Analysis of Reachability Properties in Communicating Authorization Policies

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
SIMONE FRAU
MSc. Computer Science, UniBo Bologna
born 19 December 1981
citizen of Italy

accepted on the recommendation of

Prof. Dr. David Basin, examiner
Prof. Dr. Joshua D. Guttman, co-examiner
Prof. Dr. Martin T. Vechev, co-examiner
Dr. Mohammad Torabi Dashti, co-examiner

2012
Abstract. Cryptographic protocols and authorization policies are two leading
techniques for securing software systems. The former are concerned with the en-
fforcement of secure communications in distributed systems, while the latter specify
which users under which conditions can be granted access to resources of a system.
The two have been mostly studied in isolation. Indeed, there are a number of al-
gorithms for deciding symbolic reachability in (a bounded number of sessions of)
cryptographic protocols [23,24,26,50,54]. These decidability results, however, are
not readily applicable to a more refined model of protocols in which the internal au-
thorization policies of the participants are non-trivial and indeed security-relevant.
Similarly, distributed authorization logics, such as [16, 27, 38], typically abstract
away from the communication events by assuming that all the policy statements
exchanged among the participants are simply (and solely) signed certificates. We
argue that these studies are inadequate for analyzing security as a whole, encom-
passing authorization logics, cryptographic protocols and the interface between the
two, i.e. how communication influences the policy inference, and vice versa. In-
deed, the need for integrated analysis of authorization logics and cryptographic
protocols has been recognized in the literature [3, 12, 40, 41, 51].

We present a formal language for specifying communicating authorization poli-
cies. Communicating authorization policies are distributed authorization policies
that communicate through insecure asynchronous media. The language allows us
to write declarative authorization policies. The interface between policy decisions
and communication events is specified using guards and policy updates. Guards
constrain the transmission of messages to the communication media to the sat-
sisfaction of (possibly negative) conditions on the policy and allow to make non-
deterministic choices. Policy updates modify the policy of a participant according
to the messages received; in particular, information can be introduced in the policy
and retracted from the policy. The attacker, who controls the communication me-
dia, is modeled as a message deduction engine. We give trace semantics to commu-
nicating authorization policies, and formulate a generic reachability problem. The
reachability problem subsumes the secrecy problem for security protocols [31] and
the safety problem for authorization policies [43].

We show that the reachability problem is decidable for a fragment of policies
specified in our formal language. The fragment, dubbed DC, is of practical rel-
ence, as demonstrated with examples. In particular, policies in DC belong to
a fragment of Horn theory, called AL, that allows infinite minimal models; this
singles out the fragment from many authorization languages with finite minimal
models. We give a decision algorithm for the reachability problem in specifica-
tions in DC. The algorithm extends the existing constraint reduction systems for
analyzing security protocols, by employing a novel proof search procedure for poli-
cies in AL and novel techniques for handling (symbolically) negative queries and
retracted policy statements.

Furthermore, we give a different proof technique for another decidable frag-
ment of communicating authorization policies, called DC_1, which is a strict subset
of the DC fragment. The proof technique is based on encoding the derivation of
policy statements with respect to the policy of a participant into message inference trees induced by the Dolev-Yao [31] deduction rules. The encoding benefits us in two ways: (1) it shows that the reachability problem for communicating authorization policies (in DC1) is reducible to the secrecy problem in security protocols and (2) it allows us to build upon the existing tools that have been originally developed for verifying security protocols.
Riassunto. Protocolli crittografici e politiche di autorizzazione sono due tecniche di primo piano nella protezione di sistemi software. I primi riguardano la realizzazione di comunicazioni sicure in sistemi distribuiti, mentre le seconde definiscono quali utenti (e sotto quali condizioni) possano accedere alle risorse di un sistema. Le due tecniche sono state studiate prevalentemente in maniera isolata. Infatti, vi sono svariati algoritmi per verificare il problema della raggiungibilità (di uno stato non sicuro) in protocolli crittografici (con un numero limitato di sessioni), e.g. [23, 24, 26, 50, 54].

Questi risultati, tuttavia, non sono applicabili a modelli di protocolli più dettagliati, in cui le politiche di autorizzazione interna dei partecipanti siano non banali e in effetti rilevanti per la sicurezza. Analogamente, logiche di autorizzazione distribuite, come [16, 27, 38], tipicamente astraggono la comunicazione tra partecipanti dando per assunto che questi scambino esclusivamente certificati autentic. In questo lavoro, dibattiamo che questi risultati siano inadeguati per analizzare proprietà di sicurezza nel loro complesso, includendo quindi sia le logiche di autorizzazione sia i protocolli di sicurezza e l’interfaccia tra i due, ovvero come la comunicazione influenzi le politiche di autorizzazione, e viceversa. Difatti, la necessità di un’analisi integrata di logiche di sicurezza e protocolli crittografici è stata riconosciuta nella letteratura [3, 12, 40, 41, 51].

In questa tesi, presentiamo un linguaggio per la specifica di politiche di autorizzazione comunicanti. Le politiche di autorizzazione comunicanti sono sistemi di politiche di autorizzazione distribuite che comunicano attraverso un medium non sicuro. Il linguaggio permette di specificare politiche di autorizzazione in stile dichiarativo. L’interfaccia tra le decisioni di autorizzazione e gli eventi della comunicazione viene specificata attraverso guardie e aggiornamenti. Le guardie restringono la trasmissione di messaggi al medium di comunicazione imponendo il soddisfacimento di condizioni di autorizzazione (possibilmente negative) e permettendo di compiere scelte non deterministiche. Gli aggiornamenti modificano le politiche di un partecipante a seconda del messaggio ricevuto; in particolare, le informazioni possono essere introdotte o rimosse dalle politiche. L’attaccante, che controlla il medium della comunicazione, viene modellato attraverso un sistema di deduzione di messaggi. Abbiamo attribuito alle politiche di autorizzazione comunicanti una semantica di tracce, ovvero di esecuzioni concrete del sistema, e formulato un problema di raggiungibilità generico. Il problema della raggiungibilità sussiste il problema della segretezza nel contesto di protocolli di sicurezza [31], e il problema dell’accesso sicuro nel contesto delle politiche di autorizzazione [43].

In questa tesi, mostriamo che il problema della raggiungibilità è decideribile per una classe di politiche di autorizzazione comunicanti. La classe, denominata DC, è di interesse pratico, come dimostrato dagli esempi. In particolare, le politiche in DC appartengono ad un sottosistema della teoria di Horn, chiamato AL, che permette di esprimere politiche con modelli minimi di cardinalità infinita; questo lo distingue da molti linguaggi di autorizzazione che permettono di esprimere esclusivamente politiche con modelli minimi finiti. Per dimostrare la decidibilità della classe DC forniamo una procedura per la decisione del problema della raggiun-
gibilità in specifiche appartenenti a DC. L’algoritmo estende procedure esistenti per la verifica della raggiungibilità nel contesto di protocolli di sicurezza, impiegando una nuova procedura di ricerca di dimostrazioni per politiche appartenenti a AL, e tecniche originali per il trattamento (simbolico) di decisioni negative e della ritrazione di asserzioni dalle politiche.

Inoltre, forniamo una tecnica differente di verifica del problema della raggiungibilità per un’altra classe di politiche di autorizzazione comunicanti, denominata DC₁, che è un sottoinsieme stretto della classe DC. La tecnica di verifica si basa sulla codifica delle asserzioni delle politiche di un partecipante in messaggi deducibili da un modello di attaccante Dolev-Yao [31]. Questa codifica è vantaggiosa sotto due aspetti: (1) mostra che il problema della raggiungibilità per politiche di autorizzazione comunicanti (appartenenti a DC₁) è riducibile al problema della segretezza nel contesto dell’analisi di protocolli di sicurezza, e (2) ci permette di estendere procedure già esistenti e originariamente ideate per l’analisi di protocolli di sicurezza.
## Contents

1 Introduction 1
   1.1 Contributions 3
   1.2 Related work 7
   1.3 Structure 8

2 Formalizing Communicating Authorization Policies (CAPs) 11
   2.1 Syntax 11
   2.2 Semantics 14
   2.3 The reachability decision problem REACH 17
   2.4 Two sorts: messages and infons 18

3 Examples of CAPs 21
   3.1 OAUTH, RBAC, and transitive attributes 21
   3.2 Trust relations 22
   3.3 Retraction 25
   3.4 Variables originating in negative guards or updates, and SoD 28

4 DC: A class of decidable CAPs 31
   4.1 The Dolev-Yao attacker model 31
   4.2 The AL policy theories 32
   4.3 The DC class of CAPs 33
   4.4 Examples of DC specifications 34

5 A decision procedure for REACH for CAPs in DC 47
   5.1 Preliminaries 51
   5.2 Generating constraint systems from interleavings 52
   5.3 Solving attacker constraints 55
   5.4 Solving positive policy constraints 61
   5.5 Solving negative policy constraints 69
   5.6 Enforcing the provisions 75
   5.7 Symbolic treatment of retraction 76
   5.8 Correctness of the decision procedure for REACH in CAPs in DC 92
6 Reducing reachability in CAPs to secrecy
6.1 The DC₁ class of CAPs
6.2 A decision procedure for REACH for CAPs in DC₁
7 Conclusion
Chapter 1

Introduction

Cryptographic protocols and authorization policies are two dominant approaches for securing software systems. The former are concerned with the enforcement of security-related properties of the communication in distributed systems, while the latter specify which users under which conditions can be granted access to resources of a system.

