and positive clauses derived by hyper-resolution inferences are always ground as stated in Lemma 5.9, the application of Factoring corresponds to Condensation which amounts to the elimination of duplicate literals and thus producing strictly smaller clauses because the resolvent is a strict subset of its parent clause.

Induction step

Assume that the invariant holds for all derivations of length k. We are now going to execute the $(k+1)$-th step of a derivation. We again distinguish the cases Hyper-resolution and Factoring to compute inferences.

Case: Hyper-resolution
We compute a resolvent clause E via Hyper-resolution from clauses $C \in N$ and $D_i \in N^k$ (the latter from level $[k]$) and show that the invariant conditions (i) and (ii) still hold for E.

All clauses D_i are ground by definition ($D_i \in N$ and then $\mathit{vars}(\Delta) \subseteq \mathit{vars}(\Gamma)$) or because they have been derived by hyper-resolution inference steps (Lemma 5.9). The clause $C \in N$ satisfies the conditions of $\mathcal{BDI}$ by definition and therefore, we distinguish the following three cases of C (in level $[k]$, see Figure 5.1):

Clause C satisfies $\mathcal{PVD}$ Let

$$\begin{aligned} C &= Q_1(s_{1,1}, \ldots, s_{1,m_1}), \ldots, Q_n(s_{n,1}, \ldots, s_{n,m_n}) \rightarrow \Delta_C \\ D_i &= \rightarrow Q_i(u_{i,1}, \ldots, u_{i,m_i}), \Delta_{D_i} \end{aligned}$$

for all $1 \leq i \leq n$ where D_i are all ground and therefore satisfy $\mathcal{PVD}$. Further, let

$$E = \rightarrow \Delta_C\sigma, \Delta_{D_1}, \ldots, \Delta_{D_n}$$

be the resolvent of C, D_i with $Q_i(s_{i,1}, \ldots, s_{i,m_i})\sigma = Q_i(u_{i,1}, \ldots, u_{i,m_i})$.

We first show (i) for E. Analogously to the base case (all D_i are ground and C and D_i all satisfy $\mathcal{PVD}$), it holds that $\mathit{depth}(E) \leq d$ [12] which proves (i) for E.

Next, we show (ii) separately for the atoms in the parts Δ_{D_i} (1) and $\Delta_C\sigma$ (2) of the resolvent E.

(1) Consider an atom $A(r_1, \ldots, r_q) \in \Delta_{D_i}$ such that

$$\mathit{depth}(A(r_1, \ldots, r_q)) > \frac{d}{2}.$$

Because $\Delta_{D_i} \in D_i$ and D_i is a clause from level $[k]$, the induction hypothesis holds for D_i. Additionally, because of

$$A(r_1, \ldots, r_q) \in D_i = A(r_1, \ldots, r_q) \in E,$$

we distinguish the two cases of (ii) for $A(r_1, \ldots, r_q) \in D_i$:

- In the first case (iia), we have $\mathit{warg}(A(r_1, \ldots, r_q)) \neq [\ ]$, A is reachable from a depth increasing clause satisfying BDI-2, and for all arguments $r_p \in \mathit{warg}(A(r_1, \ldots, r_q))$ holds $\mathit{depth}(r_p) \leq \frac{d}{2}$. Because of the construction of the watched arguments function and because $A(r_1, \ldots, r_q)$ is ground, (iia) also holds for $A(r_1, \ldots, r_q) \in E$.

- The other case is (iib) where we have $\mathit{warg}(A(r_1, \ldots, r_q)) = [\ ]$ and $A(r_1, \ldots, r_q)$ originates from a depth increasing clause satisfying BDI-1. Because of the construction of origination and because $A(r_1, \ldots, r_q)$ is ground, $A(r_1, \ldots, r_q) \in E$ still originates from that depth increasing clause satisfying BDI-1 and it holds $\mathit{warg}(A(r_1, \ldots, r_q)) = [\ ]$.

This proves (ii) for the atoms in the part $\Delta_{D_i} \in E$.

(2) Consider a derived atom $P(t_1, \ldots, t_r)\sigma \in \Delta_C\sigma$ where

$$\mathit{depth}(P(t_1, \ldots, t_r)\sigma) > \frac{d}{2},$$

and for which we have to prove that condition (ii), i.e., (iia) or (iib) hold(s). To obtain such an atom, we need a partner clause D_i (from level $[k]$) with an atom $Q_i(u_{i,1}, \ldots, u_{i,m_i})$,

$$\mathit{vars}(Q_i(s_{i,1}, \ldots, s_{i,m_i})) \cap \mathit{vars}(P(t_1, \ldots, t_r)) \neq \emptyset,$$

and $\mathit{depth}(Q_i(u_{i,1}, \ldots, u_{i,m_i})) > \frac{d}{2}$ because C satisfies $\mathcal{PVD}$ and D_i is ground. But for D_i holds the induction hypothesis, and therefore we have:

([k].iia) $\mathit{warg}(Q_i(u_{i,1}, \ldots, u_{i,m_i})) \neq [\ ]$, Q_i is reachable from a depth increasing clause satisfying BDI-2, and for all arguments $u_{i,p_i} \in \mathit{warg}(Q_i(u_{i,1}, \ldots, u_{i,m_i}))$ holds $\mathit{depth}(u_{i,p_i}) \leq \frac{d}{2}$, or

([k].iib) $\mathit{warg}(Q_i(u_{i,1}, \ldots, u_{i,m_i})) = [\ ]$ and $Q_i(u_{i,1}, \ldots, u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1.

In case ([k].iia), Q_i is reachable from a depth increasing clause satisfying BDI-2 and so is also P because P is reachable from Q_i by clause C. Because of $\mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i})) \neq [\,]$, it holds also

$$\mathit{warg}(Q_i(s_{i,1},\ldots,s_{i,m_i})) \neq [\,].$$

But then, it follows from Definition 5.7 (iv) that

$$\mathit{warg}(P(t_1,\ldots,t_r)) = \mathit{warg}(Q_i(s_{i,1},\ldots,s_{i,m_i}))$$

and with $Q_i(s_{i,1},\ldots,s_{i,m_i})\sigma = Q_i(u_{i,1},\ldots,u_{i,m_i})$, we obtain

$$\mathit{warg}(P(t_1,\ldots,t_r)\sigma) = \mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i})).$$

Consequently, it follows from the induction hypothesis that also

$$\mathit{depth}(t_p\sigma) \leq \frac{d}{2} \text{ for all arguments } t_p\sigma \in \mathit{warg}(P(t_1,\ldots,t_r)\sigma).$$

This satisfies the invariant condition (iia) and therefore completes the case ([k].iia).

We continue with the case ([k].iib) and prove it by contradiction: It cannot be for an atom $P(t_1,\ldots,t_r)\sigma \in \Delta_C\sigma$ that $\mathit{depth}(P(t_1,\ldots,t_r)\sigma) > \frac{d}{2}$ and $P(t_1,\ldots,t_r)\sigma$ originates from a depth increasing clause satisfying BDI-1. Thus, we assume for the atom $Q_i(u_{i,1},\ldots,u_{i,m_i})$ that

$$\mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i})) = [\,]$$

and $Q_i(u_{i,1},\ldots,u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1. Let

$$C' = \Gamma_{C'} \rightarrow Q_i(q_{i,1},\ldots,q_{i,m_i}), \Delta_{C'}$$

be this clause (from level [k]) with

$$Q_i(q_{i,1},\ldots,q_{i,m_i})\sigma' = Q_i(u_{i,1},\ldots,u_{i,m_i}).$$

Then it follows from Definition 5.6 (ii) that the atoms $Q_i(q_{i,1},\ldots,q_{i,m_i})$ and $Q_i(s_{i,1},\ldots,s_{i,m_i})$ are similar, and for all

$$y \in \mathit{vars}(Q_i(s_{i,1},\ldots,s_{i,m_i})) \cap \mathit{vars}(P(t_1,\ldots,t_r))$$

where $s_{i,p_i}|_p = y$, $q_{i,p_i}|_p = x'$, and $\mathit{depth}(x', q_{i,p_i}) > \mathit{depth}(x', \Gamma_{C'})$ holds $\mathit{depth}(y, P(t_1,\ldots,t_r)) = 0$. We have $\mathit{depth}(C') \leq \frac{d}{2}$ by definition, and $\mathit{depth}(D_i) \leq d$ according to the induction hypothesis.

We continue by showing that $\mathit{depth}(x'\sigma') \leq \frac{d}{2}$ and prove it also by contradiction. Let's assume that $\mathit{depth}(x'\sigma') > \frac{d}{2}$. This implies that there is an atom $A_l(s_{l,1}, \ldots, s_{l,m_l}) \in \Gamma_{C'}$ such that $x' \in \mathit{vars}(A_l(s_{l,1}, \ldots, s_{l,m_l}))$, and $\mathit{depth}(A_l(s_{l,1}, \ldots, s_{l,m_l})\sigma') > \frac{d}{2}$. Consequently, there is a clause

$$C'' = \Gamma_{C''} \to A_l(q_{l,1}, \ldots, q_{l,m_l}), \Delta_{C''}$$

(from level $[k-1]$) such that the atom $A_l(u_{l,1}, \ldots, u_{l,m_l})$ originates from C'' and

$$A_l(q_{l,1}, \ldots, q_{l,m_l})\sigma'' = A_l(u_{l,1}, \ldots, u_{l,m_l}).$$

Additionally, we have $A_l(u_{l,1}, \ldots, u_{l,m_l}) = A_l(s_{l,1}, \ldots, s_{l,m_l})\sigma'$.

But then the two conditions of (ii) again hold for $A_l(u_{l,1}, \ldots, u_{l,m_l}) \in C''\sigma''$ by induction hypothesis:

($[k-1]$.iia) $\mathit{warg}(A_l(u_{l,1}, \ldots, u_{l,m_l})) \neq [\ ]$ and A_l is reachable from a depth increasing clause satisfying BDI-2, and for all arguments $u_{l,p_l} \in \mathit{warg}(A_l(u_{l,1}, \ldots, u_{l,m_l}))$ holds $\mathit{depth}(u_{l,p_l}) \leq \frac{d}{2}$, or

($[k-1]$.iib) $\mathit{warg}(A_l(u_{l,1}, \ldots, u_{l,m_l})) = [\,]$ and $A_l(u_{l,1}, \ldots, u_{l,m_l})$ originates from a depth increasing clause satisfying BDI-1.

In the first case ($[k-1]$.iia), A_l is reachable from a depth increasing clause. Because C' satisfies BDI-1, it must hold according to Definition 5.7 (iv) that

$$\begin{aligned}\mathit{vars}(Q_i(q_{i,1}, \ldots, q_{i,m_i})) \subseteq & \bigcup_v \mathit{vars}(\mathit{warg}(A_v(s_{v,1}, \ldots, s_{v,m_v}))) \cup \\ & \bigcup_w \mathit{vars}(A_w(s_{w,1}, \ldots, s_{w,m_w})))\end{aligned}$$

where $A_v(s_{v,1}, \ldots, s_{v,m_v}) \in \Gamma_{C'}$ are the atoms that are reachable from a depth increasing clause, and $A_w(s_{w,1}, \ldots, s_{w,m_w}) \in \Gamma_{C'}$ are the atoms that are not reachable from a depth increasing clause at all. From the induction hypothesis follows that

$$\mathit{depth}(u_{l,p_l l}) \leq \frac{d}{2} \text{ for all arguments } u_{l,p_l} \in \mathit{warg}(A_l(u_{l,1}, \ldots, u_{l,m_l}))).$$

Additionally, it holds also $\mathit{depth}(A_w(s_{w,1}, \ldots, s_{w,m_w})\sigma'') \leq \frac{d}{2}$ according to Lemma 5.10 because A_w is not reachable from a depth increasing clause at all. Consequently, $\mathit{depth}(x'\sigma') \leq \frac{d}{2}$ which contradicts our assumption.

In the second case ($[k-1]$.iib), $A_l(u_{l,1}, \ldots, u_{l,m_l})$ is reachable from a depth increasing clause satisfying BDI-1 (it actually originates from it), and it holds that

$$\mathit{warg}(A_l(s_{l,1}, \ldots, s_{l,m_l})\sigma') = [\,].$$

Consequently, we get from Definition 5.6 (iv) that it must hold

$$\mathit{vars}(Q_i(q_{i,1}, \ldots, q_{i,m_i})) \cap \mathit{vars}(A_l(s_{l,1}, \ldots, s_{l,m_l}) = \emptyset.$$

But this contradicts

$$x' \in \mathit{vars}(A_l(s_{l,1}, \dots, s_{l,m_l}) \text{ and } x' \in Q_i(q_{i,1}, \dots, q_{i,m_i}).$$

Hence, it must hold $\mathit{depth}(x'\sigma') \leq \frac{d}{2}$.

Now, we proceed again with our case ([k].iib). The variables

$$y \in \mathit{vars}(Q_i(s_{i,1}, \dots, s_{i,m_i})) \cap \mathit{vars}(P(t_1, \dots, t_r))$$

are the only possible candidates to obtain $\mathit{depth}(P(t_1, \dots, t_r)\sigma) > \frac{d}{2}$ (remember that $\mathit{depth}(x, \Delta_C) \leq \mathit{depth}(x, \Gamma_C)$ for all $x \in \mathit{vars}(\Delta_C)$). However, from the previous intermediate result $\mathit{depth}(x'\sigma') \leq \frac{d}{2}$ follows that

$$\mathit{depth}(y\sigma) \leq \frac{d}{2}.$$

This finishes the case ([k].iib) because of the fact that no atom $P(t_1, \dots, t_r)\sigma \in \Delta_C\sigma$ exists such that $\mathit{depth}(P(t_1, \dots, t_r)\sigma) > \frac{d}{2}$ and $P(t_1, \dots, t_r)\sigma$ originates from a depth increasing clause satisfying BDI-1 which contradicts our assumption.

This completes the proof of (ii) for the atoms in the part $\Delta_C\sigma$.