A vast body of research has been devoted to the analysis of cryptographic protocols in the last three decades, focusing on a number of security properties (e.g., secrecy, authentication and many others [20, 29, 49]) and several different threat models [26, 31]. Particular attention has been paid to symbolic models, initiated with Dolev and Yao [31], that are amenable to (under certain restrictions, automated) formal analysis [48,52,54]. Similarly, formalizations of authorization policies, with a focus on safe access to system resources, have also been extensively studied: starting from the early HRU access matrix [43], to more recent declarative, logic-based languages like BINDER [27], SECPAL [16] and DKAL [38]. However, these techniques have been mostly studied separately. Thus, they do not account for the systems in which authorization logics and security protocols influence each other.

In this thesis, we look at systems where authorization logics and security protocols are employed together, but do not rely on the security of the communication or of the authorization policies alone: in these systems, interaction between the communication and the policies is integral to the correct execution of the system. The separation between the communication level (i.e. the interaction between participants of the system through a medium) and the policy level (i.e. the policies enforced by the participants of the system) can be a useful abstraction for better understanding each of these levels. The modelers can focus, for instance, on the logics of their access control policies and not worry about specifying the exact routes through which the policy statements travel. This abstraction however obscures when messages become policy statements, how policy statements are represented as messages in the communication, and whether there is a place for misinterpretation. This is the main thrust of our argument that the abstraction is
inadequate for analyzing security as a whole. While maintaining the separation between the communication and policy levels, we believe that a more refined security analysis can be performed when a precise definition of the interface between the two levels is given; i.e. how communication influences the policy inference, and vice versa.

In this thesis we present a language for the formal specification of communicating authorization policies. Communicating authorization policies are distributed authorization policies that communicate through insecure asynchronous media. Consider, as an example, the following scenario:

The human resources department of a hospital runs a policy engine to administer information about the personnel of the hospital. A doctor, Bob, asks the human resources department for a vacation leave; he does so by sending a message that is meant to indicate this request. The human resources department evaluates the request against its policy engine to verify that Bob is indeed eligible for vacation and to find a doctor that can substitute for Bob; the chosen substitute would be delegated with the access rights of Bob, in Bob’s absence. If these conditions are met, the human resources department sends a message to Bob, indicating that he has the leave to go on vacation.

When abstracting away the concrete specification of how messages are interpreted in terms of policy statements, and of how policy statements are represented as messages, a number of issues are unclear. For instance, can the scenario be executed (“Can the vacation leave be granted to Bob?”)? Can an attacker exploit a faulty interpretation of the request message to obtain an unauthorized delegation of rights (“Did Bob really request a vacation leave?”)? Can an underapproximated representation of the policy statement granting the vacation leave be forged by an attacker (“Did the human resources department really grant the vacation leave?”)? These questions can be considered as reachability problems, which ask whether or not, given a system and a property $\phi$, there exists a state in the execution of the system in which $\phi$ holds; for instance, $\phi$ could be “Bob obtains the grant to leave for vacation”.

Reachability problems in such systems depend on the interplay between the communication and the authorization policies. For this reason we have chosen communicating authorization policies as integrated models. The need for integrated analysis of authorization logics and cryptographic protocols has been recognized in the literature. For instance, the trust management model of [40, 41] extends distributed systems specifications with formulas that participants guarantee or rely on, and includes a trust management system engine that defines the policy of the system. Another example is [34], which extends the spi calculus with authorization annotations, and proposes a dependent type system to statically enforce the correctness of a specification.

Communicating authorization policies cannot be readily analyzed by existing algorithms for the analysis of security protocols or of authorization policies. Indeed, there are a number of algorithms for deciding symbolic reachability in (a bounded number of sessions of) cryptographic protocols; see [23, 24, 26, 50, 54].
However, these algorithms invariably assume that the participants are specified as sequences of “send” and “receive” events. The internal computations of the participants are limited to cryptographic manipulation of the variables that are sent and received [23, 26, 50, 54], and checking equality between them [24, 26]. These decidability results are not applicable to a more refined model of protocols in which the internal computations of the participants are non-trivial and security-relevant. Even verifying the simple scenario described above with these algorithms would require to manually approximate the internal computations of the human resources department, e.g. by abstracting its policy engine and its evaluation semantics. Similarly, distributed authorization logics, such as [16, 27, 38], abstract away from the communication events by assuming that all the policy statements exchanged among the participants are (solely) signed certificates.

1.1 Contributions

In this thesis, we provide a framework for the specification and analysis of communicating authorization policies; that is, systems in which policy decision points (PDPs) communicate with each other by exchanging messages over insecure media. The policies of such PDPs change due to receive events (e.g. upon receiving a public key certificate), and they in turn constrain the communication events (e.g. access tokens are sent only to the principals whose credentials have not been revoked). As PDPs communicate over insecure media, attacks may occur when, for instance, expired certificates are replayed, certificate revocation lists are delayed, messages are tampered with, etc. We let the attacker be in direct control of the communication media. This view is motivated by the workings of security-sensitive distributed services, such as federated identity management systems (OAUTH [33], etc.). We define a formal language for specifying communicating authorization policies, and give algorithms for deciding reachability for large classes of such policies.

In our formalism, we model communicating authorization policies as a finite number of processes. Intuitively, a process represents a PDP. Each process consists of a finite number of threads that share a policy. A threads is a finite sequence of communication events. Threads run in parallel and exchange messages with the threads of other processes over insecure media. The policy of a process is a (declarative) program which models the shared authorization policy that the threads of the process evaluate.

Threads communicate by sending and receiving messages in an asynchronous message passing environment. Each send event is constrained by a guard, and each receive event results in an update of the policy of a process. Intuitively, guards and updates belong to the policy level, as opposed to send and receive events which constitute the communication level (see Figure 1.1). In anthropomorphic terms, processes “think” at the policy level, and “talk” at the communication level.

From an operational point of view, guards are statements that, if derivable from
Figure 1.1: Two processes (i.e. policy decision points), each with two threads.

the policy of a process, allow the process to perform a corresponding send action (cf. Dijkstra’s guarded command language [30]). Updates are also statements at the policy level. When a process receives a message in one of its threads, it updates its policy correspondingly. Intuitively, updates associate meanings to the messages a process receives in terms of statements at the policy level. For example, a signed X.509 certificate sent by a certificate authority means that the authority endorses the public key and its owner, mentioned in the certificate. In this sense, the notion of updates is similar to the assumptions that are relied upon after receiving a message, in the trust management model of Guttman et al. [41]. In our language, however, updates can also include negative policy statements, that one could use to model revocation of policy statements, cf. DYNPAL [15].

We assume an all-powerful attacker who is in direct control of the insecure communication media; see Figure 1.1. In fact, the messages the PDPs exchange are passed through the attacker. This is a common (worst-case scenario) assumption in the literature. The message inference capabilities of the attacker may reflect, e.g., the Dolev-Yao threat model [31]. The attacker can indirectly manipulate the policies of the participating PDPs by, e.g., sending tampered messages which affect the update statements.

The main contributions of this thesis are:

(1) We present a formalism for specifying communicating authorization policies, and their hostile environment. A typical specification in our language consists of three components: communication level events, policy level decisions, and the interface between the two. As the interface between the levels is explicitly present in the specifications, a more precise security analysis of distributed policies becomes possible. This singles out our specification language from the formalisms which focus on either the communication or the policy level, and hence neglect their interactions.

In our formal language, the behaviour of the participants is described by the sequences of events that they perform. Unlike typical specifications of security protocols (e.g. [23, 24, 26, 50, 54]), that are limited to “send” and
“receive” events, participants in specifications written in our language can execute events of the following two types:

- Guarded send events, where the transmission of a message to the communication media is constrained to the satisfaction of a guard. Guards allow a process to ascertain (possibly negative) policy statements and to make non-deterministic choices.

- Receive events leading to an update of the policy of the participant; that is, messages received by a process are interpreted in terms of policy statements. In particular, the policy statements can be negative, thus leading to the retraction of previously asserted statements.

The policy of the participants are specified as finite sets of Horn clauses; our policy specification language can therefore be considered related to several logic-based authorization languages like Dkal [38], secpal [16], Dyn-Pal [15] and Binder [27], and logic programming languages like DataLog [2] and Prolog. Similarly, also the capabilities of the attacker are expressed as a finite set of Horn clauses, which allows for specification of a variety of different threat models, e.g. [26, 31].

We define a generic reachability problem for communicating authorization policies. The reachability problem subsumes the secrecy problem for security protocols [31] and the safety problem for authorization policies [43]. The reachability problem is undecidable in general, even when assuming a finite bound on the number of participating services. This is due to the computational power of logic programs; see [28, 42].

(2) We give an algorithm to decide reachability in a fragment of communicating authorization policies, specified in our formal language. The fragment, called DC, is presented in detail in Chapter 4. In a DC specification the message composition and decomposition capabilities of the attacker reflect the Dolev-Yao threat model [31], while the policies of the participants belong to a subset of Horn theory called AL. The AL fragment is sufficiently expressive for modelling, e.g., RBAC systems with role hierarchy and the trust application and trust delegation rules à la Dkal, which are the core of many distributed authorization logics [16, 27, 38]. The trust application and trust delegation rules intuitively state that

- Trust application: If Ann trusts Mike on statement \( f \), and Mike says \( f \), then Ann believes \( f \) holds.

- (Transitive) Trust delegation: If Ann trusts Mike on statement \( f \), and Mike delegates the right to state \( f \) to, e.g., Piet, then Ann trusts Piet on statement \( f \).

The decision algorithm we give for communicating authorization policies in DC is a modular constraint reduction procedure: each possible execu-
tion of the system is translated into a set of (symbolic) constraints modeling symbolic deduction problem. The procedure reduceDY, which solves Dolev-Yao symbolic deduction problems, builds upon the constraint reduction procedure of [23]. Constraints modeling positive policy symbolic deduction problems are solved by the proof search procedure ps for theories in AL; see Section 5.4. Constraints modeling negative policy symbolic deduction problems are solved by the procedure solveNeg, see Section 5.5. Also, a novel technique to treat retraction of policy statements symbolically is integrated in the algorithm; cf. Section 5.7. A prototype implementation of our decision algorithm is publicly available at http://www.infsec.ethz.ch/people/fraus.

(3) We give a different proof technique for another decidable fragment of communicating authorization policies, called DC1, which is a strict subset of the DC fragment. In DC1 specifications the attacker model is fixed to the Dolev-Yao attacker. The policies of the participants, in a DC1 specification, belong of a subset of Horn theory centered around the trust application and trust delegation rules (see above), and type-1 theories (formally defined in Section 6.1). The policy theories expressible in DC1 are a strict subset of theories in AL; for instance, recursive policy rules (see Section 3.1), which can be included in AL theories, are not permitted in type-1 theories. Furthermore, in DC1 specifications negative statements are not allowed in guards and in updates, and guards do not allow for non-deterministic choices.

To prove the decidability of systems belonging to the DC1 fragment, we encode the derivation of policy statements with respect to the policy of a participant into message inference trees induced by the Dolev-Yao deduction rules. The encoding benefits us in two ways: (1) it shows that the reachability problem for communicating authorization policies (in DC1) is reducible to the reachability problem in security protocols and (2) it allows us to build upon the existing tools that have been originally developed for verifying security protocols. In particular, we have extended the constraint solver of Millen and Shmatikov [50] in Prolog to validate communicating authorization policies; an implementation is available at http://www.infsec.ethz.ch/people/fraus.

The fragments DC and DC1 are of practical relevance. We demonstrate this through a number of intuitive examples in this thesis; see Chapter 3. Furthermore, several industrial service-oriented architectures studied in the context of AVANTSSAR [5] (The EU Project on Automated Validation of Trust and Security of Service-oriented Architectures) fall in the DC fragment. In Section 4.4, we discuss and formalize two extended examples based on some of the scenarios of [5]; namely, an example of an on-line car registration service procedure, and an example of an electronic health record system.
1.2 Related work

The closest related works are (1) dynamic authorization logics, and (2) security protocols annotated with authorization constraints.

(1) Dynamic authorization logics. Existing distributed authorization logics, such as [1, 16, 19, 27, 38], cannot express the dynamic aspects of distributed policies. We, in contrast, model communicating authorization policies which change due to communication events.

Our formalism allows for modeling the dynamic aspects of distributed (i.e. communicating) authorization policies. This is the fundamental difference between our formalism and the dynamic authorization logics that confine their analysis to a centralized policy decision point, such as [8, 22, 32, 36, 47].

We decide reachability in communicating authorization policies by taking into account the “low-level” cryptographic protocols that implement the policies, and also the interface between the low-level protocols and the policies. This is in contrast to the dynamic authorization logics that model the causes and effects of policy changes at the same level of abstraction as the policy, such as [15, 17]. These logics hence abstract away the mechanisms through which policy decision points communicate and possibly influence each other. For instance, the notion of updates in our language is close to the notion of effects in [15, 17]. We, however, do not associate effects to policy decisions; rather, effects (or updates) are associated with receiving messages. What we call a guarded send event in our formalism has no counterpart in [15, 17]. In our decision algorithm, we not only take into account the asynchronous communications among the policy decision points, but also we can explicitly model the capabilities of the hostile entity (e.g. the Dolev-Yao [31] attacker) that controls their communications.

In [3], Alvaro et al introduce a declarative language, Dedalus, based on Datalog with negation [56] that incorporates a logical notion of time as an attribute of predicates. The language allows to express important features of distributed systems such as mutable state and asynchronous communication. Despite the infinite number of derivations possible due to the time attribute, the authors extend the classical notions of safety and stratification of Datalog with negation to Dedalus. The notion of safety entails a finite minimal model in each state (i.e., modulo the time attribute), whereas our models can be infinite; e.g., we support transitive rules like the Trust Delegation rule $\text{tdOn}(a, \text{tdOn}(b, x))$ $\leftarrow$ $\text{tdOn}(a, x)$ (see, e.g., Section 3.2).

In [53], the authors use a variant of deontic logic to reason about changes in authorization policies. They assume that the policies are explicitly given as a finite-state Kripke model. We, in contrast, model systems that generally induce infinite state spaces.

(2) Security protocols annotated with authorization constraints.

Our formalism is related to the body of research on annotating security proto-
cols with authorization constraints [4, 9–11, 34, 41]. None of these works gives an algorithm for deciding reachability. Some notable exceptions are [12] and [51]. In [12], Barletta et al present a framework based on a temporal extension of first-order logic that allows to formalize service-oriented systems; specifications in the framework include a Workflow level and a Policy Management level, similar to our communication and policy levels. A decision procedure for a fragment of specifications is given; however, no active attacker is included in their models. In [51], Mödersheim gives a decision procedure for a fragment of ASLan (the AVANTSSAR Specification Language, cf. [6]). The language consists in an infinite-state transition system and a Horn theory evaluated locally in every state. The fragments of Horn theory considered in [51] is incomparable to AL. The transition system allows to pose negative queries and to retract facts from the state. However, variables in negative queries and retracted facts are bound by positive queries; we, in contrast, allow for free variables in negative guards and negative updates that are interpreted as universally quantified. Furthermore, only “explicit” predicates (i.e., explicitly represented in the state) can appear in negative queries in [51], whereas we allow also “implicit” predicates (i.e., obtainable through deduction) in negative queries.

A remark is due on the semantics of retraction of facts in our framework, with respect to other semantics of retraction in related works. We remark that retraction in our framework only affects extensional facts; intensional facts, unless derived (only) from the retracted extensional facts, are not influenced. We note that also in [51] only extensional facts can be retracted. Similarly, in the Prolog logic programming language [45] only extensional facts can be retracted; however, states in Prolog are represented as multisets of atoms; consequently, facts can be retracted as many times as their multiplicity. In Dedalus [3], the semantics of retraction is modeled explicitly as part of the program: through use of “persistence until retraction” rules a modeler can define what facts are affected by retraction, and how the retraction might be propagated.

1.3 Structure

In Chapter 2 we formally define the syntax and semantics of our specification language for communicating authorization policies, respectively in Sections 2.1 and 2.2. In Section 2.3, we define a reachability problem for specifications in the framework. The choice of a two-sorted term algebra for the framework is discussed in Section 2.4. In Chapter 3, a number of examples are presented to demonstrate the expressivity of our framework. In Chapter 4, we introduce a decidable fragment of communicating authorization policies, called DC. We demonstrate the practical relevance of the fragment by giving formalizations for two extended examples, in Section 4.4. In Chapter 5 we present a decision procedure for the reachability problem for specifications in DC. The procedure invokes a number of subroutines,
that are discussed in detail in dedicated sections. In particular, in Section 5.4 we give a proof search procedure for \( \text{AL} \) theories; in Section 5.4 we give a procedure for solving negative constraints; in Section 5.7 we discuss a technique to treat retraction of facts symbolically. In Chapter 6 we present another decidable fragment of communicating authorization policies, called \( \text{DC}_1 \). In Section 6.2 we give a decision procedure for the reachability problem in specifications in \( \text{DC}_1 \), that is based on reducing the problem to the secrecy problem in security protocol analysis. Finally, in Chapter 7, we draw some conclusions on the work presented.