Clause C satisfies BDI-1 Let

$$\begin{aligned} C = \;& \overbrace{Q_1(s_{1,1}, \dots, s_{1,m_1}), \dots, Q_n(s_{n,1}, \dots, s_{n,m_n})}^{\Gamma_C} \;\rightarrow \\ & \qquad\qquad P_1(t_{1,1}, \dots, t_{1,k_1}), \dots, P_l(t_{l,1}, \dots, t_{l,k_l}), \Delta_C \\ D_i = \;& \qquad\qquad\qquad\qquad \rightarrow\; Q_i(u_{i,1}, \dots, u_{i,m_i}), \Delta_{D_i} \end{aligned}$$

for all $1 \leq i \leq n$ where C is a depth increasing clause and for all $P_o(t_{o,1}, \dots, t_{o,k_o})$ with $1 \leq o \leq l$ exists exactly one argument t_{o,p_j} where

$$x \in (\mathit{vars}(t_{o,p_j})) \cap \mathit{vars}(\Gamma_C))$$

and

$$\mathit{depth}(x, t_{o,p_j}) > \mathit{depth}(x, \Gamma_C).$$

All partner clauses D_i are positive ground and satisfy $\mathcal{PVD}$. Let

$$E =\rightarrow\; P_1(t_{1,1}, \dots, t_{1,k_1})\sigma, \dots, P_l(t_{l,1}, \dots, t_{l,k_l})\sigma, \Delta_C\sigma, \Delta_{D_1}, \dots, \Delta_{D_n}$$

be the resolvent of C, D_i with $Q_i(s_{i,1}, \dots, s_{i,m_i})\sigma = Q_i(u_{i,1}, \dots, u_{i,m_i})$. We prove the invariant property (i) and (ii) separately for the atoms in Δ_{D_i} (1), $\Delta_C\sigma$ (2), and $P_o(t_{o,1}, \dots, t_{o,k_o})\sigma$ (3) of the resolvent E.

(1) We start with the atoms in Δ_{D_i} which are ground and therefore satisfy the invariant condition (i) and (ii) by induction hypothesis analogously as in the case where C satisfies $\mathcal{PVD}$.

(2) The next part are the atoms in $\Delta_C\sigma$, for which we first prove (i). For all variables $x \in \mathit{vars}(\Delta_C)$ holds:

$$\mathit{depth}(x\sigma, \Delta_C\sigma) = \mathit{depth}(x, \Delta_C) + \mathit{depth}(x\sigma).$$

Because Δ_C is not depth increasing, it holds for all variables $x \in (\mathit{vars}(\Delta_C) \cap \mathit{vars}(\Gamma_C))$ that

$$\mathit{depth}(x, \Delta_C) \leq \mathit{depth}(x, \Gamma_C).$$

Consequently, we can replace $\mathit{depth}(x, \Delta_C)$ with $\mathit{depth}(x, \Gamma_C)$ and get

$$\begin{aligned}\mathit{depth}(x\sigma, \Delta_C\sigma) &\leq \mathit{depth}(x, \Gamma_C) + \mathit{depth}(x\sigma)\\ &= \mathit{depth}(x\sigma, \Gamma_C\sigma)\end{aligned}$$

and $\Gamma_C = \bigcup_i Q_i(s_{i,1}, \ldots, s_{i,m_i})$. With

$$Q_i(s_{i,1}, \ldots, s_{i,m_i})\sigma = Q_i(u_{i,1}, \ldots, u_{i,m_i})$$

follows that $\mathit{depth}(Q_i(s_{i,1}, \ldots, s_{i,m_i})\sigma) \leq d$ for all i and, therefore,

$$\mathit{depth}(x\sigma, \Delta_C\sigma) \leq d.$$

Thus, it holds $\mathit{depth}(\Delta_C\sigma) \leq d$ which proves (i) for the atoms in $\Delta_C\sigma$.

It remains to prove (ii) for the atoms in $\Delta_C\sigma$. We consider a derived atom $A(t_1, \ldots, t_r)\sigma \in \Delta_C\sigma$ where

$$\mathit{depth}(A(t_1, \ldots, t_r)\sigma) > \frac{d}{2}.$$

and show that the invariant condition (iia) or (iib) holds for $A(t_1, \ldots, t_r)\sigma$. To obtain such an atom, we need a partner clause

$$D_i = \rightarrow Q_i(u_{i,1}, \ldots, u_{i,m_i}), \Delta_{D_i}$$

(from level $[k]$) with $\mathit{depth}(Q_i(u_{i,1}, \ldots, u_{i,m_i})) > \frac{d}{2}$ and

$$\mathit{vars}(Q_i(s_{i,1}, \ldots, s_{i,m_i})) \cap \mathit{vars}(A(t_1, \ldots, t_r)) \neq \emptyset,$$

because $\mathit{depth}(C) \leq \frac{d}{2}$, and Δ_C is not depth increasing. But for D_i holds the induction hypothesis, from which it follows that

([k].iia) $\mathit{warg}(Q_i(u_{i,1}, \ldots, u_{i,m_i})) \neq [\ ]$, Q_i is reachable from a depth increasing clause satisfying BDI-2, and for all arguments $u_{i,p_i} \in \mathit{warg}(Q_i(u_{i,1}, \ldots, u_{i,m_i}))$ holds $\mathit{depth}(u_{i,p_i}) \leq \frac{d}{2}$, or

([k].iib) $\mathit{warg}(Q_i(u_{i,1}, \ldots, u_{i,m_i})) = [\ ]$ and $Q_i(u_{i,1}, \ldots, u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1.

In case ([k].iia), Q_i is reachable from a depth increasing clause satisfying BDI-2 and so is also A because A is reachable from Q_i with the clause C. Because of $\mathit{warg}(Q_i(u_{i,1},\dots,u_{i,m_i})) \neq [\,]$, it holds also

$$\mathit{warg}(Q_i(s_{i,1},\dots,s_{i,m_i})) \neq [\,].$$

But then, it follows from Definition 5.6 (v) that

$$\mathit{warg}(A(t_1,\dots,t_r)) = \mathit{warg}(Q_i(s_{i,1},\dots,s_{i,m_i}))$$

and with $Q_i(s_{i,1},\dots,s_{i,m_i})\sigma = Q_i(u_{i,1},\dots,u_{i,m_i})$ we obtain

$$\mathit{warg}(A(t_1,\dots,t_r)\sigma) = \mathit{warg}(Q_i(u_{i,1},\dots,u_{i,m_i})).$$

Consequently, it follows from the induction hypothesis that also

$$\mathit{depth}(t_p\sigma) \leq \frac{d}{2} \text{ for all arguments } t_p\sigma \in \mathit{warg}(A(t_1,\dots,t_r)\sigma).$$

This satisfies the invariant condition (iia) and therefore completes the case ([k].iia).

We continue with the case ([k].iib) and prove it by contradiction: It cannot be for an atom $A(t_1,\dots,t_r)\sigma \in \Delta_C\sigma$ that $\mathit{depth}(A(t_1,\dots,t_r)\sigma) > \frac{d}{2}$ and $A(t_1,\dots,t_r)\sigma$ originates from a depth increasing clause satisfying BDI-1. Thus, we assume for the atom $Q_i(u_{i,1},\dots,u_{i,m_i})$ that

$$\mathit{warg}(Q_i(u_{i,1},\dots,u_{i,m_i})) = [\,]$$

and $Q_i(u_{i,1},\dots,u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1. Let

$$C' = \Gamma_{C'} \rightarrow Q_i(q_{i,1},\dots,q_{i,m_i}), \Delta_{C'}$$

be this clause (from level [k]) with

$$Q_i(q_{i,1},\dots,q_{i,m_i})\sigma' = Q_i(u_{i,1},\dots,u_{i,m_i}).$$

Then it follows from Definition 5.6 (ii) that the atoms $Q_i(q_{i,1},\dots,q_{i,m_i})$ and $Q_i(s_{i,1},\dots,s_{i,m_i})$ are similar, and for all

$$y \in \mathit{vars}(Q_i(s_{i,1},\dots,s_{i,m_i})) \cap \mathit{vars}(A(t_1,\dots,t_r))$$

where $s_{i,p_i}|_p = y$, $q_{i,p_i}|_p = x'$, and $\mathit{depth}(x', q_{i,p_i}) > \mathit{depth}(x', \Gamma_{C'})$ holds $\mathit{depth}(y, A(t_1,\dots,t_r)) = 0$. With $\mathit{depth}(C') \leq \frac{d}{2}$, $\mathit{depth}(D_i) \leq d$, and $\mathit{depth}(x'\sigma') \leq \frac{d}{2}$ which follows analogously as in the case for PVD, we obtain that

$$\mathit{depth}(y\sigma) \leq \frac{d}{2}$$

because

$$Q_i(s_{i,1},\ldots,s_{i,m_i})\sigma = Q_i(u_{i,1},\ldots,u_{i,m_i}) = Q_i(q_{i,1},\ldots,q_{i,m_i})\sigma'$$

where the variables $y \in \mathit{vars}(Q_i(s_{i,1},\ldots,s_{i,m_i})) \cap \mathit{vars}(A(t_1,\ldots,t_r))$ were the only possible candidates to obtain $\mathit{depth}(A(t_1,\ldots,t_r)\sigma) > \frac{d}{2}$ at all (remember that $\mathit{depth}(x,\Delta_C) \leq \mathit{depth}(x,\Gamma_C)$ because Δ_C is not depth increasing). These candidates have now been proven not to be depth increasing, and the case ([k].iib) is finished because there is no atom $A(t_1,\ldots,t_r)\sigma \in \Delta_C\sigma$ such that $\mathit{depth}(A(t_1,\ldots,t_r)\sigma) > \frac{d}{2}$ and $A(t_1,\ldots,t_r)\sigma$ originates from a depth increasing clause satisfying BDI-1 which contradicts our assumption.
This completes the proof of (ii) for the part $\Delta_C\sigma$.

(3) The remaining elements are the atoms $P_o(t_{o,1},\ldots,t_{o,k_o})$ with $1 \leq o \leq l$ where it exists exactly one argument t_{o,p_j} in each atom where

$$x \in (\mathit{vars}(t_{o,p_j})) \cap \mathit{vars}(\Gamma_C))$$

and

$$\mathit{depth}(x,t_{o,p_j}) > \mathit{depth}(x,\Gamma_C).$$

We first prove the invariant condition (i) by contradiction: Assume for any atom $P_o(t_{o,1},\ldots,t_{o,k_o})$ that we have derived $P_o(t_{o,1},\ldots,t_{o,k_o})\sigma$ having an argument $t_{o,p_o}\sigma$ with $\mathit{depth}(t_{o,p_o}\sigma) > d$. Because of $\mathit{depth}(C) \leq \frac{d}{2}$ and

$$Q_i(s_{i,1},\ldots,s_{i,m_i})\sigma = Q_i(u_{i,1},\ldots,u_{i,m_i}),$$

there must be a partner clause $D_i = \rightarrow Q_i(u_{i,1},\ldots,u_{i,m_i}), \Delta_{D_i}$ (from level [k]) for which it holds $\mathit{depth}(Q_i(u_{i,1},\ldots,u_{i,m_i})) > \frac{d}{2}$. But then, it follows from the induction hypothesis, that

([k].iia) $\mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i})) \neq [\ ]$, Q_i is reachable from a depth increasing clause satisfying BDI-2, and for all arguments $u_{i,p_i} \in \mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i}))$ holds $\mathit{depth}(u_{i,p_i}) \leq \frac{d}{2}$, or

([k].iib) $\mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i})) = [\ ]$ and $Q_i(u_{i,1},\ldots,u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1.

In case ([k].iia), Q_i is reachable from a depth increasing clause satisfying BDI-2 and so is also P_o because P_o is reachable from Q_i with the clause C. From the induction hypothesis follows that $\mathit{depth}(u_{i,p_i}) \leq \frac{d}{2}$ for all $u_{i,p_i} \in \mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i}))$. Additionally, it holds according to Definition 5.6 (iv) that

$$\mathit{vars}(P_o(t_{o,1},\ldots,t_{o,k_o})) \subseteq \bigcup_v \mathit{vars}(\mathit{warg}(Q_v(s_{v,1},\ldots,s_{v,m_v}))) \cup \bigcup_w \mathit{vars}(Q_w(s_{w,1},\ldots,s_{w,m_w}))$$

where $Q_v(s_{v,1},\ldots,s_{v,m_v})$ are the atoms in Γ_C that are reachable from a depth increasing clause, and $Q_w(s_{w,1},\ldots,s_{w,m_w})$ are the atoms in Γ_C that are not reachable from a depth increasing clause at all. Therefore, we have

$$\mathit{depth}(Q_w(u_{w,1},\ldots,u_{w,m_w})) \leq \frac{d}{2}$$

according to Lemma 5.10 and because of the induction hypothesis, it holds

$$\mathit{depth}(\mathit{warg}(Q_v(u_{v,1},\ldots,u_{v,m_v}))) \leq \frac{d}{2}.$$

Hence, we can calculate for all $x \in \mathit{vars}(P_o(t_{o,1},\ldots,t_{o,k_o}))$:

$$\mathit{depth}(x\sigma, P_o(t_{o,1},\ldots,t_{o,k_o})\sigma) = \underbrace{\mathit{depth}(x, P_o(t_{o,1},\ldots,t_{o,k_o})}_{\leq\frac{d}{2}} +$$

$$\underbrace{\max\{\mathit{depth}(\mathit{warg}(Q_v(u_{v,1},\ldots,u_{v,m_v}))), \mathit{depth}(Q_w(u_{w,1},\ldots,u_{w,m_w}))\}}_{\leq\frac{d}{2}} \leq d.$$

In case ([k].iib), we have $\mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i})) = [\ ]$ and the atom $Q_i(u_{i,1},\ldots,u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1. Because C satisfies BDI-1, it holds according to Definition 5.6 (iv) that

$$\begin{aligned}\mathit{vars}(P_o(t_{o,1},\ldots,t_{o,k_o})) \subseteq & \bigcup_v \mathit{vars}(\mathit{warg}(Q_v(s_{v,1},\ldots,s_{v,m_v}))) \cup \\ & \bigcup_w \mathit{vars}(Q_w(s_{w,1},\ldots,s_{w,m_w}))\end{aligned}$$

where $Q_v(s_{v,1},\ldots,s_{v,m_v})$ are the atoms that are reachable from a depth increasing clause, and $Q_w(s_{w,1},\ldots,s_{w,m_w})$ are the atoms that are not reachable from a depth increasing clause at all. However, $Q_i(u_{i,1},\ldots,u_{i,m_i})$ is reachable from a depth increasing clause satisfying BDI-1 (it actually originates from it), and therefore $\mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i})) = [\,]$. Consequently, we get

$$\mathit{vars}(P_o(t_{o,1},\ldots,t_{o,k_o})) \cap \mathit{vars}(Q_i(s_{i,1},\ldots,s_{i,m_i})) = \emptyset.$$

But this is contradicting because we have assumed to derive an atom $P_o(t_{o,1},\ldots,t_{o,k_o})\sigma \in E$ having an argument $t_{o,p_o}\sigma$ with $\mathit{depth}(t_{o,p_o}\sigma) > d$, which means on the other hand that we need an atom $Q_i(u_{i,1},\ldots,u_{i,m_i})$ with $\mathit{depth}(Q_i(u_{i,1},\ldots,u_{i,m_i})) > \frac{d}{2}$, and

$$\mathit{vars}(P_o(t_{o,1},\ldots,t_{o,k_o})) \cap \mathit{vars}(Q_i(s_{i,1},\ldots,s_{i,m_i})) \neq \emptyset.$$

This finishes the case ([k].iib) and therefore proves the invariant condition (i) for the atom $P_o(t_{o,1},\ldots,t_{o,k_o})\sigma$.

Showing the invariant condition (ii) is easy because C satisfies BDI-1, and it holds according to Definition 5.6 (iii) that

$$warg(P_o(t_{o,1},\dots,t_{o,k_o}) = [\,].$$

Additionally, the atom $P_o(t_{o,1},\dots,t_{o,k_o})\sigma$ obviously originates from a depth incrasing clause C satisfying BDI-1 which proves (ii).

Clause C satisfies BDI-2 Let

$$\begin{aligned} C &= \overbrace{Q_1(s_{1,1},\dots,s_{1,m_1}),\dots,Q_n(s_{n,1},\dots,s_{n,m_n})}^{\Gamma_C} \rightarrow P(t_1,\dots,t_m),\Delta_C \\ D_i &= \phantom{Q_1(s_{1,1},\dots,s_{1,m_1}),\dots,Q_n(s_{n,1},\dots,s_{n,m_n})} \rightarrow Q_i(u_{i,1},\dots,u_{i,m_i}),\Delta_{D_i} \end{aligned}$$

where C is a depth increasing clause and $depth(x,t_j) > depth(x,\Gamma_C)$ for some variables $x \in vars(t_j)$. All partner clauses D_i are positive ground and satisfy $\mathcal{PVD}$. Let

$$E = \rightarrow P(t_1\sigma,\dots,t_m\sigma),\Delta_C\sigma,\Delta_{D_1},\dots,\Delta_{D_n}$$

be the resolvent of C, D_i with $Q_i(s_{i,1},\dots,s_{i,m_i})\sigma = Q_i(u_{i,1},\dots,u_{i,m_i})$.

We prove the invariant properties (i) and (ii) separately for the atoms in Δ_{D_i} (1), $\Delta_C\sigma$ (2), and $P(t_1,\dots,t_m)\sigma$ (3) of the resolvent E.

(1) We start with the atoms in Δ_{D_i}, which are ground and therefore satisfy the invariant conditions (i) and (ii) by induction hypothesis analogously as in the case where the clause C satisfies $\mathcal{PVD}$.

(2) The next part are the atoms in $\Delta_C\sigma$, for which we first prove the invariant condition (i). For all variables $x \in vars(\Delta_C)$ holds:

$$depth(x\sigma,\Delta_C\sigma) = depth(x,\Delta_C) + depth(x\sigma).$$

Because the atoms in Δ_C are not depth increasing, we have

$$depth(x,\Delta_C) \le depth(x,\Gamma_C)$$

for all $x \in vars(\Delta_C)$. Consequently, we can approximate $depth(x,\Delta_C)$ with $depth(x,\Gamma_C)$ and get

$$\begin{aligned} depth(x\sigma,\Delta_C\sigma) &\le depth(x,\Gamma_C) + depth(x\sigma) \\ &= depth(x\sigma,\Gamma_C\sigma) \end{aligned}$$

and $\Gamma_C = \bigcup_i Q_i(s_{i,1},\dots,s_{i,m_i})$. With

$$Q_i(s_{i,1},\dots,s_{i,m_i})\sigma = Q_i(u_{i,1},\dots,u_{i,m_i})$$

follows that $\mathit{depth}(Q_i(s_{i,1},\dots,s_{i,m_i})\sigma) \leq d$ for all i and, therefore,

$$\mathit{depth}(x\sigma, \Delta_C\sigma) \leq d.$$

Thus, it holds $\mathit{depth}(\Delta_C\sigma) \leq d$ which proves the invariant condition (i) for the atoms in $\Delta_C\sigma$.

Next, we have to prove the invariant condition (ii) for the atoms in $\Delta_C\sigma$ for which we consider a derived atom $A(t_1,\dots,t_r)\sigma \in \Delta_C\sigma$ where

$$\mathit{depth}(A(t_1,\dots,t_r)\sigma) > \frac{d}{2}.$$

We prove the invariant condition (ii) by contradiction and show that such an atom cannot exist. To obtain such an atom, we need a partner clause

$$D_i = \rightarrow Q_i(u_{i,1},\dots,u_{i,m_i}), \Delta_{D_i}$$

with

$$Q_i(s_{i,1},\dots,s_{i,m_i})\sigma = Q_i(u_{i,1},\dots,u_{i,m_i})$$

where $\mathit{depth}(Q_i(u_{i,1},\dots,u_{i,m_i})) > \frac{d}{2}$, and

$$\mathit{vars}(Q_i(s_{i,1},\dots,s_{i,m_i})) \cap \mathit{vars}(A(t_1,\dots,t_r)) \neq \emptyset,$$

because $\mathit{depth}(C) \leq \frac{d}{2}$, and $\mathit{depth}(x,\Delta_C) \leq \mathit{depth}(x,\Gamma_C)$ as the atoms from Δ_C are not depth increasing. But for D_i holds the induction hypothesis, from which it follows that

([k].iia) $\mathit{warg}(Q_i(u_{i,1},\dots,u_{i,m_i})) \neq [\ ]$, Q_i is reachable from a depth increasing clause satisfying BDI-2, and for all arguments $u_{i,p_i} \in \mathit{warg}(Q_i(u_{i,1},\dots,u_{i,m_i}))$ holds $\mathit{depth}(u_{i,p_i}) \leq \frac{d}{2}$, or

([k].iib) $\mathit{warg}(Q_i(u_{i,1},\dots,u_{i,m_i})) = [\ ]$ and $Q_i(u_{i,1},\dots,u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1.

In case ([k].iia), the atom Q_i is reachable from a depth increasing clause satisfying BDI-2 and so is also A because A is reachable from Q_i by the clause C. Now consider a variable

$$x \in (\mathit{vars}(Q_i(s_{i,1},\dots,s_{i,m_i})) \cap \mathit{vars}(A(t_1,\dots,t_r))).$$

According to Definition 5.7 (v), having that Q_i is reachable from a depth increasing clause, it must hold that $x \in \mathit{vars}(\mathit{warg}(Q_i(s_{i,1},\dots,s_{i,m_i})))$. But then, it follows with

$$Q_i(s_{i,1},\dots,s_{i,m_i})\sigma = Q_i(u_{i,1},\dots,u_{i,m_i})$$

from $\mathit{depth}(u_{i,p_i}) \leq \frac{d}{2}$ for all $u_{i,p_i} \in \mathit{warg}(Q_i(u_{i,1},\dots,u_{i,m_i}))$ that also

$$\mathit{depth}(s_{i,p_i}\sigma) \leq \frac{d}{2} \text{ for all } s_{i,p_i}\sigma \in \mathit{warg}(Q_i(s_{i,1},\dots,s_{i,m_i})\sigma).$$

Because $A(t_1, \ldots, t_r)$ is not depth increasing, we obtain that $depth(A(t_1, \ldots, t_r)\sigma) \leq \frac{d}{2}$ which contradicts our assumption and finishes the case ([k].iia).

The other case is ([k].iib) where we have $warg(Q_i(u_{i,1}, \ldots, u_{i,m_i})) = [\ ]$ and the atom $Q_i(u_{i,1}, \ldots, u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1. Let

$$C' = \Gamma_{C'} \rightarrow Q_i(q_{i,1}, \ldots, q_{i,m_i})\Delta_{C'}$$

be this clause with $Q_i(q_{i,1}, \ldots, q_{i,m_i})\sigma' = Q_i(u_{i,1}, \ldots, u_{i,m_i})$. Then it follows from Definition 5.6 (ii) that $Q_i(q_{i,1}, \ldots, q_{i,m_i})$ and $Q_i(s_{i,1}, \ldots, s_{i,m_i})$ are similar, and for all

$$y \in vars(Q_i(s_{i,1}, \ldots, s_{i,m_i})) \cap vars(A(t_1, \ldots, t_r))$$

where $s_{i,p_i}|_p = y$, $q_{i,p_i}|_p = x'$, and $depth(x', q_{i,p_i}) > depth(x', \Gamma_{C'})$ holds $depth(y, A(t_1, \ldots, t_r)) = 0$. With $depth(C') \leq \frac{d}{2}$, $depth(D_i) \leq d$, and $depth(x'\sigma') \leq \frac{d}{2}$ which follows analogously as in the case for PVD, we obtain that

$$depth(y\sigma) \leq \frac{d}{2}$$

because

$$Q_i(s_{i,1}, \ldots, s_{i,m_i})\sigma = Q_i(u_{i,1}, \ldots, u_{i,m_i}) = Q_i(q_{i,1}, \ldots, q_{i,m_i})\sigma'$$

where the variables $y \in vars(Q_i(s_{i,1}, \ldots, s_{i,m_i})) \cap vars(A(t_1, \ldots, t_r))$ were the only possible candidates to obtain $depth(A(t_1, \ldots, t_r)\sigma) > \frac{d}{2}$ (remember that $depth(x, \Delta_C) \leq depth(x, \Gamma_C)$ because Δ_C is not depth increasing). These candidates have now been proven not to be depth increasing, and the case ([k].iib) is finished because there is no atom $A(t_1, \ldots, t_r)\sigma \in \Delta_C\sigma$ such that $depth(A(t_1, \ldots, t_r)\sigma) > \frac{d}{2}$ and $A(t_1, \ldots, t_r)\sigma$ originates from a depth incrasing clause satisfying BDI-1 which contradicts our assumption. This completes the proof of the invariant condition (ii) for the atoms in $\Delta_C\sigma$.

(3) The remaining part is the atom $P(t_1, \ldots, t_m)$ with its depth increasing argument position j.
We first prove the invariant condition (i) by contradiction for $P(t_1, \ldots, t_m)$. Because the argument position j is the only argument position increasing the depth with $depth(x, t_j) > depth(x, \Gamma_C)$ for one or more variables $x \in vars(t_j)$, we assume that

$$depth(t_j\sigma) > d$$

in $P(t_1\sigma, \ldots, t_m\sigma) \in E$. Because $C \in N$, we have $depth(P(t_1, \ldots, t_m)) \leq \frac{d}{2}$ by definition. Hence, we need a partner clause

$$D_i = \rightarrow Q_i(u_{i,1}, \ldots, u_{i,m_i}), \Delta_{D_i}$$

(from level $[k]$) such that $depth(Q_i(u_{i,1}, \ldots, u_{i,m_i})) > \frac{d}{2}$ and

$$vars(Q_i(s_{i,1}, \ldots, s_{i,m_i})) \cap vars(t_j) \neq \emptyset.$$

But for D_i holds the induction hypothesis, from which it follows that

([k].iia) $warg(Q_i(u_{i,1}, \ldots, u_{i,m_i})) \neq [\ ]$, Q_i is reachable from a depth increasing clause satisfying BDI-2, and for all arguments $u_{i,p_i} \in warg(Q_i(u_{i,1}, \ldots, u_{i,m_i}))$ holds $depth(u_{i,p_i}) \leq \frac{d}{2}$, or

([k].iib) $warg(Q_i(u_{i,1}, \ldots, u_{i,m_i})) = [\ ]$ and $Q_i(u_{i,1}, \ldots, u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1.

In case ([k].iia), the atom Q_i is reachable from a depth increasing clause C' satisfying BDI-2. Let

$$C' = \Gamma_{C'} \to P'(q_1, \ldots, q_{m'}), \Delta_{C'}$$

be this clause and $C' \neq C$. Thus, C is another depth increasing clause satisfying BDI-2 and P is reachable from Q_i with the clause C. But then, P is reachable from P' which violates Definition 5.8.
Hence, it must hold $C' = C$, and it follows from Definition 5.7 (iii) (1), that P and Q_i are of the same arity, i.e., $m = m_i$.
If $arity(Q_i) = arity(P) = 1$ then the requirement of Definition 5.7 (iii) (3), namely

$$vars(s_j) \cap vars(P(t_1, \ldots, t_m)) = \emptyset,$$

contradicts our assumption of having

$$\underbrace{vars(Q_i(s_j))}_{=vars(s_j)} \cap \underbrace{vars(t_j)}_{=vars(P(t_j))} \neq \emptyset.$$

Otherwise, we have $arity(Q_i) = arity(P) > 1$. Definition 5.7 (ii) states that $t_p \in warg(P(t_1, \ldots, t_m))$ for all argument positions $p \neq j$, and $t_j \notin warg(P(t_1, \ldots, t_m))$. Consequently, it follows together with Definition 5.7 (iii) (2) that $s_{i,p} = t_p$ for all argument positions $p \neq j$. According to ([k].iia), it holds $depth(u_{i,p}) \leq \frac{d}{2}$ for all $p \neq j$, and with

$$Q_i(u_{i,1}, \ldots, u_{i,m_i}) = Q_i(s_{i,1}, \ldots, s_{i,m_i})\sigma$$

follows that also $depth(s_{i,p}\sigma) \leq \frac{d}{2}$. Further, it holds because of condition Definition 5.7 (iii) (3) that

$$vars(s_j) \cap vars(P(t_1, \ldots, t_m)) = \emptyset,$$

and for this reason, we obtain for all

$$x \in vars(t_j) \cap vars(Q_i(s_{i,1}, \ldots, s_{i,m_i}))$$

that

$$x \in \bigcup_{p \neq j} vars(s_{i,p})$$

Consequently, we can calculate

$$depth(t_j\sigma) = \underbrace{depth(t_j)}_{\leq \frac{d}{2}} + \underbrace{depth(x\sigma)}_{\leq \frac{d}{2}} \leq d.$$

This contradicts our assumption that $depth(t_j\sigma) > d$ and completes the first case ([k].iia).

The other case is ([k].iib), where we have $warg(Q_i(u_{i,1}, \ldots, u_{i,m_i})) = [\ ]$ and the atom $Q_i(u_{i,1}, \ldots, u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1. Let

$$C' = \Gamma_{C'} \to Q_i(q_{i,1}, \ldots, q_{i,m_i})\Delta_{C'}$$

be this clause with $Q_i(q_{i,1}, \ldots, q_{i,m_i})\sigma' = Q_i(u_{i,1}, \ldots, u_{i,m_i})$. Then it follows from Definition 5.6 (ii) that $Q_i(q_{i,1}, \ldots, q_{i,m_i})$ and $Q_i(s_{i,1}, \ldots, s_{i,m_i})$ are similar, and for all

$$y \in vars(Q_i(s_{i,1}, \ldots, s_{i,m_i})) \cap vars(P(t_1, \ldots, t_m))$$

where $s_{i,p_i}|_p = y$, $q_{i,p_i}|_p = x'$, and $depth(x', q_{i,p_i}) > depth(x', \Gamma_{C'})$ holds $depth(y, P(t_1, \ldots, t_m)) = 0$. But this contradicts the construction of BDI-2 where $depth(x, t_j) > depth(x, \Gamma_C)$ and then $depth(x) = depth(y) > 0$. This completes the case ([k].iib) and proves the invariant condition (i).