**Publications.**


Chapter 2

Formalizing Communicating Authorization Policies (CAPs)

We formally define the syntax (Section 2.1) and semantics (Section 2.2) of our language. In Section 2.3, we define a generic reachability problem for communicating authorization policies. Finally, in Section 2.4, we discuss the type discipline we consider in this thesis.

2.1 Syntax

A (multi-sorted) signature is a tuple \( (S, \Sigma, V) \), where \( S \) is a finite set of sorts, \( \Sigma \) is a countable set of function symbols, and \( V \) is a countably infinite set of variables. The sets \( \Sigma \) and \( V \) are disjoint. As usual, each element of \( V \) has a sort associated to it, and the set of variables for each sort \( s \), denoted \( V_s \), is countably infinite. The sorted term algebra induced by the signature is denoted \( T_{\Sigma(V)} \). In the following, we only consider two-sorted signatures: terms are either a message term or an infon term. The word “infon” is borrowed from [38]. In Section 2.4 we motivate our choice of typing.

For a term \( t \), \( \text{var}(t) \) is the set of variables appearing in \( t \). Term \( t \) is ground iff \( \text{var}(t) = \emptyset \); hence, \( T_{\Sigma(\emptyset)} \) is the set of ground terms. A message is a ground message term. A fact is a ground infon term.

We fix a signature \( (S, \Sigma, V) \). An event is either a send event or a receive event. A send event is of the form \( g \triangleright \text{snd}(m) \), where \( g \) is a guard and \( m \) is a message term. A receive event is of the form \( \text{rcv}(m) \triangleright u \), where \( m \) is a message term and \( u \) is an update. A guard \( g \) is of the form \( g_\exists g_\not\exists \), where \( g_\exists \) and \( g_\not\exists \) are disjoint finite sets of infon terms. An update \( u \) is of the form \( u_+ u_- \), where \( u_+ \) and \( u_- \) are disjoint finite sets of infon terms. We refer to \( g_\exists \) (respectively, \( u_+ \)) as a positive guard (respectively, positive update), and to \( g_\not\exists \) (respectively, \( u_- \)) as a negative guard (respectively, negative update). To improve readability, we may superscript negative guards and negative updates with \( \neg \).

In systems specified in our formal language, the behaviour of each process (i.e.
a participant of the system, formally defined later in the section) is described by
the sequences of events that the process performs. Each event can be considered as
having two parts, one belonging to the communication level, the other belonging
to the policy level.

The communication level component of an event describes the interaction be-
tween the processes of the system. This consists in sending messages to, and re-
ceiving messages from, a communication medium, as it is usual in asynchronous
message passing environments. Hence the communication level is closely related
to the models that are used for specifying security protocols, e.g. [23, 31, 50, 55].
The policy level component of an event defines the policy decisions that processes
make, how the decisions affect the communication and, in turn, how communica-
tion influences future decisions. This is what singles out our formal language from
other formalisms for specifying security protocols.

More in detail, guards constrain the execution of send events to the satisfac-
tion of authorization requirements. A guard $g$ consists of two syntactically dis-
joint finite sets of infon terms $g_\exists$ and $g_\exists$, representing respectively the positive
and negative authorization queries of the guard: the corresponding send event is
performed if the positive query is satisfied and the negative queries are not. For
instance, a guard $\{is\_pk\_of(k, piet)\}\{is\_revoked(k)\}$, where the policy state-
ment $is\_pk\_of(k, piet)$ denotes that $k$ is a public key of $piet$ and the policy state-
ment $is\_revoked(k)$ denotes that $k$ has been revoked, is satisfied when the state-
ment $is\_pk\_of(k, piet)$ can be ascertained and the statement $is\_revoked(k)$ can be
refuted. The precise semantics of guards is given in Section 2.2.

Updates, performed upon receiving a message from the network, allow the
processes of the system to make changes to their policies. These updates usually
represent the “meaning” that processes give, at the policy level, to the messages
received at the communication level. For instance, a process that receives the mes-
 sage sig(sk(ann), pk_cert(piet, k)) (i.e., a certified key $k$ of $piet$, signed by $ann$)
can interpret the message as said(ann, is\_pk\_of(k, piet)), in terms of policy state-
ments; here, said($x$, $y$) denotes that agent $x$ has said statement $y$, and the statement
is\_pk\_of(k, piet) denotes that $k$ is a key of $piet$. An update $u$ consists of two dis-
joint set of infon terms, the positive updates $u_+$ and the negative updates $u_-$. The
positive updates $u_+$ are added to the policy statements of the process; this notion is
similar to the assumptions which are relied upon after receiving a message, in the
trust management model of [41]. The negative updates $u_-$, instead, are removed
from the policy statements of the process; this feature is in particular useful for
modeling retraction of previously assumed credentials. We come back to guards
and updates shortly.

A thread $t$ is a finite sequence of events $t = e_1, \cdots, e_n, n \geq 0$. For a vari-
 able $V$ appearing in thread $t$, we say the event $e_i$, with $i \in \{1, \cdots, n\}$, is the origin of $V$ in $t$ if $i$ is the smallest index of the events in which $V$ appears. We
only consider threads $t$ that satisfy the origination property, i.e., for any variable $V$
 appearing in $t$ the following holds:
• if the origin of $V$ is the event $\text{rcv}(m) \triangleright u_+ u_-$, then either $V$ appears in $m$, or $V$ appears in $u_-$ and $V$ does not appear elsewhere in the thread;

• if the origin of $V$ is the event $g_3 g_3 \triangleright \text{snd}(m)$, then either $V$ appears in $g_3$, or $V$ appears in $g_3$ and $V$ does not appear elsewhere in the thread.

Variables in a specification can originate at four different points: in a received message, in a positive guard, in a negative guard, or in a negative update. Variables that originate in messages received from the network are under the control of the attacker, who can instantiate them with any of the terms that he knows (or that he can construct). This corresponds to the origination assumption common in security protocol models, e.g. [23, 50]. Variables originating in positive guards are under the control of the process that evaluates the guard: they can assume any value under which the guard evaluates to true. Variables originating in received messages and in positive guards denote therefore definite values, and they can be referred to in the remainder of the thread. In contrast, variables originating in negative guards or in negative updates are interpreted as universally quantified; e.g., a guard $\{\{\text{is\_revoked}(K)\}\}$, with $K$ being a free variable, is interpreted as the first-order formula $\forall K. \neg \text{is\_revoked}(K)$. This point becomes clear in our execution semantics, Section 2.2. Because the variables that originate in negative guards or updates are not instantiated with any value, they can not be referred to in the remainder of the thread. See the examples of Section 3.4 for further explanations.

A process $\pi$ is a tuple $(\eta_\pi, \Omega_\pi, I_\pi)$, where $\eta_\pi$ is a finite set of threads, $\Omega_\pi$ is a finite set of facts, called the extensional knowledge or extensional policy of $\pi$, and $I_\pi$ is a finite set of infon clauses, called the policy of $\pi$. An infon clause (otherwise called infon rule) is of the form $a \leftarrow a_1, \ldots, a_n$, with $n \geq 0$ and $a, a_1, \ldots, a_n$ being infon terms.

An attacker model $A$ is a pair $(\Omega_A, I^A)$, where $\Omega_A$ is a finite set of messages, called the extensional knowledge of the attacker, and $I^A$, referred to as the policy of the attacker, is a finite set of message clauses. A message clause (otherwise called message rule) is of the form $m \leftarrow m_1, \ldots, m_n$, with $n \geq 0$ and $m, m_1, \ldots, m_n$ being message terms.

The attacker is thus identified in our model with his extensional knowledge (which stores the set of the messages observed by the attacker) and his policy (which encodes the attacker’s deduction capabilities over messages). Moreover, the attacker is able to send and receive messages; these capabilities are reflected in the execution model, defined in Section 2.2.