Next, we continue by showing the invariant condition (ii) for the atom $P(t_1, \ldots, t_m)\sigma$, and assume that $depth(P(t_1, \ldots, t_m)\sigma) > \frac{d}{2}$. Because C satisfies BDI-2, we only have to verify condition (iia) for the atom $P(t_1, \ldots, t_m)\sigma$. We prove (iia) by contradiction and assume that we have an argument $t_p\sigma$ with $p \neq j$ and therefore $t_p\sigma \in warg(P(t_1, \ldots, t_m)\sigma)$ but $depth(t_p\sigma) > \frac{d}{2}$.
The clause C satisfies BDI-2, and for this reason, we have

$$depth(x, t_p) \leq depth(x, \Gamma_C)$$

for all argument positions $p \neq j$ and variables $x \in vars(t_p) \cap vars(\Gamma_C)$. Thus, we need an atom $Q_i(s_{i,1}, \ldots, s_{i,m_i}) \in \Gamma_C$ having an argument s_{i,p_i} such that $depth(s_{i,p_i}\sigma) > \frac{d}{2}$ and $vars(s_{i,p_i}) \cap vars(t_p) \neq \emptyset$. Let

$$D_i = \ \to Q_i(u_{i,1}, \ldots, u_{i,m_i}), \Delta_{D_i}$$

be the corresponding clause (from level [k]) with

$$Q_i(s_{i,1}, \ldots, s_{i,m_i})\sigma = Q_i(u_{i,1}, \ldots, u_{i,m_i})$$

for which the induction hypothesis holds:

([k].iia) $\mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i})) \neq [\,]$, Q_i is reachable from a depth increasing clause satisfying BDI-2, and for all arguments $u_{i,p_i} \in \mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i}))$ holds $\mathit{depth}(u_{i,p_i}) \leq \frac{d}{2}$, or

([k].iib) $\mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i})) = [\,]$ and $Q_i(u_{i,1},\ldots,u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1.

In case ([k].iia), it follows analogously as in the case ([k].iia) of the proof of the invariant condition (i) that $\mathit{depth}(u_{i,p}) \leq \frac{d}{2}$ for all argument positions $p \neq j$. With

$$Q_i(s_{i,1},\ldots,s_{i,m_i})\sigma = Q_i(u_{i,1},\ldots,u_{i,m_i})$$

follows that $\mathit{depth}(s_{i,p}\sigma) \leq \frac{d}{2}$ and because of Definition 5.7 (iii) (2), we can eventually conclude that also $\mathit{depth}(t_p\sigma) \leq \frac{d}{2}$ for all argument positions $p \neq j$ in $P(t_1,\ldots,t_m)\sigma)$ which satisfies the invariant condition (iia).

In the other case ([k].iib), we have $\mathit{warg}(Q_i(u_{i,1},\ldots,u_{i,m_i})) = [\,]$ and $Q_i(u_{i,1},\ldots,u_{i,m_i})$ originates from a depth increasing clause satisfying BDI-1. We prove this case by contradiction. Let

$$C' = \Gamma_{C'} \rightarrow Q_i(q_{i,1},\ldots,q_{i,m_i})\Delta_{C'}$$

be the previously mentioned clause satisfying BDI-1 with

$$Q_i(q_{i,1},\ldots,q_{i,m_i})\sigma' = Q_i(u_{i,1},\ldots,u_{i,m_i}).$$

From Definition 5.6 (ii) follows that $Q_i(q_{i,1},\ldots,q_{i,m_i})$ and $Q_i(s_{i,1},\ldots,s_{i,m_i})$ are similar, and for all

$$y \in \mathit{vars}(Q_i(s_{i,1},\ldots,s_{i,m_i})) \cap \mathit{vars}(P(t_1,\ldots,t_m))$$

where $s_{i,p_i}|_p = y$, $q_{i,p_i}|_p = x'$, and $\mathit{depth}(x', q_{i,p_i}) > \mathit{depth}(x', \Gamma_{C'})$ holds $\mathit{depth}(y, P(t_1,\ldots,t_m)) = 0$. With $\mathit{depth}(C') \leq \frac{d}{2}$, $\mathit{depth}(D_i) \leq d$, and $\mathit{depth}(x'\sigma) \leq \frac{d}{2}$ which follows analogously as in the case for PVD, we obtain $\mathit{depth}(y\sigma) \leq \frac{d}{2}$. But then $\mathit{depth}(t_p\sigma) \leq \frac{d}{2}$ must holds which contradicts our assumption of $\mathit{depth}(t_p\sigma) > \frac{d}{2}$.
This finishes the case ([k].iib) and also completes the proof of the invariant condition (iia) for the atom $P(t_1,\ldots,t_m)\sigma$.

Case: Factoring
The application of the Factoring rule on a (positive) clause C is handled analogously to the base case and can be reduced to the application of the Condensation rule, and thus producing a resolvent which is a strict subset of its parent clause C. $\Diamond$

In order to emphasize the thin line between decidability and undecidability on our new class definition, we present two examples each violating only one of the conditions of $\mathcal{BDI}$, and show, that it is possible to encode the PCP [30] using the relaxed conditions. Consider an alphabet $\Sigma = \{0, 1\}$. We construct a clause set such that an instance of the PCP problem has a solution if and only if the clause set is unsatisfiable. We encode strings over '0', '1' by using terms built from the constant a and the monadic function symbols f_0, f_1. For example, the word 011 would be represented as $f_1(f_1(f_0(a)))$. The corresponding string s for a term is denoted as $f_s(x)$. For a PCP instance $((u_1, v_1), (u_2, v_2), \ldots, (u_m, v_m))$, the overall clause set representing the PCP encoding is:

$$\begin{array}{rcll} & \rightarrow & P(f_{u_i}(a), f_{v_i}(a)) & 1 \leq i \leq m \qquad (1.1)\\ P(x,y) & \rightarrow & P(f_{u_i}(x), f_{v_i}(y)) & 1 \leq i \leq m \qquad (1.2)\\ P(x,x) & \rightarrow & & \qquad (1.3)\end{array}$$

The clauses of the form (1.1) represent the start state for m words and clause (1.2) the recursion to construct larger words. Eventually, clause (1.3) neglects the existence of a common word.

Consider the below clause set of Example 5.12. The clauses (2.1) and (2.4) both satisfy condition $\mathcal{PVD}$ while the clauses (2.2) and (2.3) both satisfy the conditions of BDI-2. In contrast to the standard formalization of the PCP problem, the extension of words (original clause (1.2)) is now spread over two clauses ((2.2) and (2.3)). However, these clauses in combination do not satisfy $\mathcal{BDI}$-(iv), because the predicate P is reachable from Q_i (and vice versa).

Example 5.12

$$\begin{array}{rcll} & \rightarrow & P(f_{u_i}(a), f_{v_i}(a)) & 1 \leq i \leq m \qquad (2.1)\\ P(x,y) & \rightarrow & Q_i(f_{u_i}(x), y) & 1 \leq i \leq m \qquad (2.2)\\ Q_i(x,y) & \rightarrow & P(x, f_{v_i}(y)) & 1 \leq i \leq m \qquad (2.3)\\ P(x,x) & \rightarrow & & \qquad (2.4)\end{array}$$

So dropping the reachability condition of $\mathcal{BDI}$ leads to an undecidable clause class.

Consider the below clause set of Example 5.13. Here, we have used the same idea as in Example 5.12, namely, to distribute the extension of words over several clauses (3.2)-(3.5). The clauses (3.1) and (3.6) satisfy $\mathcal{PVD}$ while the remaining clauses are candidates to satisfy BDI-1. Starting from the clauses (3.2), the atoms with Q_i, R_i-predicates occuring in (3.3)-(3.5) are all similar, respectively. However, the variable condition of BDI-1-(iv) is violated in (3.3) and (3.4). Consider one of the clauses resulting from formula (3.3) as an example: The atom $Q_i(f_{u_i}(x), y)$ is reachable from a depth increasing clause (a clause resulting from (3.2)), $\mathit{vars}(\mathit{warg}(Q_i(f_{u_i}(x), y))) = \emptyset$ and there are no other atoms that are not reachable from a depth increasing clause on the left hand side. Consequently, the right hand side of the clause had to be ground to satisfy condition BDI-1-(iv).

Example 5.13

$$\begin{aligned}
&\rightarrow && P(f_{u_i}(a), f_{v_i}(a)) && 1 \leq i \leq m && (3.1)\\
P(x,y) &\rightarrow && Q_i(f_{u_i}(x), y), R_i(x, f_{v_i}(y)) && 1 \leq i \leq m && (3.2)\\
Q_i(f_{u_i}(x), y) &\rightarrow && R_i(x, f_{v_i}(y)) && 1 \leq i \leq m && (3.3)\\
R_i(x, f_{v_i}(y)) &\rightarrow && Q_i(f_{u_i}(x), y) && 1 \leq i \leq m && (3.4)\\
Q_i(f_{u_i}(x), y), R_i(x, f_{v_i}(y)) &\rightarrow && P(f_{u_i}(x), f_{v_i}(y)) && 1 \leq i \leq m && (3.5)\\
P(x,x) &\rightarrow && && && (3.6)
\end{aligned}$$

So dropping condition BDI-1-(iv) leads to an undecidable clause class.

In general, any violation of the conditions of BDI-1 or BDI-2 results in a clause class where Hyper-resolution is no longer a decision procedure. The above two clause sets show that at least two of the conditions are mandatory in order to obtain a decidable clause class.

5.4 From Hyper to Ordered Resolution

Hyper-resolution enumerates all positive ground facts from a given clause set. For many practical applications this is not feasible. For example, in the context of our authorization analysis, thousands of authorization definitions for a large number of users need to me modeled. They imply a huge number of derivable ground facts representing the exact authorization instantiations for all these users. Therefore, we want to employ a specific selection strategy on atoms in order to avoid the naive enumeration of all derivable positive ground clauses. Consider the following abstract, but real world, set of clauses as an example to sketch the idea. Assume 10.000 ground atoms $\rightarrow A(a_i, b_j)$ relating authorizations a_i to possible values b_j. Assume 10.000 ground atoms of the form $\rightarrow \mathit{Holds}(u_i, a_j)$ that assign authorizations a_j to users u_i. Then already a clause of the form $\mathit{Holds}(x,y), A(y,z) \rightarrow \mathit{Access}(x,z)$ results already in a potential quadratic (10k*10k) number of concrete access rights. However, in some business process, these rights are only needed in a very specific way, e.g., a clause of the form $P(x,y,z)$, $\mathit{Access}(x_1,x)$, $\mathit{Access}(x_1,y)$, $\mathit{Access}(x_1,z) \rightarrow Q(x_1,y,z)$ requires three specific rights in order to derive $Q(x_1,y,z)$. If the atom $P(x,y,z)$ can be selected at first in this clause, then the overhead of generating all access rights for all users in order to reason about $Q(x_1,y,z)$ can be prevented. Therefore, we want to turn ordered resolution with selection into a decision procedure for $\mathcal{BDI}$.

In general, Ordered Resolution is not a decision procedure for $\mathcal{BDI}$. However, as it is shown below, the $\mathcal{BDI}$ class justifies two additional reduction rules that then make Ordered Resolution terminating.

Theorem 5.14
Let N be an unsatisfiable clause set of the class $\mathcal{BDI}$ and

$$d_N = 2 \cdot \max\{\mathit{depth}(\Delta) \mid (\Gamma \to \Delta) \in N\}.$$

Consider a hyper-resolution proof of the empty clause with ordering $\succ$. Then there is a (non-ground) ordered resolution proof of the empty clause with respect to $\succ$ and an arbitrary selection strategy such that $\mathit{depth}(C) \leq d_N$ for all clauses C derived in this ordered resolution proof.

Proof. By Theorem 5.11 there is a hyper-resolution proof of the empty clause where any generated clause does not exceed the depth bound d_N. Having a hyper-resolution proof for N with depth bound d_N, we can construct an inconsistent subset S of N^* and ground it by taking the substitutions used in the hyper-resolution proof such that all ground clauses still have depth bound d_N. By refutational completeness of the ordered resolution calculus with selection, it is still possible to derive the empty clause from S. Because all inferences are ground in the refutation of S, any derived ground clause respects the depth bound d_N. Using the standard lifting lemma, a non-ground refutation of the original set N is constructed where it still holds $\mathit{depth}(C) \leq d_N$ for all clauses C derived by the ordered resolution calculus with an arbitrary selection strategy. $\Diamond$

We exploit Theorem 5.14 by the following two parameterized reduction rules that eventually enable a finite saturation of a $\mathcal{BDI}$ clause set via ordered resolution with selection.

Definition 5.15 (Variable Condensation(k))
The reduction

$$\frac{C}{C\sigma_{1,2}, \ldots, C\sigma_{l-1,l}}$$

where $\mathit{vars}(C) = \{x_1, \ldots, x_l\}$, $l > k$ and $\sigma_{i,j} = \{x_i \mapsto x_j\}$ for all i, j with $1 \leq i < l$, $i < j \leq l$, is called *Variable Condensation.*

Definition 5.16 (Depth Cutoff(k))
The reduction

$$\frac{C}{}$$

where $\mathit{depth}(C) > k$ is called *Depth Cutoff.*

Theorem 5.17
Let N be a finite set of clauses satisfying $\mathcal{BDI}$ and Σ' be the signature symbols

occurring in N. Then the ordered resolution calculus with an arbitrary selection strategy together with *Depth Cutoff*(d_N) and *Variable Condensation*(e_N) where $d_N = 2 \cdot \max\{\mathit{depth}(\Delta_C) \mid C \in N\}$ and $e_N = |\{t \mid t \in T(\Sigma'), \mathit{depth}(t) \leq d_N\}|$ is complete and terminating.

Proof. It follows from Theorem 5.14 that the standard ordered resolution calculus with an arbitrary selection strategy is able to derive the empty clause and none of the derived clauses in the ordered resolution proof exceeds the depth of d_N. Thus, if we have a clause D with $\mathit{depth}(D) > d_N$, we apply *Depth Cutoff*(d_N) on D and discard it as it will not be required to refute N in case of a contradiction.

Additionally, with respect to the finitely many signature symbols in N and the depth limit d_N only e_N many different ground terms need to be considered in any proof. Therefore, we can apply *Variable Condensation*(e_N) on any (derived) clause D such that the total number of different variables in any derived clause is bounded as well. $\Diamond$

The ordered resolution calculus including the two additional reduction rules Variable Condensation and Depth Cutoff provides a decision procedure for $\mathcal{BDI}$. Recall that termination is achieved for resolution if the length (number of literals in a clause) and depth (maximal depth of a literal in a clause) of newly generated clauses are finitely bound [20]. This is ensured by our additional rules. Any derived clause is bounded by the depth limit (otherwise Depth Cutoff is applied). Further, it is also not possible to generate clauses with infinite different literals because only finitely many different ground terms can be constructed taking the depth limit into account. For the non-ground case together with the depth bound, one can only increase the number of variables but then the rule Variable Condensation applies.

6 Implementation

I have implemented the verification of the $\mathcal{BDI}$ class criteria (the $\mathcal{BDI}$ properties presented in Chapter 5) into the theorem prover SPASS version 3.8 (not officially released yet, latest release is 3.7, [37]). The implementation makes it possible to automatically decide for a given problem, whether it satisfies the requirements of $\mathcal{BDI}$ or not. If the conditions of $\mathcal{BDI}$ are satisfied, then the prover is guaranteed to terminate.

This section contains the details related to the implementation where the following algorithms explain all essential functions. Additionally, the algorithms use helper functions not containing further logic which are therefore not presented in detail. References to essential functions are written in typewriter font, e.g., `wargEqual`(), and calls of helper functions are written italics, e.g., *getLiteral*().

I have created a new module `bdi` where most of the implementation's source code is located. In addition, it was necessary to extend some existing modules, for example, to store additional information about clauses, to print the predicates (graph nodes) of a graph, or to compare nodes in a graph.

The input to the main loop of the prover is – after some preprocessing – a list of clauses representing the input problem. I have attached the analysis of the clauses at the position right after where the input has been parsed and the corresponding list of (input) clauses has been created, but before the actual main loop is started.

At first, the predicate dependency graph is created by adding edges for all predicate symbols occurring in the input clauses as described in Section 4.2.5. Afterwards, an algorithm presented by Tarjan [34], which uses depth-first search in order to identify, and number strongly connected components (the cycles) is applied to the graph. Eventually, the conditions for $\mathcal{BDI}$ (see Section 5.8) are checked with the aid of the predicate dependency graph.

The *graph* data structure contains the total number of nodes, a list of graph nodes (and some minor auxiliary attributes that not need to be further described). The *graph node* structure in turn consists mainly of an incremented node number, a list of neighbors, and a pointer to store arbitrary additional information. During the construction of the graph, I use several lists for remembering clauses to avoid duplicated loops over all clauses when checking the sub-conditions of $\mathcal{BDI}$. Please note that the implementation always works with pointers, so the clauses are stored only once, and every list stores just pointers to selected clauses. The following structure shows the additional information that is attached to every graph node.

```
typedef struct BDINODEINFO_HELP
{
  /* List of pointers to clauses that can be used
     for resolution (= clauses with literals on the
     left hand side where the literal symbol matches
     the node symbol. */
  LIST    candidates;

  /* List of pointers to clauses having positive
     literals which share the same predicate symbol. */
  LIST    clausesPosSameSymbol;

  /* List of pointers to all clauses where the
     predicate symbol occurs. */
  LIST    clausesOccuring;

  /* List of argument positions to watch. */
  LIST    warg;

  /* Switch to avoid warg list overwriting. */
  BOOL    wargEmpty;

} BDINODEINFO_STRUCT;
```

As mentioned above, the conditions of $\mathcal{BDI}$ are checked using the dependency graph after its construction and identification of the strongly connected components. This task is presented in algorithm 1. In order to store further information about a clause, I have extended the module `clause` by the following additional attributes:

- depthIncLiterals: A list of depth increasing literals where it exists a variable x in the clause $\Gamma \rightarrow \Delta$ such that $depth(x, \Delta) > depth(x, \Gamma)$. If the list is empty, the clause is not depth increasing.

- satPVDa, satPVDb, satPVD: Flags to store the result of the condition check of the single, and combined conditions of $\mathcal{PVD}$, respectively (see Definition 5.1, and algorithm 1, line 6, and algorithm 2).

- satBDI1: Flag to store the result of the BDI-1 condition check (see Definition 5.6, and algorithm 1, line 19).

- satBDI2: Flag to store the result of the BDI-2 condition check (see Definition 5.7, and algorithm 1, line 11).

After the initialization, a loop over all input clauses is started in line 4 in order to check the conditions of $\mathcal{PVD}$ and to determine whether a clause is depth increasing or not. Please note that the access of class attributes or properties is handled in following algorithms using the dot operator, see line 5 for the first occurrence. The condition check for $\mathcal{PVD}$ is mainly done by the function `bdi_satPVD()`, called in line 6, which marks the depth increasing variables with varDepthIncMark, stamps the depth increasing argument positions of the related literal, and remembers the depth increasing literals of the clause in the list depthIncLiterals of that clause. The result of the check is then stored in the attribute satPVD of the clause. Of course, if a clause violates the single condition satPVDa (PVD-(i), Definition 5.1), it can never satisfy a BDI condition (line 7) because PVD-(i) must hold both for BDI-1 and BDI-2. Otherwise, if only PVD-(ii) is violated, the clause is depth increasing and has to be stored in line 9 to be considered for checking the conditions of BDI-1, and BDI-2 in the subsequent loops.

The loop to check the BDI-2 conditions starts in line 11, where the function `bdi_satBDI2()` is actually doing the job: It checks the single BDI-2 conditions and initializes also the watched argument lists that are needed for the following BDI-1 condition check. Additionally, it remembers all clauses satisfying BDI-2 in order to be able to check BDI-(iv) at the end (line 23).

The subsequent loop (line 16) considers again all depth increasing clauses to check the BDI-1 conditions (function `bdi_satBDI1()`, line 19), but only if the corresponding clause does not already satisfy BDI-2 (line 17). The result of the BDI-1 condition check is also stored in the clause, like the previous results PVD, and BDI-2.

Eventually, a loop over all depth increasing clauses is started in line 20, in order to check whether the clause satisfies BDI-1 or BDI-2. If a clause satisfies BDI-2, all other clauses satisfying BDI-2 have to be checked in a separate loop in line 23 for reachability, because in order to satisfy BDI-(iv), two depth increasing clauses both satisfying BDI-2 may not reachable from each other. This requirement is implemented by taking the first literal from the respective depth increasing literal lists (lines 24-25)[1], and check reachability of these two literals by means of the graph, if the literals are not equal. The equality is excluded and can be excluded because it represents the case that the clauses Clause and Clause2 are equal and we want to check reachability only for two different clauses Clause and Clause2.

[1]Please note that there exists only one element in the depth increasing literal list for a clause satisfying BDI-2.

Algorithm 1: `bdi_checkConditions(InputClauses, Graph)`

input: A list of all input clauses, and the predicate graph resulting from the input clauses.

effect: Checks the $\mathcal{BDI}$ conditions for each input clause and prints information if one violates a condition.

```
   /* Init */
1  int varDepthIncMark = getMark();
2  LIST depthIncClauses = ∅; /* Stores all depth increasing
   clauses */
3  LIST BDI2 = ∅; /* Stores all clauses satisfying BDI-2 */
4  foreach clause Clause from InputClauses do
5      Clause.depthIncLiterals = ∅;
6      Clause.satPVD = bdi_satPVD(Clause, varDepthIncMark, Graph);
       /* See Algorithm 2 */
7      if (Clause.satPVDa == FALSE) then
           /* Print: Clause violates BDI conditions. */
8      else if (Clause.satPVDb == FALSE) then
           /* Clause is depth increasing */
9          depthIncClauses = depthIncClauses ∪ {Clause};
10 foreach clause Clause from depthIncClauses do
11     Clause.satBDI2 = bdi_satBDI2(Clause, InputClauses,
       depthIncClauses, varDepthIncMark, Graph); /* See Algorithm 3
       */
12     if (Clause.satPVDa == FALSE
13     and Clause.satBDI2 == TRUE
14     and length(Clause.depthIncClauses) == 2) then
           /* Remember clause to check BDI-(iv) */
15         BDI2 = BDI2 ∪ {Clause};
16 foreach clause Clause from depthIncClauses do
17     if (Clause.satBDI2 == TRUE) then
           /* Current clause OK, continue with next clause */
18     else
19         Clause.satBDI1 = bdi_satBDI1(Clause, depthIncClauses,
           varDepthIncMark, Graph); /* See Algorithm 4 */
```

```
20 foreach clause Clause from depthIncClauses do
21     if (Clause.bdi_satBDI1 == TRUE) then
           /* Current clause OK, continue with next clause */
22     else if (Clause.bdi_satBDI2 == TRUE) then
           /* Check all (other) clauses satisfying BDI-2 if they
              are reachable from each other */
23         foreach clause Clause2 from BDI2 do
               /* Get first literals from corresponding depth
                  increasing literals lists */
24             LITERAL Literal1 = first(Clause.depthIncClauses);
25             LITERAL Literal2 = first(Clause2.depthIncClauses);
               /* Check predicates */
26             if (predSymbol(Literal1) $\neq$ predSymbol(Literal2) and
               bdi_isReachable(Literal1, Literal2, Graph)) then
                   /* Print:  Violation of BDI-(iv).  */
27     else
           /* Print:  Clause violates BDI conditions.  */
```

Algorithm 2 shows the implementation of the single condition checks that need to hold for $\mathcal{PVD}$ (see Definition 5.1) for a clause. After the initialization in the lines 1-2, all variable positions from the variables occurring in Γ and Δ are collected in the lines 3-4, and 5-6, respectively. The function *varPos*(`<list>`) simply extracts all variables and their positions from the given list of literals.

The next step is to verify all collected variable positions. The loop over all positions starts in line 7, and the condition PVD-(i) is verified at first, in line 9: Every variable varDelta (at some position in Δ) must satisfy varDelta $\in$ *vars*(Γ). If this requirement is violated, the loop can be immediately left and the result returned (which is **FALSE** because the given clause does not satisfy the conditions of $\mathcal{PVD}$). Otherwise, every variable occurring at some position in Γ is examined. If the same variable symbol is considered (varDelta = varGamma, line 14), and the depth of that variable in Δ (with respect to its position) is larger than the depth of the same variable occurring in Γ in the given clause, a depth increasing variable and literal has been found. Then, in line 16-17, the depth increasing literal is stored in the clause and the literal, as well as the corresponding argument, and the depth increasing variable are marked. Further, in line 19, condition PVD-(ii) is set to false if not yet done and the loop continues to examine the remaining variable positions (as we want to mark all depth increasing variables).

The section starting in line 20 eventually checks whether the given clause (now with the depth increasing variable positions marked) can be considered as a candidate to satisfy BDI-2. This is not the case if it has more than one depth increasing literal for which the graph node information is adjusted by flushing the watched argument list and setting the flag wargEmpty = **TRUE** to prevent later overriding of the watched argument list. The idea and implementation of the watched argument list is explained in detail in algorithm 6. The node information is attached to each predicate symbol and is accessible via the predicate symbol of the literal.

Algorithm 2: `bdi_satPVD(Clause, depthIncMark, Graph)`

input : A clause, a depth increasing mark, and the predicate graph.
effect : Checks the $\mathcal{PVD}$ conditions on the given clause.
output: Returns true if the clause satisfies $\mathcal{PVD}$, false otherwise.

```
   /* Init */
1  Clause.satPVDa = Clause.satPVDb = TRUE;
2  LIST VarPosGamma = VarPosDelta = $\emptyset$;
   /* Collect variable positions in $\Gamma$ */
3  foreach VarPos from varPos($\Gamma$) of Clause do
4      VarPosGamma = VarPosGamma $\cup$ {VarPos};
   /* Collect variable positions in $\Delta$ */
5  foreach VarPos from varPos($\Delta$) of Clause do
6      VarPosDelta = VarPosDelta $\cup$ {VarPos};
7  foreach VarPosFromDelta $\in$ VarPosDelta do /* Check variables at
   the positions */
8      SYMBOL varDelta = getVarAtPos(VarPosFromDelta);
9      if (varDelta $\notin$ vars($\Gamma$)) then
10         Clause.satPVDa = FALSE;
           /* Print: Clause violates PVD-(i). */
11         break;
12     foreach VarPosFromGamma $\in$ VarPosGamma do
13         SYMBOL varGamma = getVarAtPos(VarPosFromGamma);
14         if (varDelta == varGamma and
           depth(VarPosFromDelta, $\Delta$) > depth(VarPosFromGamma, $\Gamma$))
           then
15             Clause.depthIncLiterals = Clause.depthIncLiterals
               $\cup$ {getLiteral(VarPosFromDelta)};
16             Stamp depth increasing argument of literal and literal itself;
17             Mark depth increasing variable (position) with
               depthIncMark;
```

```
18            if (Clause.satPVDb == TRUE) then
19                Clause.satPVDb = FALSE;
                  /* Print:  Clause violates PVD-(ii).  */
              /* Continue with other depth increasing vars */
20    if (length(Clause.depthIncLiterals) > 1) then
          /* Clause can never satisfy BDI-2, possibly BDI-1;
             adjust graph node info for all literals from
             depthIncLiterals   */
21        foreach Literal from depthIncLiterals do
22            GRAPHNODE Node = getNode(Literal);
23            Node.nodeInfo.warg = ∅;
24            Node.nodeInfo.wargEmpty = TRUE;
25 return (Clause.satPVDa and Clause.satPVDb);
```

The algorithm 3 shows the implementation of the single condition checks for BDI-2 as well as the initialization of the watched argument list for the predicates. After the initialization in the lines 1-3, the attribute PVDa (condition PVD-(i)) is checked for the given clause in lines 5-6 because it is a requirement for BDI-2, too. The PVDa attribute has been previously set in algorithm 2 which is executed prior to the current algorithm. Afterwards, the number of depth increasing literals in the depth increasing literal list of the given clause is verified to be exactly one in the lines 7-8. If these preconditions are satisfied, the actual examination of the positive literals of the clause starts in line 9. In lines 12-13 it is checked for every stamped (depth increasing) literal whether the depth increasing argument position(s) of the given literal are the same for all positive literal occurrences with the same predicate symbol in the other input clauses. Please note that the corresponding clauses sharing the same predicate symbol have been stored during the graph construction in the list clausesPosSameSymbol to avoid repeated loops over all input clauses.

The next part is the check of the number of depth increasing arguments as well as to determine the watched argument positions for the depth increasing literal and the reachable literals in the same cycle. For this purpose, a loop over all arguments of the depth increasing literal is necessary (starting in line 16) where the number of marked arguments is verified at first in line 20. Further, the depth increasing literal and argument position is stored for later use in the lines 22-23. If the argument is not marked and therefore not depth increasing, its position is stored in the watched argument list that is kept in the additional information of the literals graph node (line 25).