We are now ready to define communicating authorization policies, hereafter referred to as CAPS. A CAP is a tuple $(\langle S, \Sigma, V \rangle, \pi_1, \ldots, \pi_\ell, A)$, where $(\langle S, \Sigma, V \rangle)$ is a signature, $\pi_1, \ldots, \pi_\ell$ are (honest) processes and $A$ is an attacker model, all defined using the signature $(\langle S, \Sigma, V \rangle)$. To avoid trivial name clashes, variables appearing in different threads are assumed to be distinct.
2.2 Semantics

We start with defining substitutions. Fix a signature \((\mathcal{S}, \Sigma, \mathcal{V})\). A substitution \(\sigma\) is a sorted mapping from a finite set of variables to \(\mathcal{T}_{\Sigma(V)}\). Substitutions are written as \(\sigma = \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}\). The domain of \(\sigma\), denoted \(\text{dom}(\sigma)\), is the set \(\{v_1, \ldots, v_n\}\) and the range of \(\sigma\), denoted \(\text{ran}(\sigma)\), is the set of variables that appear in \(t_1, \ldots, t_n\). We assume that every substitution is idempotent, i.e., the domain and the range of any substitution are disjoint. Substitutions are homomorphically extended to \(\mathcal{T}_{\Sigma(V)}\). The application of substitution \(\sigma\) to term \(t\) is denoted \(t\sigma\). A substitution \(\sigma\) is grounding for term \(t\), if and only if \(t\sigma\) is ground. For a finite set of terms \(T = \{t_1, \ldots, t_n\}\), we write \(T\sigma\) for the set \(\{t_1\sigma, \ldots, t_n\sigma\}\). The definition of grounding substitution extends to sets of terms as expected: a substitution \(\sigma\) is grounding for a finite set of terms \(T\) if and only if all the elements of \(T\sigma\) are ground.

Let \(H\) be a finite set of (infor or message) clauses. The ground deduction relation \(\vdash^H\) induced by \(H\) is the smallest set \(\vdash^H \subseteq 2^{\mathcal{T}(\emptyset)} \times \mathcal{T}_{\Sigma(\emptyset)}\) where

1. If \(u \in T\), then \((T, u) \in \vdash^H\), for any \(u\) and \(T\).

2. For any \(t \leftarrow t_1, \ldots, t_n \in H\) and any grounding substitution \(\sigma\) for the terms \(\{t, t_1, \ldots, t_n\}\), if \((T, t\sigma)\) belongs to \(\vdash^H\) for all \(i \in \{1, \ldots, n\}\), then \((T, t\sigma)\) also belongs to \(\vdash^H\).

The relation \(\vdash^H\) always uniquely exists; cf. [57]. The notation \(T \vdash^H a\) stands for \((T, a) \in \vdash^H\). For a finite set of terms \(T\), we write \(T \vdash^HT'\) only if \(T \vdash^H a\) for all \(a \in T'\). Observe that the ground deduction relation \(\vdash^H\) induced by a set of (message and infor) clauses is analogous to the semantics of Horn clauses; indeed, when a predicate is used to wrap around terms in (infor or messages) clauses, they become Horn clauses.

We define how a set of terms, possibly containing variables, is removed from a set of ground terms; this is used to define the semantics of retraction in CAP specifications. Given a set of ground terms \(A\) and a set of terms \(B\), \(\mathcal{R}(A, B)\) denotes the set of all substitutions \(\sigma\) such that \(A' = B\sigma\), for some \(A' \subseteq A\). The retraction of \(B\) from \(A\), for a ground set of terms \(A\) and a set of terms \(B\), denoted \(A\downarrow B\), is defined as \(A\downarrow B = A \setminus \bigcup_{\sigma \in \mathcal{R}(A,B)} B\sigma\). That is, \(B\) is seen as a “pattern”; any subset of \(A\) that can be matched against the pattern is then removed from \(A\).

Example 1. Let \(B = \{f(X), g(X,Y)\}\) be a set of terms, where \(X\) and \(Y\) are variables. Consider a set of ground terms \(A = \{f(a), g(a,b)\}\). Then, \(A\downarrow B = \emptyset\), because for substitution \(\sigma = \{X \mapsto a, Y \mapsto b\}\) we have \(B\sigma = A\). Now, consider the set of ground terms \(C = \{f(a), g(a,b), g(a,c), g(b,d)\}\). The retraction of \(B\) from \(C\) is the set \(C\downarrow B = \{g(b,d)\}\); this is because \(\mathcal{R}(C, B)\) consists of \(\sigma = \{X \mapsto a, Y \mapsto b\}\) and \(\sigma' = \{X \mapsto a, Y \mapsto c\}\), and therefore

\[
C\downarrow B = C \setminus \{(f(a),g(a,b)) \cup \{f(a),g(a,c)\}\} = \{g(b,d)\}.
\]
Given a process $\pi = (\eta, \Omega, I_\pi)$, we use $\vdash^\pi$ as a shorthand for $\vdash^{I_\pi}$, and we refer to the set $\{ a \mid \Omega \vdash^\pi a \}$ as the knowledge of $\pi$. That is, the knowledge of $\pi$ is the set of ground terms that can be “deduced” from the extensional knowledge of $\pi$ using its policy. Similarly, the ground deduction relation of the attacker model $\mathcal{A} = (\Omega_\mathcal{A}, I_\mathcal{A})$ is denoted by $\vdash^\mathcal{A}$ and the knowledge of the attacker is the set $\{ m \mid \Omega_\mathcal{A} \vdash^\mathcal{A} m \}$.

Fix a CAP $\text{cap} = ((\mathbb{S}, \Sigma, \mathcal{V}), \pi^0, \cdots, \pi^l, \mathcal{A}^0)$. A configuration of $\text{cap}$ is an $\ell + 1$ tuple $(\pi_1, \cdots, \pi_\ell, \mathcal{A})$ where the first $\ell$ elements are processes, and the last element is an attacker model. Intuitively, a configuration of $\text{cap}$ represents a “snapshot” of $\text{cap}$ at a given moment in its execution, storing the extensional knowledge of the attacker and the extensional knowledge of the processes, and the remainder of the threads (i.e., the events that are yet to be executed). The initial configuration of $\text{cap}$ is the tuple $(\pi_1^0, \cdots, \pi_\ell^0, \mathcal{A}^0)$. We define trace semantics for CAPs. Let $Z$ be the set of all configurations of $\text{cap}$, i.e., the set of all tuples of the form $z = (\pi_1, \cdots, \pi_\ell, \mathcal{A})$. We define the relation $\rightarrow \subseteq Z \times E \times Z$ as: $(z, e, z') \in \rightarrow$ iff $z = (\pi_1, \cdots, \pi_i, \cdots, \pi_\ell, \mathcal{A})$, $z' = (\pi_1, \cdots, \pi_i', \cdots, \pi_\ell, \mathcal{A}')$, $\mathcal{A} = (\Omega_\mathcal{A}, I_\mathcal{A})$, $\pi_i = (\eta, \Omega, I)$ and $\gamma = e \cdot \gamma' \in \eta$ for some thread $\gamma'$ and $i \in \{1, \cdots, \ell\}$, and one of the following conditions hold:

- $e = g_3 g_3 \rightarrow \text{snd}(t)$, $\mathcal{A}' = (\Omega_{\mathcal{A}} \cup \{ t \sigma \}, I_{\mathcal{A}})$ and $\pi_i' = (\eta \setminus \{ \gamma \} \cup \{ \gamma' \sigma \}, \Omega, I)$, for some substitution $\sigma$ that satisfies the following conditions:

  - $\sigma$ is grounding for $g_3$, and $\Omega \vdash^{\pi_i} g_3 \sigma$.
  - If $g_3 \neq \emptyset$ then there exists no grounding substitution $\sigma'$ for $g_3 \sigma$ such that $\Omega \vdash^{\pi_i} g_3 \sigma'$. 

- $e = \text{recv}(t) \rightarrow u_+ u_-, \mathcal{A}' = \mathcal{A}$ and $\pi_i' = (\eta \setminus \{ \gamma \} \cup \{ \gamma' \sigma \}, \Omega_{u_- \sigma} \cup u_+ \sigma, I)$, for some grounding substitution $\sigma$ for $t$ such that $\Omega_{\mathcal{A}} \vdash^{\mathcal{A}} t \sigma$.

The relation $\rightarrow$ defines the conditions under which a CAP can make a transition from a configuration to another, through an event. If the event is a guarded send event $g_3 g_3 \rightarrow \text{snd}(t)$, the guard $g_3 g_3$ evaluates to true if and only if

$$\exists x_1, \cdots, x_n. g_3 \gamma (\bar{\mathcal{A}} y_1, \cdots, y_m, g_3)$$

where $\{ x_1, \cdots, x_n \} = \text{var}(g_3)$ and $\{ y_1, \cdots, y_m \} = \text{var}(g_3) \setminus \text{var}(g_3)$. If the guard evaluates to true, a message $t \sigma$ is sent to the network, for some model $\sigma$ of the formula above; observe that $\sigma$ is grounding for $t$ due to the origination property, cf. Section 2.1. Note also that variables originating in guards allow one to formulate expressive queries. For instance, consider the event

$$\{ \text{is\_phd}(S) \} \{ \text{reviewing}(S, P) \}^{-} \rightarrow \text{snd}(\{ \text{review\_request}(S, \text{paper}) \}_{\text{pk}(S)})$$

a message $\text{review\_request}(S, \text{paper})$ (denoting a request to review the paper $\text{paper}$) is sent to the network, encrypted with the public key of $S$, if $S$ is a Ph.D. student.
The semantics of the process modeling a storage server that hosts users and allocates space to store their files is defined as the set of all its traces. We say that configuration \( z \) is reachable in cap iff there exists a trace \( z_0e_1z_1\cdots e_nz_n \) in the semantics of cap with \( n \geq 0 \) and \( z = z_n \).