The call of the function `bdi_setWargReachable()` in line 26 passes the watched argument list to all literals occurring in the same cycle than the current depth increasing literal. Please note that the watched argument list gets only passed to those literals having the wargEmpty flag not set, otherwise the watched argument list of the corresponding literal remains unchanged. A detailed explanation of the watched arguments and their implementation is given in the context of Algorithm 6.

The next part is the exploration of the negative literals of the given clause in the loop starting in line 27). For all atoms that are reachable from the depth increasing literal with common variables (line 29), the conditions of BDI-2-(iii) have to be checked: The arity of the negative literal and the depth increasing literal must be equal (BDI-2-(iii)-(1), lines 30-31), the respective watched arguments must be equal (BDI-2-(iii)-(2), lines 32-33), and the depth increasing argument and the negative literal are not allowed to have common variables (BDI-2-(iii)-(3), lines 34-35). Afterwards, the variables are collected for the later check of condition BDI-2-(v), either only from the arguments at watched argument positions if the literal is reachable from a depth increasing clause (lines 36-37), or by taking all variables from the literal otherwise (lines 38-39).

In order to check BDI-2-(iv), it's necessary to scan all clauses in the same cycle or component than the given clause for a literal that is reachable from the current depth increasing literal. For the matching clauses, we require for any literal in the succedent with a non-empty watched argument list that the watched argument list for all atoms in the antecedent is either identical or empty (checked by `wargEmptyOrEqual()` in line 42). The function stops and returns **FALSE** if this requirement is violated.

Finally, another loop over all positive literals of the given clause is performed to check the condition BDI-2-(v) in the lines 44-46. This separate loop is needed because its contents can't be moved to the first loop over the succedent of the clause (in line 9) due to dependencies. The reason is the necessary collection of the corresponding variables that only takes place in the loop over the negative literals of the given clause (starting in line 27) and which in turn needs to be executed after a loop over the positive literals (also because of dependencies). If a literal different from the depth increasing literal is encountered and the variables of the non-depth-increasing literal are not in the list of allowed variables (line 45), then the condition BDI-2-(v) is violated.

If none of the previous checks has failed, the given clause satisfies BDI-2 and the function `sat_BDI2()` returns **TRUE** in line 47.

Algorithm 3: `bdi_satBDI2`(Clause, depthIncClauses, depthIncMark, Graph)

input : A clause, the list of depth increasing clauses, a depth increasing mark, and the predicate graph.
effect : Checks the BDI-2 conditions on the given clause.
output: Returns true if the clause satisfies BDI-2, false otherwise.

```
/* Init */
LITERAL depthIncLiteral = NULL;
int depthIncArgPos = 0;
LIST allowedVars = $\emptyset$;
GRAPHNODE Node;
if (Clause.satPVDa == FALSE) then
    return FALSE;
if (length(Clause.depthIncLiterals) > 1) then
    return FALSE ;  /* Print: More than 1 depth increasing literal */
foreach Literal $\in \Delta$ of Clause do
    if (Literal is stamped) then
        Node = getNode(Literal);
        if (bdi_isUniquelyDepthInc(Literal, Node.nodeInfo.clausesPosSameSymbol) == FALSE) then
        /* See Algorithm 5 */
            return FALSE; /* Not uniquely depth increasing */
        /* Init search for depth increasing literal and argument */
        countDephIncArgs = 0;
        argPos = 0;
        foreach Argument $\in$ Literal do
            argPos++;
            if (Argument is marked) then
                countDephIncArgs++;
                if (countDephIncArgs > 1) then
                    /* Print: > 1 depth increasing argument */
                    return FALSE;
                depthIncLiteral = Literal;    /* Remember literal and argument position */
                depthIncArgPos = argPos;
            else
                Node.nodeInfo.warg = Node.nodeInfo.warg $\cup$ {argPos};
        bdi_setWargReachable(Literal, Graph);   /* See Algorithm 6 */
```

```
27 foreach Literal $\in \Gamma$ of Clause do
28     Node = getNode(Literal);
29     if (bdi_isReachable(Literal, depthIncLiteral, Graph) and
       vars(Literal) $\cap$ vars(depthIncLiteral) $\neq \emptyset$) then
       /* BDI-2-(iii)-(1), see Algorithm 7 */
30         if arity(Literal) $\neq$ arity(depthIncLiteral) then
31             return FALSE /* Print:  arities not equal */
32         else if (wargEqual(Node.nodeInfo.warg,
           getNode(depthIncLiteral).nodeInfo.warg) == FALSE) then
           /* BDI-2-(iii)-(2) */
33             return FALSE ; /* Print:  warg lists not equal */
34         else if (vars(Literal) $\cap$
           vars(getArgument(depthIncLiteral, depthIncArgPos)) $\neq \emptyset$) then
           /* BDI-2-(iii)-(3) */
               /* Print:  Violation of BDI-2-(iii)-(3) */
35             return FALSE;
       /* Collect vars for BDI-2-(v) */
36     if (bdi_isReachableFromDIC(Literal, depthIncClauses, Graph)) then
       /* Reachable from DIC, only vars from warg list; see
       Algorithm 8 */
37         allowedVars = allowedVars $\cup$ vars(Node.nodeInfo.warg);
38     else /* Not reachable from DIC, take all vars */
39         allowedVars = allowedVars $\cup$ vars(Literal);
40 foreach clause Clause2 in same cycle than Clause do /* BDI-2-(iv)
   */
41     foreach Literal $\in \Delta$ of Clause2 do
42         if (bdi_isReachable(Literal, depthIncLiteral, Graph)) and
           wargEmptyOrEqual(Clause2, Graph) == FALSE then /* See
           Algorithm 9 */
43             return FALSE ; /* Print:  BDI-2-(iv) violated.  */
44 foreach Literal $\in \Delta$ of Clause do /* BDI-2-(v) */
45     if (Literal $\neq$ depthIncLiteral and vars(Literal) $\cap$ allowedVars == $\emptyset$)
       then
46         return FALSE ;     /* Print:  BDI-2-(v) violated.  */
47 return TRUE;
```

The algorithm 4 shows the implementation of the single condition checks for BDI-1. After the initialization, the attribute PVDa (condition PVD-(i)) is checked for the given clause in the lines 3-4 analogously as in algorithm 3 for `bdi_satBDI2()`.

A loop over the antecedent of the given clause is started in line 5 where the variables for the later check BDI-1-(iv) are collected at first in the lines 7-10, analogously as in algorithm 3, line 36-39, for `bdi_satBDI2()`. Also, for any atom in the succedent of the given clause with a non-empty watched argument list it is checked that the watched argument list for all atoms in the antecedent is either identical or empty by calling the function `wargEmptyOrEqual()` in line 11.

Next, a loop over the positive literals of the given clause is started in line 13 in order to check the stamped (depth increasing) literals. Another loop to explore the partner clauses from the candidate clauses list of the current literal is started in line 16. If the literal with the same predicate symbol from the (partner) clause under inspection is unifiable with the current depth increasing literal (line 18), similarity is checked and the depth increasing variable positions from the partner literal are collected by means of the function `bdi_termSimilarVarCollect()` in line 20. If the literals are similar, each collected variable position varPos from a variable occurring in the partner clause is verified to have zero depth in line 22 which corresponds to the requirement BDI-1-(ii). Otherwise, if the unifying partner literal is not similar, the condition BDI-1-(ii) is also violated (line 24). Finally, the variables from the depth increasing literal are verified to be also member in the list of the collected variables (allowedVars) in line 25. If this requirement is not satisfied, there is a violation of condition BDI-1-(iv) and the function `sat_BDI1()` returns **FALSE**.

If none of the previous checks has failed, the given clause satisfies BDI-1 and the function `sat_BDI1()` returns **TRUE** in line 27.

Algorithm 4: `bdi_satBDI1(Clause, depthIncClauses, depthIncMark, Graph)`

input : A clause, the list of depth increasing clauses, a depth increasing mark, and the predicate graph.
effect : Checks the BDI-1 conditions on the given clause.
output: Returns true if the clause satisfies BDI-1, false otherwise.

```
   /* Init */
 1 LIST allowedVars = $\emptyset$;
 2 GRAPHNODE Node;
 3 if (Clause.satPVDa == FALSE) then
 4     return FALSE;
```

```
5  foreach Literal $\in \Gamma$ of Clause do
6      Node = getNode(Literal);
       /* Collect vars for BDI-1-(iv) */
7      if (bdi_isReachableFromDIC(Literal, depthIncClauses, Graph)) then
       /* Reachable from DIC, take only variables from warg
       list, see Algorithm 8 */
8          allowedVars = allowedVars $\cup$ vars(Node.nodeInfo.warg);
9      else /* Not reachable from DIC, take all variables */
10         allowedVars = allowedVars $\cup$ vars(Literal);
11     if (wargEmptyOrEqual(Clause, Graph) == FALSE) then /* See
       Algorithm 9 */
           /* Print:  Violation of BDI-1-(v).  */
12         return FALSE;
13 foreach Literal $\in \Delta$ of Clause do
14     Node = getNode(Literal);
15     if (Literal is stamped) then
16         foreach partner clause Clause2 $\in$ Node.nodeInfo.candidates do
           /* Examine resolution partners */
17             foreach literal with same predicate symbol
               Literal2 $\in$ Clause2 do
18                 if (isUnifiable(Literal, Literal2)) then
19                     LIST varPosCheck = $\emptyset$;
                       /* Check similarity and collect depth
                          increasing variables in partner literal
                          */
20                     if (bdi_termSimilarVarCollect(Literal, Literal2,
                       varDepthIncMark, varPosCheck)) then /* See
                       algorithm 10 */
21                         foreach varPos $\in$ varPosCheck do /* Verify
                           variable positions */
22                             if (depth(varPos, Clause2) $> 0$) then
                                   /* Print:  Violation of BDI-1-(ii)
                                      */
23                                 return FALSE;
24                     else
                           /* Print:  Violation of BDI-1-(ii),
                              literal in partner clause is not
                              similar */
           /* Check vars (collected previously) */
25         if (vars(Literal) $\cap$ allowedVars == $\emptyset$) then
               /* Print:  Violation of BDI-1-(iv).  */
26             return FALSE;
27 return TRUE;
```

Algorithm 5 checks for the property of uniquely depth increasing literals by taking a depth increasing literal as a reference, and comparing the arguments of the literals with the same predicate symbol (line 3) occurring in the given list of clauses one by one (line 6).

Algorithm 5: `bdi_isUniquelyDepthInc(Literal, Clauses)`

input : A literal, and a list of clauses to check for the property of uniquely depth increasing literals.
effect : Checks the literals occurring in the given clauses to have equal depth increasing argument positions than the given literal.
output: True or false depending if the literal occurs uniquely depth increasing in the list or not.

```
1 foreach Clause from Clauses do
2     foreach Literal2 from Clause do
3         if (equalSymbols(Literal2, Literal) == FALSE) then
          /* Symbols not equal -> continue with next literal */
4             continue;
5         else
6             foreach Argument1 from Literal, Argument2 from Literal2
              do
7                 if ((Argument1 is stamped) and (Argument2 is not
                  stamped)) then
8                     return FALSE;
9 return TRUE;
```

Algorithm 6 shows how the watched argument position lists are determined for reachable predicate symbols in the same cycle.

The initialization in the lines 1-2 extracts the node data from the given literal and prepares a list for storing nodes that need to be reprocessed. Afterwards, a loop over all graph nodes (representing the predicates) is performed to find all predicates connected within the same cycle than the given literal (using the cycle/component number for comparison, line 5). Additionally, only the predicates are of interest whose watched argument list has not been forced to stay empty (also checked in line 5). For satisfying candidates, the watched argument list is copied from the given node to the currently explored node in line 6. The next step is to verify the arity of the two nodes' predicate symbols in line 7. If the arities are different, then the current node needs to be remembered in the reprocess list for further examination (line 8) because the watched arguments positions cannot be adapted one-to-one from Node to the current node Node2.

Consider the following two atoms $P(x_1, x_2, f(x_3)), Q(x_1, g(a), x_2, f(x_3))$ occurring in one clause for a better understanding of the argument lists and assume that the argument positions $1 = x_1, 2 = x_2$ for P are watched. As mentioned above, the watched argument list of P (positions $1, 2$) is initially copied to the watched argument lists of all other reachable predicates in the same cycle (in this example on the predicate Q). However, one easily can see that the (watched) argument positions $1, 2$ of P are identical to the arguments $1, 3$ of Q. Hence, the initially copied watched argument positions $1, 2$ for Q are wrong and need to be fixed. The correction is done by identifying the same arguments and their corresponding positions between two atoms in the same clause even if there are other arguments in between. The only requirement is that the arguments stay sorted. It is not permitted to swap positions, e.g., to have the argument position of x_2 before the argument position of x_1 in Q. In our example, the argument in between in $Q(x_1, g(a), x_2, f(x_3))$ is $g(a)$ that causes an update of the initially watched argument positions for Q, i.e., positions $1, 3$ instead of $1, 2$.

The examination of the arguments, detection of the correct positions for predicates with a different arity and the subsequent update of the wrong positions is done in the loop over the remembered nodes from the reprocess list starting in line 9. For every remembered node that has already a (possibly wrong) watched argument list, all appropriate clauses (loop starting in line 10) and literals (lines 13-22) are searched to use as source (from where the watched argument positions are read from) and target (where the watched arguments have to be updated). For any found pair of source and target literal (line 23), the mapping process between the arguments of source and target literal is performed (starting in line 25). As illustrated in the previous example for P and Q, the arguments from the source literal at the given watched argument position of Node are read and the corresponding (same) arguments are searched in the target literal. The new watched argument positions in the target literal are preliminary temporary and stored in the list wargTargTmp (line 28). Afterwards, after a verification of the correct length (line 29) by comparing the temporary list with the current (possibly still wrong) watched argument list of the given node, the temporary list is assigned as the new watched argument list to the given node (line 30), and the loop continues with the next predicate to review (line 31). Otherwise, if the length of the new watched argument list differs from the length of the existing list, the old watched argument list is kept for the predicate.

Algorithm 6: `bdi_setWargReachable(Literal, Graph)`

input : A literal, and the predicate graph of the input clauses.
effect : Searches all connected nodes (predicates) in the same component and copies the watched argument list to the connected node unless the connected node has been forced to stay empty for some reason (flag wargEmpty).
output: Nothing.

```
   /* Init */
 1 GRAPHNODE Node = getNode(Literal);
 2 LIST ReprocessList = ∅;
 3 LIST GraphNodes = getGraphNodes(Graph);
 4 foreach Node2 from GraphNodes do /* Search all connected
   predicates in the same component */
 5     if ((Node2.compNumber == Node.compNumber) and
       (Node2.wargEmpty == FALSE)) then /* Same
       component/reachable, and watched argument position list
       is not forced to stay empty -> copy watched argument
       position list to reachable node */
 6         Node2.nodeInfo.warg = Node.nodeInfo.warg;
 7         if (arity(Node2.symbol) ≠ arity(Node.symbol)) then /* Arity
           different -> remember node to update warg list after
           further examination */
 8             ReprocessList = ReprocessList ∪ {Node2};
 9 foreach Node2 from ReprocessList do
10     foreach Clause from Node2.clausesOccurring do /* Search for a
       clause having a literal with the current symbol */
11         LITERAL LiteralSrc = NULL;
12         LITERAL LiteralTarg = NULL;
13         foreach Literal2 ∈ Clause do /* Examine literals */
14             GRAPHNODE Node3 = getNode(Literal2);
15             if ((LiteralSrc ≠ NULL) and (LiteralTarg ≠ NULL)) then
               /* Literals found, stop search */
16                 break;
17             if (Node3.nodeInfo.warg == ∅) then /* warg list must
               be non-empty, continue search */
18                 continue;
19             if (arity(Node3.symbol)) == arity(Node.symbol)) then
20                 LiteralSrc = Literal2;
21             else
22                 LiteralTarg = Literal2;
```

```
            if ((LiteralSrc $\neq$ NULL) and (LiteralTarg $\neq$ NULL)) then
                LIST wargTargTmp = $\emptyset$;
                foreach ArgPos $\in$ Node.nodeInfo.warg do /* Read
                arguments from the source literal and try to find
                the same arguments in the target literal; for any
                matches, store the argument position in the target
                literal */
                    Argument = Fetch the argument from the source literal
                    at position ArgPos;
                    targetArgPos = Find the same argument than Argument
                    in the target literal, and read its position;
                    wargTargTmp = wargTargTmp $\cup$ {targetArgPos};
                if (length(wargTargTmp) == length(Node.nodeInfo.warg))
                then /* Verify list lengths, assign new warg list
                only if equal */
                    Node.nodeInfo.warg = wargTargTmp;
                    break ;       /* Continue with next predicate */
```

Algorithm 7 shows the check of reachability for two literals: This is very straightforward, as all reachable literals (the predicate symbols in the same cycle) have the same component number.

Algorithm 7: `bdi_isReachable(Literal1, Literal2, Graph)`

input : Two literals, and the predicate graph of the input clauses.
effect : Checks whether the first literal is reachable from the second by comparing their node component numbers.
output: True or false.

```
1 GRAPHNODE Node1 = getNode(Node1);
2 GRAPHNODE Node2 = getNode(Node2);
3 return (Node1.compNumber == Node2.compNumber);
```

Algorithm 8 represents the implementation to check general reachability of the given literal from a depth increasing clause. This is simply achieved by executing the sub-function `bdi_isReachable()` (representing the reachability check for two literals) in line 3 on every depth increasing literal from each depth increasing clause.

Algorithm 8: `bdi_isReachableFromDIC(Literal, Clauses, Graph)`

input : A literal, a list of depth increasing clauses, and the predicate graph of the input clauses.
effect : Checks if the given literal is reachable from a depth increasing clause.
output: True or false.

```
1 foreach Clause from Clauses do
2     foreach Literal2 from Clause.depthIncLiterals do /* Check
      reachability between Lit and literals from depth
      increasing literals list */
3         if (bdi_isReachable(Literal, Literal2, Graph)) then /* see
          Algorithm 7 */
4             return TRUE;
5 return FALSE;
```

Algorithm 9 checks the requirement for a clause that a non-empty watched argument list of a literal in Δ of the given clause requires all the watched argument lists of the literals from Γ of that clause either to be empty or equal to the non-empty list of the literal in Δ. This is implemented by looping over the positive literals of the given clause in line 1. If a non-empty watched argument list is encountered (line 3), the watched arguments of all negative literals in that clause are compared (lines 4-7) by means of the function `wargEqual()` (see Algorithm 11).

Algorithm 9: `bdi_wargEmptyOrEqual(Clause, Graph)`

input : A clause to examine, and the predicate graph of the input clauses.
effect : Checks the warg list for all literals from Δ. For non-empty lists, all atoms from Γ have either equal warg lists or the warg list is empty.
output: True or false.

```
1 foreach Literal $\in \Delta$ of Clause do
2     GRAPHNODE Node = getNode(Literal);
3     if (Node.nodeInfo.warg $\neq \emptyset$) then
4         foreach Literal2 $\in \Gamma$ of Clause do
5             GRAPHNODE Node2 = getNode(Literal2);
6             if ((Node2.nodeCompInfo.warg $\neq \emptyset$) and
              (wargEqual(Literal, Literal2) == FALSE)) then /* see
              Algorithm 11 */
7                 return FALSE;
8 return TRUE;
```

Algorithm 10 is a recursive function used to check the similarity of two given terms by traversing the tree representations of the terms. Additionally, the function collects the variables from the compared term (Term2) that are subject to further checks (which have to be carried out separately, see Algorithm 4, line 21). For storing the variables, a pointer to a list object (varcheck) is passed to the function such that the list is still available after leaving the function. The function traverses the tree representation of the terms one by one and subsequently examines the single components. The beginning in line 1 checks for equal terms which are trivially similar. Contrary, if two terms have different (top) symbols (checked in line 3), they can never be similar and the algorithms immediately stops by returning **FALSE**. Otherwise, the arguments of the two given terms are explored using a loop. In case of having a variable as the first (argument) term and a constant as the second (argument) term (line 7), the terms are similar but there are no variables to collect. Otherwise, if there is a variable or constant as the first term, and a variable as the second term (line 9), the first term is checked to be depth increasing (it is then marked with the depth increasing mark depthIncMark). If the depth increasing condition is satisfied, the second term (which is a variable) is then stored in the list *varcheck* in line 11. For the case of neither having variables nor constants, the present terms must be functional for which we go into recursion with the arguments in line 13 (and such traversing the tree). Eventually, in line 16, the function `bdi_termSimilarVarCollect()` returns **TRUE** if both argument lists of a function have been processed one by one and no argument is left, and **FALSE** otherwise.

The following examples depict the working principle of the previously described algorithm. Consider the terms $f(g(x))$ and $f(g(a))$ where x is a variable, and a a constant. For this input, the function symbols f and g are recursively examined to be equal one by one, and then on the most deeply argument level, the case in line 7 is entered with the terms Argument1 $= x$ and Argument2 $= a$, respectively.

Another example are the terms $f(g(x)), f(g(y))$. Let's assume that the variable x in the first term has been marked to be depth increasing. Then, after two recursion steps for examining the function symbols f and g, the case in line 9 is entered for the variables and the variable y is then collected because x at the same tree position than y has been marked to be depth increasing.

Algorithm 10: `bdi_termSimilarVarCollect(Term1, Term2, depthIncMark, varcheck)`

input : Two literals/terms to check for similarity, the depth increasing stamp and a pointer to a list for storing variables.
effect : Check two literals/terms for similarity and collect the variables that need to have depth=0 (this check has to be carried out separately for the collected variables after leaving this function).
output: True or false.

```
1  if (Term1 == Term2) then /* Equal terms are similar */
2      return TRUE;
3  else if (equalSymbols(Term1, Term2) == FALSE) then /* Compare symbols */
4      return FALSE;
5  else
6      foreach Argument1 ∈ getArgsList(Term1), Argument2 ∈ getArgsList(Term2) do
7          if (isVariable(Argument1) and isConstant(Argument2)) then
8              continue ;  /* Similar, but no variables to collect */
9          if ((isVariable(Argument1) or isConstant(Argument1)) and isVariable(Argument2)) then
10             if (Argument1 is marked with depthIncMark) then
11                 varcheck = varcheck ∪ {Argument2};
12             continue ;       /* no further subterm -> go to next argument */
13         else if (bdi_termSimilarVarCollect(Argument1, Argument2, depthIncMark, varcheck) == FALSE) then
14             return FALSE ;          /* not similar -> different symbols */
15     if (argument lists both have been processed one by one, and no argument is left) then
16         return TRUE;
17     else
18         return FALSE;
```

Algorithm 11 shows the implementation of the comparison between the watched argument lists of two literals. After the initialization in the lines 1-2, the length of the two literals' watched argument lists is compared at first in line 3. In order to be equal, also the number of watched arguments must be equal. In line 5 starts a loop for the pairwise comparison of the argument positions. At first, in line 6, it is verified whether the stored watched argument position is in fact an admissible argument position available for the present literal. Second, in line 8, the actual arguments referenced by their positions are read and compared to each other.

Algorithm 11: `bdi_wargEqual(Literal1, Literal2)`

```
input : Two literals.
effect : Verifies whether the arguments from the given literals that are
         specified by their corresponding watched argument lists, are
         similar (pairwise position checking).
output: True or false.

 1 GRAPHNODE Node1 = getNode(Literal1);
 2 GRAPHNODE Node2 = getNode(Literal2);
 3 if (length(Node1.nodeInfo.warg) $\neq$ length(Node2.nodeInfo.warg)) then
 4     return FALSE;
 5 foreach
   ArgPos1 $\in$ Node1.nodeInfo.warg, ArgPos2 $\in$ Node2.nodeInfo.warg do
   /* Check watched arguments */
       /* 1.  Check argument positions */
 6     if ((ArgPos1 $>$ length(getArgsList(Literal1))) or
       (ArgPos2 $>$ length(getArgsList(Literal2)))) then
 7         return FALSE;
       /* 2.  Check arguments */
 8     if (getArg(Literal1, ArgPos1) $\neq$ getArg(Literal2, ArgPos2)) then
 9         return FALSE;
10 return TRUE;
```

7 Solving SAP Authorization Problems with $\mathcal{BDI}$

The process of solving SAP authorization problems in first-order logic is generally the following: First, the problem needs to be decidable. This is easy if the problem (usually identified by its clausal structure) belongs to a known decidable class, i.e., all clauses satisfy the properties of the class. However, if there are (some) clauses violating these properties, it is maybe still possible to establish satisfiability by transforming and rewriting the violating candidates such that they become conform with the decidable class requirements afterwards. The last step in the solving process is eventually the application of the (problem) solving strategy which exists for all decidable classes.

Applying this process to the SAP authorization problems, the first step is the identification of the problem structure with respect to decidability. Fortunately, the SAP authorization problems have served as a base to develop the $\mathcal{BDI}$ class properties (Chapter 5), and thus, almost all clauses of these problems already satisfy the $\mathcal{BDI}$ requirements. The $\mathcal{BDI}$ properties check on a problem (instance) is carried out automatically with the theorem prover Spass including the extension described in Chapter 6. As a result of the check, the program outputs affected clauses violating the $\mathcal{BDI}$ criteria and additional information about the violated property for the respective clause.

The following clause represents the situation that the authorization value `STAR` matches every other required authorization value, denoted by the variable *xav*.

$$\begin{aligned}&\forall\ xu,\ xaon,\ xaof,\ xav\ .\\ &\quad \mathit{UserProfile}(\mathit{userProfileEntry}(xu,\ \mathit{authObj}(xaon,\ xaof,\ \texttt{STAR})))\\ &\rightarrow \mathit{Access}(xu,\ \mathit{authObj}(xaon,\ xaof,\ xav))\end{aligned}$$

It violates the variable subset property of $\mathcal{PVD}$, BDI-1 (i), BDI-2 (i) and thus $\mathcal{BDI}$. The fulfillment of the $\mathcal{BDI}$ requirements is achieved in this example by adding an additional monadic atom $\neg \mathit{Authorization}(xav)$ to the formula and a definition of the admissible constants for the predicate *Authorization* that can be substituted for the variable *xav* which doesn't change the meaning of the original formula.

Finally, if the (potentially modified) problem (instance) satisfies the $\mathcal{BDI}$ requirements, the solving strategy is applied to the problem. In the context of my SAP authorization experiments, I have used Hyper-Resolution with Factoring as the decision procedure to deal with the SAP example problems.

However, Ordered Resolution together with the two additional reduction rules making Ordered Resolution terminating (Section 5.4) should be used for larger real-world SAP authorization system instances because it is a more efficient and general decision procedure.

In practice, I have utilized the theorem prover Spass with Hyper-Resolution and Factoring on a server equipped with an Intel Xeon Quad Core X5460 CPU running at 3.16GHz, and 16 GB RAM.

I examined several SAP problem variants, all representing the formalization of the purchase process and typical real-world authorization setups for this process. The input problems differ in the number of users with (a) 500 users and (b) 5.000 users and the users also have different combinations of authorization values assigned using single and composite roles. Further, I have checked the business policies SoD and the four-eyes principle (Section 3.4) on the purchase process and the authorization setup.

The input file (a) with 500 users could be saturated in less than 1 second while the larger number of users in the input file (b) with 5.000 users (but with similar authorizations) increased the runtime to 1m34.795s.

The runtime results for the policy checks are similar: A violation of the four-eyes principle for an input file with 500 users could be detected in 1.033s and a similar policy check with 5.000 users results in a runtime of 1m44.318s. Also, it doesn't matter how many violations exist: A deliberate manipulation of the input problem to obtain more than one violation of a business policy yields a runtime of 1m44.472s which is almost identical to the run with only one violation. This is actually an obvious result because in all cases where the input problem is contradictory, Spass outputs a proof and shows the corresponding parts leading only to the first contradiction – no matter how many other contradictions exist. In order to detect all contradictions, it is necessary to fix the contradictory setup and to reapply Spass to find further business policy violations. In such cases, one could imagine to implement a reuse of an already saturated subset of the input problem for gain a speedup.

8 Application to the TPTP Library

The automated theorem prover Spass including the extension described in Chapter 6 has been applied to the problems of the TPTP Library, Version 6.1.0 [33] in order to detect potential new decidable problems and to find known undecidable problems "close" to the $\mathcal{BDI}$ class that may have only a few clauses responsible for the overall non-termination of the problem.

The current version 6.1.0 of the TPTP Library contains 21.172 problems. Our analysis of the problems ran on the Max Planck Institute's own server cluster. It consists of 128 Dell PowerEdge M610 Blade Servers, each with 2x Intel Xeon Quad Core E5620 running at 2.40 GHz, and 48 GB RAM. An internal script provided by the Information Services & Technology department (IST) of the Max Planck Institute is available which takes care of scheduling and submitting jobs to the cluster.

I have written a separate bash shell script for preprocessing that is called by the job scheduling script in place of directly calling Spass on every problem from the library. The preprocessing is used to rule out problems containing equality (because $\mathcal{BDI}$ does not deal with equality), as well as problems exceeding a size of more than 3 MB (the total size is calculated from the problem file itself and potentially included axiom files). If the problem satisfies the previous requirements, the extended version of Spass is called by the shell script with the originally provided arguments. These arguments are in our case:

- -Loops=0
  Stop after one main loop because we are only interested in analysis of the problem and we do not want to saturate it.