Note that the extensional knowledge of the processes and the attacker in any reachable configuration of cap are indeed sets of ground terms. This is due to the origination property (see Section 2.1) and the conditions the semantics enforces on substitutions. Furthermore, the transition relation defined above does not change the policies of the processes or the attacker models (only threads and extensional knowledge are subject to change).

**Remark 1.** We can encode disjunctive guards into our framework. Note that positive guards of the form \( g_3 = \{t_1,\ldots,t_m\} \) are interpreted conjunctively; that is, \( g_3 \) holds if and only if all \( t_i \), for \( i \in \{1,\ldots,m\} \), can be inferred from the policy of the process in which \( g_3 \) appears. The evaluation of the guard can be therefore seen as evaluating the formula \( \exists \overrightarrow{x} . t_1 \land \cdots \land t_m \), where \( \overrightarrow{x} \) denotes the list of free variables in \( g_3 \).

A disjunctive guard is of the form \( g_1 \lor \cdots \lor g_n \), where \( g_i \) are guards, with \( i \in \{1,\ldots,n\} \). We interpret the disjunctive guard as: \( \exists \overrightarrow{x} . g_1 \lor \cdots \lor g_n \), where \( \overrightarrow{x} \) denotes the list of free variables in \( g_1,\ldots,g_n \). Since existential quantifier distributes over \( \lor \), this is equivalent to the formula \( \exists \overrightarrow{x} . g_1 \lor \cdots \lor \exists \overrightarrow{x} . g_n \).

Based on this observation, the guarded send event \( d_2g_3 \triangleright \operatorname{snd}(m) \) where \( d_2 \) is a disjunctive guard of the form \( g_1 \lor \cdots \lor g_n \) can be encoded into our framework.
by “branching” over $d_\exists$. Let us explain this point with a simple example. Take the sequence of events $\epsilon_1 \cdot (d_\exists g_\exists \triangleright \text{snd}(m)) \cdot \epsilon_2$, where $\epsilon_1$ and $\epsilon_2$ are sequences of events, and $d_\exists = g_1 \lor g_2$ with $g_1$ and $g_2$ being positive guards. We create two interleavings (and the corresponding traces in the semantics) as: $\epsilon_1 \cdot (g_1 g_\exists \triangleright \text{snd}(m)) \cdot \epsilon_2$ and $\epsilon_1 \cdot (g_2 g_\exists \triangleright \text{snd}(m)) \cdot \epsilon_2$, where the guards $g_1$ and $g_2$ are evaluated according to the semantics defined before in this section. It is easy to see that a configuration is reachable through the original sequence of events $\epsilon_1 \cdot (d_\exists g_\exists \triangleright \text{snd}(m)) \cdot \epsilon_2$ if and only if it is reachable through $\epsilon_1 \cdot (g_1 g_\exists \triangleright \text{snd}(m)) \cdot \epsilon_2$, or through $\epsilon_1 \cdot (g_2 g_\exists \triangleright \text{snd}(m)) \cdot \epsilon_2$.

We make use of disjunctive guards in Chapter 6. Observe that in order to comply with the origination property (see Section 2.1) for the guarded send event $d_\exists g_\exists \triangleright \text{snd}(m)$, with $d_\exists = g_1 \lor \cdots \lor g_n$, only the variables that originate in all $g_i$, with $i \in \{1, \cdots, n\}$, are considered to originate in the (disjunctive) positive guard $d_\exists$. Therefore, any variable that appears for the first time in $d_\exists$ but does not originate in $d_\exists$ should not be referred to in $g_\exists$, in $m$, or in the remainder of the thread where this send event appears. For example, consider the following thread with a single guarded send event:

$$d_\exists g_\exists \triangleright \text{snd}(m)$$

where $d_\exists = \{f(X)\} \lor \{g(X, Y)\}$. Now, letting $g_\exists = \{h(Y, Z)\}$ would result in an ill-formed thread, because variable $Y$ is referred to in $g_\exists$, while $Y$ appears in $d_\exists$ but it does not originate in $d_\exists$.

(\text{end of remark})

2.3 The reachability decision problem $\text{REACH}$

Fix a $\text{CAP}$, defined as $\text{cap} = ((S, \Sigma, V), \pi_1, \cdots, \pi_\ell, A)$. A goal $G$ for cap is an $\ell + 1$ tuple $G = (f_1, \cdots, f_\ell, m)$, where $f_i$ are facts and $m$ is a message. Intuitively, goals are conditions on the knowledge of the processes and the attacker: the knowledge of $\pi_i$ must entail $f_i$, and $m$ should be deducible for the attacker. The reachability problem $\text{REACH}(\text{cap}, G)$ asks whether, or not, there exists a reachable configuration $((\eta_1, \Omega_1), \cdots, (\eta_\ell, \Omega_\ell), \Omega_A)$ in $\text{cap}$ such that

$$\forall i \in \{1, \cdots, \ell\}. \Omega_i \vdash f_i \land \Omega_A \vdash m$$

These cases are denoted by $\text{REACH}(\text{cap}, G) = \mathbb{T}$ and $\text{REACH}(\text{cap}, G') = \mathbb{F}$, respectively. A few remarks are due:

1. As a convention, we may write $\text{REACH}(\text{cap}, \pi : f)$ or $\text{REACH}(\text{cap}, A : m)$ when we are interested only in the knowledge of process $\pi$, or the knowledge of the attacker $A$. This can be obviously reduced to the reachability problem defined above, by adding dummy ground terms to the knowledge of the processes/attacker whose knowledge is of no interest to us.
2. The decision problem \textsc{reach} subsumes the \textit{secrecy problem} for security protocols (with a bounded number of sessions). The secrecy problem asks whether the attacker can learn a supposedly secret message \(m\) through interacting with honest processes, e.g. see [50]. This problem can be represented as \textsc{reach}(\textit{cap}, \textit{A} : m).

3. Reachability questions can be extended to guards. Suppose that we want to know whether or not, the \textit{CAP} specification \textit{cap} ever reaches a configuration where \(\Psi = \exists X. f(X) \land \exists Y. g(X, Y)\) is satisfied in the policy of process \(\pi\). This corresponds to checking whether the guard \(\{f(X)\} \{g(X, Y)\}^-\) will ever be satisfied in process \(\pi\). We can simply add to \(\pi\) a “helper” thread which consists of a single guarded send event:

\[
\{f(X)\} \{g(X, Y)\}^- \bowtie \text{snd}(\text{fresh\_term})
\]

where \text{fresh\_term} is a message not appearing anywhere else in \textit{cap}. Now, \(\Psi\) is ever satisfied in the policy of \(\pi\) iff \textsc{reach}(\textit{cap}, \textit{A} : \text{fresh\_term}) = T.

Due to the semi-decidability of message and infon clauses (whose semantics reflects that of general Horn clauses), \textsc{reach} is in general undecidable (even though any \textit{CAP} consists of finitely many processes, and each process runs finitely many threads). We remark that \textsc{reach} is semi-decidable if the ground deduction problem is decidable for the policies of the processes and the attacker; the ground deduction problem for a finite set of (message or infon) clauses \(H\), asks whether \(T \vdash H \ u\), for an arbitrary ground term \(u\) and a finite set of ground term \(T\). This is due to the fact that any \textit{CAP} specification induces a finite number of interleavings of events, even though a specification in general induces an infinite number of traces.

In Chapter 4 and Section 6.1 we identify sets of \textit{CAP}s for which the problem \textsc{reach} is decidable.

2.4 Two sorts: messages and infons

In this section, we motivate our choice of a two-sorted term algebra. In the context of this thesis, we consider two-sorted signatures \((S, \Sigma, \nu)\) where \(S = \{\text{msg}, \text{infon}\}\); \text{msg} denotes the sort of message terms, while \text{infon} denotes the sort of infon terms.

Message terms, denoted \(\text{msg}\_t(S, \Sigma, \nu)\), are the terms that processes send to, and receive from, the communication media. They are used by processes of a \textit{CAP} to communicate with each other. We assume that for each function symbol \(f\) of sort \text{msg} in \(\Sigma\), the arguments of \(f\) are all of sort \text{msg}. In other words, subterms of messages are messages. The set of function symbols \(\Sigma\) of sort message typically would contain the usual cryptographic primitives; i.e., pairing functions, symmetric and asymmetric encryption function, etc. See, for example, Section 4.1.