- -timelimit=3600
  A time limit of 3600 seconds per problem that is added in addition to -Loops=0 to prevent excessive input parsing. Sometimes, reading the problem file and building its clausal form by means of the integrated tool Flotter inside Spass took a very long time.

- -CNFRenaming=0
  This option causes the formulas not to be equivalently rewritten during proof search which is sometimes a benefit. Rewriting is not necessary because we do only problem analysis which doesn't need a saturation of the input.

- -PGiven
  Print the input formula as output.

During every run, the modified program SPASS outputs violations of the $\mathcal{BDI}$ conditions for any affected clause.

Afterwards, the job scheduling script calls a program “analyzer” which parses the SPASS output using regular expressions and writes the result(s) into a MySQL database. In our case, we are interested in the total number of clauses for a problem, to count all clauses violating any of the $\mathcal{BDI}$ conditions and additionally to count the clauses violating only BDI-1 or BDI-2 conditions. Of course, the problem name, its rating (specifying the difficulty with respect to ATP) and some other not important values at this place are also written into the database[1].

Having this data available in a database table provides an easy way for querying. For example, I have identified a total number of 21 undecidable problems “close” to the $\mathcal{BDI}$ class, with a rating > 0, having at least 5 clauses, and where only 1-4 clauses from the set of all clauses violate the BDI-1/BDI-2 conditions (trivial violations due to the variable condition of $\mathcal{PVD}$ have been ignored). The following Table 8 lists this query result. These problems may be altered without much effort into equivalent but decidable problems by replacing or rewriting the $\mathcal{BDI}$ violating clauses.

Another interesting question is to determine the native core for the class, i.e., how many problems from the TPTP Library directly satisfy the $\mathcal{BDI}$ requirements as they are provided – without modification or preprocessing: The result is a total number of 418 clauses.

In total there are 1.402 problems having up to 5 clauses violating the $\mathcal{BDI}$ requirements from which 1.221 problems contradict only the $\mathcal{PVD}$ conditions ($\mathcal{PVD}$ is included in $\mathcal{BDI}$). Consequently, 181 problems contradicting BDI-1/-2 properties remain. Further requiring a non-empty rating for the 181 problems reduces the number to 42.

The last query selects all problems where up to 50% of the clauses of the respective problem may violate the $\mathcal{BDI}$ requirements: It yields a total number of 2.950 problems.

[1]The analyzer script is a general tool also used by other people and therefore collects a whole bunch of data from the SPASS output. It has been extended to collect the data specific to the $\mathcal{BDI}$ class

Problem	**Rating**	**Number of Clauses**	**Violating BDI**	**Violating only BDI-1/-2**
SYN311-1	0.17	6	2	2
MSC013+1	0.56	11	1	1
MSC014+1	0.4	9	1	1
SET830-2	0.22	10	3	2
NUM284-1.014	0.33	6	2	1
NUM017-1	0.5	24	3	1
SWB011+2	0.25	10	2	1
SWB019+2	0.25	12	2	1
SWB022+2	0.18	30	2	1
SWB025+2	0.09	29	3	2
SWB012+2	0.18	31	3	3
SWV245-2	0.11	8	3	1
SWV293-2	0.11	9	3	3
SWV305-2	0.11	11	1	1
SWV286-2	0.11	8	2	2
SWV290-2	0.11	10	3	3
SWV246-2	0.11	8	3	1
SWV289-2	0.11	9	2	2
SWV300-2	0.11	14	3	3
SWV287-2	0.11	9	2	2
SWV247-2	0.22	10	3	1

Table 8.1: Problems “close” to the $\mathcal{BDI}$ class

9 Conclusion

I have presented the formalization of the authorization structure of the SAP system in first-order logic. The authorization analysis experiments revealed a specific structure of clauses that includes recursive definitions of predicates and a depth increasing term structure. Knowing in advance, if a first-order problem is decidable or not is essential for the practical application of first-order theorem proving tools. Because the so far studied decidable classes (for example, [6, 35, 12, 17, 19, 1]) do not allow the growth of the term depth for newly generated clauses by the respective resolution or superposition calculus, they were not suitable for the SAP authorization instances. This fact was the motivation to construct a new first-order decidable clause class, named $\mathcal{BDI}$ (Bounded Depth Increase). For the new clause class $\mathcal{BDI}$ defined in this thesis, the term structure of clauses belonging to the class is not restricted at all and predicates may have an arbitrary number of arguments. An overall bounded term depth is guaranteed by restricting the form of recursive definitions for predicates that occur in the clause set. For the $\mathcal{BDI}$ class any considered resolvent has a depth of at most $2n$ where n is the maximal depth of a clause in the initial set (Theorem 5.11). By requiring that all variables occurring in a positive literal of a clause also occur in a negative one of that clause, (positive) Hyper-resolution generates only ground clauses (Lemma 5.9), implying together the depth bound termination and therefore decidability of the $\mathcal{BDI}$ class (together with Factoring as a reduction rule). Thus, as Hyper-resolution terminates on $\mathcal{BDI}$, it enjoys the finite model property. In addition to the termination using Hyper-resolution, we showed that even any ordered resolution calculus with selection cutting off clauses with terms exceeding some a priori bound and variable condensing clauses exceeding a certain limit of different variables, decides the class. The ordered resolution calculus extended in this way can in fact efficiently decide properties for large $\mathcal{BDI}$ clause sets generated out of SAP authorization structures.

I have implemented the established criteria of the new class $\mathcal{BDI}$ into the theorem prover SPASS [37] (Chapter 6). This extension of SPASS makes it possible to automatically decide for a given problem, whether it satisfies the requirements of $\mathcal{BDI}$ or not. If the conditions of $\mathcal{BDI}$ are satisfied, then the prover is guaranteed to terminate. Otherwise, for a given problem, SPASS outputs the clauses violating the $\mathcal{BDI}$ requirements as well as additional information about the violated property for the respective clause.

I have run several experiments with problems representing the formalization of the purchase process (Section 4.2.3) together with a typical SAP authorization setup (Section 4.2.1) with different number of users. The input problems have been checked to conform to the $\mathcal{BDI}$ properties using the extended version of Spass. As a result of the checks, some clauses in the corresponding input file violated $\mathcal{BDI}$ properties, but after some little modifications all clauses eventually satisfied the $\mathcal{BDI}$ requirements. The example instances then have been solved using Spass only with Hyper-Resolution and Factoring. This demonstrates that even this (limited) decision procedure was able to solve the example instances but also points out that a more sophisticated decision procedure like Ordered Resolution with the proposed extension (Section 5.4) is needed if the problems become larger.

Additionally, the extended version of Spass has been applied to the problems of the TPTP Library [33] in order to detect potential new decidable problems and to find problems "close" to the $\mathcal{BDI}$ class having only a few clauses which are responsible for the overall non-termination of the problem. The result yields a total number of 21 undecidable problems with a rating > 0 having at least 5 clauses where only 1-4 clauses are allowed to violate the BDI-1/BDI-2 conditions (trivial violations due to the variable condition of $\mathcal{PVD}$ have been ignored). These candidates may be changed without much effort into equivalent but satisfiable problems by rewriting the $\mathcal{BDI}$ violating clauses and their dependencies. However, the exploration of the ways for rewriting or altering of these clauses is beyond the scope of this thesis and future work.

A very natural direction of research is the extension of the $\mathcal{BDI}$ class or the derivation of new deciable classes from it. An obvious modification would be to turn the variable conditions from succedent to antecedent and adopt the resolution strategy accordingly. Then, it is certainly possible to extend condition BDI-2 (Definition 5.7) to several depth growing argument positions. Furthermore, it might be possible to relax the requirement of equal watched argument lists (Definition 5.6 (v) and 5.7 (iv)) and replace it by a subset-like condition (because lists are ordered compared to pure sets). But this is not fully elaborated and future work.

Another area of future work might be the connection to the SAP system towards a fully automated system. The formulas representing the authorizations of such a system at a given point in time could be automatically extracted and transformed into an input problem for the theorem prover. Merged with the (manual) formalization of the processes and business policies (which don't need to be altered frequently after the initial setup) the extracted SAP authorization instance could be automatically verified to be contradiction-free.

Bibliography

[1] Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Transactions on Computational Logic 10(1), 4:1–4:51 (2009)

[2] Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press (1998)

[3] Bachmair, L., Ganzinger, H.: On restrictions of ordered paramodulation with simplification. In: Stickel, M. (ed.) 10th International Conference on Automated Deduction, Lecture Notes in Computer Science, vol. 449, pp. 427–441. Springer Berlin Heidelberg (1990), http://dx.doi.org/10.1007/3-540-52885-7_105

[4] Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. Research Report MPI-I-91-208, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany (September 1991), revised version in the Journal of Logic and Computation 4, 3 (1994), pp. 217–247

[5] Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 1, chap. 2, pp. 19–99. Elsevier and MIT Press (2001)

[6] Bachmair, L., Ganzinger, H., Waldmann, U.: Superposition with simplification as a decision procedure for the monadic class with equality. In: Gottlob, G., Leitsch, A., Mundici, D. (eds.) Computational Logic and Proof Theory, Third Kurt Gödel Colloquium. LNCS, vol. 713, pp. 83–96. Springer (August 1993)

[7] Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. Journal of Applied Logic 7(1), 58–74 (jan 2009)

[8] Börger, E., Grädel, E., Gurevich, Y.: The classical decision problem. Perspectives in mathematical logic, Springer (1996)

[9] Church, A.: A note on the entscheidungsproblem. pp. 40–41 (1936)

[10] Dershowitz, N.: Orderings for term-rewriting systems. Theor. Comput. Sci. 17, 279–301 (1982)

[11] Diestel, R.: Graph Theory, 4th Edition, Graduate texts in mathematics, vol. 173. Springer (2012)

[12] Fermüller, C.G., Leitsch, A., Hustadt, U., Tamet, T.: Resolution decision procedures. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. II, chap. 25, pp. 1791–1849. Elsevier (2001)

[13] Fermüller, C.G., Leitsch, A., Tammet, T., Zamov, N.K.: Resolution Methods for the Decision Problem, Lecture Notes in Computer Science, vol. 679. Springer (1993)

[14] Georgieva, L., Hustadt, U., Schmidt, R.A.: Hyperresolution for guarded formulae. J. Symbolic Computat 36, 2003 (2000)

[15] Georgieva, L., Hustadt, U., Schmidt, R.: A new clausal class decidable by hyperresolution. In: Voronkov, A. (ed.) Automated DeductionCADE-18, Lecture Notes in Computer Science, vol. 2392, pp. 260–274. Springer Berlin Heidelberg (2002), `http://dx.doi.org/10.1007/3-540-45620-1_21`

[16] Hay, D., Healy, K.A.: Defining Business Rules – What Are They Really? Final Report Revision 1.3, the Business Rules Group, formerly the GUIDE Business Rules Project (2000), `http://www.businessrulesgroup.org/first_paper/BRG-whatisBR_3ed.pdf`, [Online, accessed 2007-12-05: `http://www.businessrulesgroup.org/first_paper/BRG-whatisBR_3ed.pdf`]

[17] Hustadt, U., Schmidt, R.A., Georgieva, L.: A survey of decidable first-order fragments and description logics. Journal of Relational Methods in Computer Science 1, 251–276 (2004)

[18] ISACA: Management, Planning and Organization of IS, chap. 2, pp. 88–91. Information Systems Audit and Control Association (2005), [Online, accessed 2007-12-05: `http://www.isaca.org/Content/ContentGroups/Certification3/CRM_Segregation_of_Duties.pdf`]

[19] Jacquemard, F., Rusinowitch, M., Vigneron, L.: Tree automata with equality constraints modulo equational theories. In: Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4130, pp. 557–571. Springer (2006)

[20] Joyner, Jr., W.H.: Resolution strategies as decision procedures. J. ACM 23(3), 398–417 (Jul 1976), `http://doi.acm.org/10.1145/321958.321960`

[21] Kamin, S., L'evy, J.J.: Two generalizations of the recursive path ordering. (1980), unpublished note

[22] Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–267. Pergamon Press, New York (1970)

[23] Lamotte, M.: Analysis of Authorizations in SAP R/3. Master's thesis, Fachhochschule Trier (January 2008)

[24] Lamotte-Schubert, M., Weidenbach, C.: BDI: A new decidable first-order clause class. In: LPAR (short papers). pp. 62–74 (2013)

[25] Lamotte-Schubert, M., Weidenbach, C.: BDI: a new decidable clause class. Journal of Logic and Computation (December 2014)

[26] Lamotte-Schubert, M., Weidenbach, C.: BDI: A New Decidable First-order Clause Class. In: Mcmillan, K., Middeldorp, A., Sutcliffe, G., Voronkov, A. (eds.) LPAR-19. EPiC Series, vol. 26, pp. 62–74. EasyChair (2014)

[27] Lutz, C., Sattler, U., Tobies, S.: A suggestion for an n-ary description logic. In: Description Logics (1999)

[28] Nivelle, H.D.: Resolution decides the guarded fragment. (1998), iLLC report CT-98-01, University of Amsterdam, The Netherlands

[29] Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, chap. 6, pp. 335–367. Elsevier (2001)

[30] Post, E.L.: A variant of a recursively unsolvable problem. J. Symbolic Logic 12(2), 255–56 (1946)

[31] Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM 12(1), 23–41 (1965)

[32] SAP AG: SAP Library: Users and Roles (BC-CCM-USR) (2001), `http://help.sap.com/printdocu/core/Print46c/en/data/pdf/BCCCMUSR/BCCCMUSR.pdf`

[33] Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)

[34] Tarjan, R.: Depth first search and linear graph algorithms. SIAM Journal on Computing (1972)

[35] Weidenbach, C.: Towards an automatic analysis of security protocols in first-order logic. In: Ganzinger, H. (ed.) 16th International Conference on Automated Deduction, CADE-16. LNAI, vol. 1632, pp. 314–328. Springer (1999)

[36] Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 2, chap. 27, pp. 1965–2012. Elsevier (2001)

[37] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: Spass version 3.5. In: Schmidt, R.A. (ed.) CADE. Lecture Notes in Computer Science, vol. 5663, pp. 140–145. Springer (2009), `http://dblp.uni-trier.de/db/conf/cade/cade2009.html#WeidenbachDFKSW09`

[38] Wikipedia: Automated theorem proving – Wikipedia, the free encyclopedia (2014), `http://en.wikipedia.org/wiki/Automated_theorem_proving`, [Online, accessed 2014-12-30: `http://en.wikipedia.org/wiki/Automated_theorem_proving`]