Infon terms, denoted \(\text{infon}\_t(S, \Sigma, \nu)\), represent policy statements of the processes; we adopt the notion of infons from [38, 39]. Infons are pieces of information; for
example, \texttt{can\_read(piet, file12)} stipulating that Piet can read a certain file named \textit{file12}. An infon does not admit a truth value, i.e. it is not false or true. However, we say that an infon \textit{i holds}, in the context of a process \textit{\pi}, when \textit{i} can be inferred from the policy of \textit{\pi}. Infons can be seen as the “interface” between the communication level and policy level for honest processes. That is, if the policy of Ann entails the fact \texttt{can\_read(piet, file12)}, then the process Ann “knows” that Piet may read \textit{file12}, and she may thus grant him read access to \textit{file12}.

A function symbol \textit{f} in \Sigma, of sort infon and arity \textit{n}, associates a type to each of its arguments; that is, it is a function \textit{s1 × ... × sn → T_{\Sigma(V)}^{\text{infon}}}, with \textit{s_i} ∈ \{msg, infon\} for all \textit{i} ∈ \{1, ..., n\}. Unlike message terms, arguments of functions of type infon can therefore be either message terms or infon terms.

The simplest form of an infon is constructed by applying wrappers to message terms; i.e., an infon constructor function whose arguments are all of type msg. For example, if the message term \textit{n} is a nonce, then the wrapper \texttt{is\_fresh} can be used to construct the piece of information \texttt{is\_fresh(n)}, denoting that the nonce \textit{n} is fresh. Other infons are constructed by applying infon constructor functions to other infons (and messages). For example, \texttt{said(\pi, is\_fresh(n))} is an infon that states that process \textit{\pi} said that \textit{n} is a fresh nonce; then, the infon \texttt{said(\pi', said(\pi, is\_fresh(n)))} states that process \textit{\pi'} said that: process \textit{\pi} said that \textit{n} is fresh.

Note that the sorts msg and infon are indeed disjoint. Ill-typed terms are ignored in our study.
Chapter 3

Examples of CAPs

In this chapter, we illustrate the main features of our formalism through a few examples. The first example concerns an OAuth 2.0 authorization endpoint that is constrained by an RBAC system with transitive attributes. The second example pertains to secure delegation of trust for distributing public key certificates. The third example discusses mechanisms for trust and permission revocation. Finally, we illustrate the feature of variables originating in negative guards or negative updates, and demonstrate their use to express separation of duty constraints.

3.1 OAuth, RBAC, and transitive attributes

Suppose that Ann wants a remote print service, called RPS, to access (on her behalf) the file brochure.pdf stored on a file server. RPS uses OAuth 2.0’s Authorization Code Flow protocol to point Ann to the file server where she can authenticate herself and grant RPS access to the file. OAuth 2.0 is one of the emerging standards in open protocols for secure authorization [9, 33]. After Ann authenticates herself to the file server, the file server needs to first check that Ann is authorized to access the file. If so, the file server sends a URI redirect message to Ann’s Web browser, along with an authorization code. RPS can use the authorization code to access the file, after authenticating itself at a token endpoint. To keep the example simple, we do not model the entire scenario. Instead, we focus on the file server, which is an OAuth 2.0 authorization endpoint.

Suppose that the file server is constrained by an RBAC policy with two roles, user and admin. The server stores two types of files: public and confidential. Any symbolic link to a public file is deemed public, and any link to a confidential file is deemed confidential. By transitivity, links to “links to confidential (respectively public) files” are confidential (respectively public). Users may access any public file or public link; admins may access any confidential file or link. Admins inherit all the rights attributed to users. Here is a formalization of the policy of the file server in the standard Alice & Bob notation; in the remainder of the thesis we will use capital letters to denote variables, and constants will be written with lower-case
letters.

\[
\text{has\_role}(A, \text{user}) \leftarrow \text{has\_role}(A, \text{admin})
\]

\[
\text{can\_access}(A, F) \leftarrow \text{has\_role}(A, \text{user}), \text{has\_attrib}(F, \text{public})
\]

\[
\text{can\_access}(A, F) \leftarrow \text{has\_role}(A, \text{admin}), \text{has\_attrib}(F, \text{confid})
\]

\[
\text{has\_attrib}(\text{link\_to}(F), X) \leftarrow \text{has\_attrib}(F, X)
\]

The rules above are *infon clauses*. Infons denote pieces of information that the file server “knows”; e.g. the file server’s policy engine knows that Ann has the role admin. The infon clauses thereby model the server’s information inference rules. For example, the first rule states that the file server knows that if \(A\) has the role admin, then \(A\) has the role user too, due to the role hierarchy. The second (respectively, the third) rule above states that users (respectively, admins) can read public (respectively, confidential) files. The fourth rule states that links to public (respectively, confidential) objects are themselves public (respectively, confidential). Observe that the last rule is a *recursive* rule, in that the same function constructor (i.e., \(\text{has\_attrib}(\cdot, \cdot)\)) is used in both the conclusion and the premise of the rule.

The *extensional policy*, otherwise called the *extensional knowledge*, of the file server reflects the “current state” of the policy. For example, if the file server knows (only) that Ann is an admin and that *brochure.pdf* is public, then its extensional knowledge will be the set

\[
\{ \text{has\_role}(\text{ann}, \text{admin}), \text{has\_attrib}(\text{brochure.pdf}, \text{public}) \}
\]

Now, let us consider the communication between Ann and the file server. After Ann permits RPS to access *brochure.pdf*, the file server sends back an authorization code to Ann’s *user agent*, e.g. her Web browser. The following guarded send event is part of the file server’s thread:

\[
\{ \text{can\_access}(A, F) \}\{\} \Rightarrow \text{snd}(\text{auth\_code}(F), R_{UR\text{I}}, S_{INFO})
\]

The send event \(\{s_1, \ldots, s_j\}\{n_1, \ldots, n_k\} \Rightarrow \text{snd}(m)\) can be executed only if all the *authorization queries* \(s_1, \cdots, s_j\) follow from the policy, and at least one of the *negative authorization queries* \(n_1, \cdots, n_k\) does not; a formal definition of guarded send events is given in Section 2.1. In the case of the guarded send event above, if the file server’s policy implies \(\text{can\_access}(\text{ann}, \text{brochure.pdf})\), then an authorization code for the file *brochure.pdf* is sent to Ann’s browser. The authorization code is accompanied with a redirection URI \(R_{UR\text{I}}\) and RPS’s state information \(S_{INFO}\). The user agent redirects Ann to RPS, which uses the authorization code to access the requested file after authenticating itself to a token endpoint. Note that, given the policy and extentional knowledge above, the file server can indeed derive the infon \(\text{can\_access}(\text{ann}, \text{brochure.pdf})\).

### 3.2 Trust relations

Ann knows that the root certificate authority RCA is trusted on certifying public keys. Ann asks RCA for Piet’s public key. RCA is however temporarily over-
loaded, and redirects Ann’s request to a local certificate authority CA. RCA trusts CA on public keys. CA sends back to Ann a public key of Piet. Here, we focus on two processes: Ann and CA.

The policy of the process CA contains all public keys belonging to Piet in our scenario. CA also stores a list of revoked public keys. For example, if CA knows that $pk_1$, $pk_2$ and $pk_3$ are public keys of Piet, and also that $pk_2$ has been revoked, CA’s extensional knowledge will contain the policy statements

$$
\text{is} \_ \text{pk} \_ \text{of} (pk_1, \text{piet}) \\
\text{is} \_ \text{pk} \_ \text{of} (pk_2, \text{piet}) \\
\text{is} \_ \text{pk} \_ \text{of} (pk_3, \text{piet}) \\
\text{is} \_ \text{revoked} (pk_2)
$$

CA signs and sends out valid public keys upon request. The following thread, which is a sequence of two events, within the CA process models this behavior:

$$\text{rcv}(A, P) \triangleright \{\} \neg \{\text{is} \_ \text{pk} \_ \text{of} (K, \text{piet})\}\{\text{is} \_ \text{revoked} (K)\} \neg \triangleright \text{snd}(\text{sig} (\text{sk} (\text{ca}), \text{pk} \_ \text{cert} (P, K)), \text{sig} (\text{sk} (\text{rca}), \text{is} \_ \text{pk} \_ \text{certified} (\text{ca})))$$

Here CA receives a pair of names A and P: A is asking CA for a public key of P. This message could have been redirected to CA by, e.g., RCA. Receiving this message binds the variables $A$ and $P$ in the rest of the thread. We remark that the events of the thread are executed sequentially. The message neither adds policy statements to CA’s extensional knowledge, nor retracts any policy statements; hence the sequence $\{\} \neg$ after $\text{rcv}$. We come back to this point shortly. Next, the CA sends out the message $\text{pk} \_ \text{cert} (P, K)$, signed by CA, denoting that CA endorses $K$ as a valid public key of $P$. The endorsement is accompanied by a message signed by RCA which certifies that RCA trusts CA on public keys ($\text{is} \_ \text{pk} \_ \text{certified} (\text{ca})$). Digital signature is denoted $\text{sig} (\cdot, \cdot)$. CA is assumed to have obtained the RCA-signed certificate prior to this exchange.

The CA sends out this message only if:

- CA’s policy entails $\text{is} \_ \text{pk} \_ \text{of} (K, P)$, that is $K$ is a public key of $P$, and
- the policy does not entail $\text{is} \_ \text{revoked} (K)$, i.e. $K$ has not been revoked.

The variable $K$, originating in the guard $\text{is} \_ \text{pk} \_ \text{of} (K, P)$, allows the CA to make a non-deterministic choice. In this example, the CA may choose to send either $pk_1$ or $pk_3$ as a valid public key for Piet. The key $pk_2$, however, cannot be sent, because it has been revoked according to CA’s policy.

We now turn to Ann’s process. The thread that is pertinent to this scenario is given below:

$$\{\} \neg \triangleright \text{snd}(\text{ann}, \text{piet}) \\
\text{rcv}(\text{sig} (\text{sk} (X), \text{pk} \_ \text{cert} (\text{piet}, K)), \text{sig} (\text{sk} (R), \text{is} \_ \text{pk} \_ \text{certified} (X))) \triangleright \\
\{\text{said} (X, \text{is} \_ \text{pk} \_ \text{of} (K, \text{piet}), \text{said} (R, \text{tdon} (X, \text{is} \_ \text{pk} \_ \text{of} (K, \text{piet}))))\} \neg \\
\{\text{is} \_ \text{pk} \_ \text{of} (K', \text{piet})\} \neg \triangleright \text{snd}(\{\text{payload} \}_K))$$
Ann sends out a message (destined to RCA) asking for Piet’s public key. The
thread is tailored to handle delegation: Ann expects to receive a properly formatted
message, signed by an arbitrary process \( X \), given that \( X \) has RCA’s trust on public
keys. Receiving such a message affects the extensional knowledge of Ann. Ann
adds two statements to her knowledge:

- \( \text{said}(X, \text{is}_{-\text{pk} \cdot \text{of}}(K, \text{piet})) \) states that \( X \) endorses \( K \) as a public key of
  Piet. Note that \( \text{is}_{-\text{pk} \cdot \text{of}} \) in this process is local to Ann, i.e. it is independent
  of the internals of CA’s policy, which happens to use the same function symbol for storing public keys and their owners.

- \( \text{said}(R, \text{tdon}(X, \text{is}_{-\text{pk} \cdot \text{of}}(K, \text{piet}))) \) states that \( R \), which in this case it is
  expected to be \( \text{rca} \), certifies that \( X \) is trusted on (denoted \( \text{tdon} \)) public keys;
in particular, any public key associated to Piet.

In general, Ann may interpret a message either by adding policy statements to,
or by retracting policy statements from, her extensional knowledge. That is, the
receive event \( \text{rcv}(m) \) \( \triangleright \{s_1, ..., s_j\}\{n_1, ..., n_k\}^{-} \) means that upon receiving mes-
sage \( m \), the positive policy statements \( s_1, \cdots, s_j \) are all added to Ann’s extensional
knowledge, and the negative policy statements \( n_1, \cdots, n_k \) are all retracted from her
extensional knowledge. Let us assume that CA sends to Ann the message

\[
\text{sig}(\text{sk}(\text{ca}), \text{pk}_{\cdot \text{cert}}(\text{piet}, pk_3)), \text{sig}(\text{sk}(\text{rca}), \text{is}_{\cdot \text{pk} \cdot \text{certified}}(\text{ca}))
\]

Then, Ann adds to her knowledge the following statements:

\[
\begin{align*}
(s_1) & \quad \text{said}(\text{ca}, \text{is}_{-\text{pk} \cdot \text{of}}(pk_3, \text{piet})) \\
(s_2) & \quad \text{said}(\text{rca}, \text{tdon}(\text{ca}, \text{is}_{-\text{pk} \cdot \text{of}}(pk_3, \text{piet})))
\end{align*}
\]

We remark that Ann does not retract any part of her extensional knowledge here.
We come back to the notion of retraction in Section 3.3.

Finally, Ann sends the message payload to Piet, encrypted with \( K’ \), only if
Ann knows that \( K’ \) is Piet’s public key (asymmetric encryption is denoted \( \{\cdot\} \), cf.
Section 4.1).

With respect to Ann’s policy, we have mentioned above that Ann does not
directly trust CA on public keys. She trusts RCA, who delegates its rights (with
respect to endorsing public keys) to CA. We model Ann’s policy below, where
rules are labeled for ease of reference.

\[
\begin{align*}
(\text{TR}) & \quad \text{tdOn}(\text{rca}, \text{is}_{-\text{pk} \cdot \text{of}}(K, U)) \leftarrow \\
(\text{TD}) & \quad \text{tdOn}(P, \text{tdOn}(Q, X)) \leftarrow \text{tdOn}(P, X) \\
(\text{TA}) & \quad X \leftarrow \text{said}(P, X), \text{tdOn}(P, X)
\end{align*}
\]

The first rule \((\text{TR})\), with no preconditions, models Ann’s trust relation: Ann
knows that RCA is trusted on \((\text{tdon})\) certifying any public key. The second rule
\((\text{TD})\) is the essence of transitive trust delegation: If \( P \) is trusted on \( X \), then \( P \)
can delegate this to \( Q \). We remark that delegation of trust need not in general be
transitive; see [38] for more on transitive and non-transitive delegation. The third rule \((TA)\) describes trust application: If Ann knows that \(P\) said \(X\), and she knows that \(P\) is trusted on \(X\), then Ann can infer \(X\); see, e.g., [38] for more details on trust application.

Assuming that the statements \((s_1)\) and \((s_2)\) have been added to Ann’s extensional knowledge, we show that Ann’s policy would entail: \(pk_3\) is Piet’s public key. Thus Ann would send the message \(payload\) to Piet encrypted using \(pk_3\). The following tree, annotated with the names of rules and statements used, shows Ann’s deduction steps.

\[
\begin{align*}
\text{(tdon(\text{rca, is}\_\text{pk\_of}(pk_3, piet)))} & \quad \text{(TR)} \\
\text{(tdon(\text{rca, tdon(\text{ca, is}\_\text{pk\_of}(pk_3, piet))})} & \quad \text{(TD)} \\
\text{(is\_\text{pk\_of}(pk_3, piet))} & \quad \text{(TA)}
\end{align*}
\]

A typical security goal for this scenario is: would Ann ever send a message encrypted with \(pk_2\) (which has been revoked) to Piet?. Another example: for any key \(K\), if Ann infers that \(K\) is Piet’s public key, then Ann trusts RCA on ‘\(K\) is Piet’s public key’. These goals can be expressed as reachability decision problems; cf. Section 2.3. A side note: the scenario described above is susceptible to replay attacks because there is no freshness guarantees for the certified keys received by Ann.

### 3.3 Retraction

Retracting policy statements is a feature often needed for modeling revocation of roles and permissions. Consider a hospital where a sensitive ward can be accessed only by the personnel who work in the ward, and have been vaccinated against a particular virus. The process that controls access to the ward would then contain a thread of the form:

\[
\begin{align*}
\text{rcv(U, sig(sk(ppc), is\_vaccinated(U)))} & \quad \text{\{vaccinated(U)\}\{\neg\}} \\
\text{\{in\_ward(U), vaccinated(U)\}\{\neg\}} & \quad \text{\text{snd(access\_token(U))}}
\end{align*}
\]

That is, an access token is sent to \(U\) only if \(U\) works in the ward (which is denoted \(in\_ward(U)\)), and \(U\) has been vaccinated (denoted \(vaccinated(U)\)). The statement that \(U\) has been vaccinated is added to the extensional knowledge of the process only if the hospital’s personnel protection center (PPC) has put its signature on the message \(is\_vaccinated(U)\).

Now, suppose the vaccination must be repeated every six months. The process would then need to inquire the PPC about the status of \(U\)’s vaccination: whether it is recent or not. Here is a (partial) specification of the thread in the process that contacts the PPC:

\[
\begin{align*}
\text{\{\}\{\neg\}} & \quad \text{\text{snd(u\_status(U))}} \\
\text{\text{rcv(sig(sk(ppc), has\_expired(X)))}} & \quad \text{\{\}\{\text{vaccinated}(X)\}\{\neg\}}
\end{align*}
\]
That is, the process asks the PPC about the vaccination status ($v_{status}$) of $U$. The PPC would send back to the process a signed message declaring that $X$’s vaccination has expired. The PPC would let $X = U$ if $U$’s vaccination has expired; otherwise, the PPC lets $X$ be a dummy name. Then, the process retracts the fact $\text{vaccinated}(X)$ from its policy, obtaining the effect of actually retracting the policy statement $\text{vaccinated}(U)$ if $X = U$, or no effect otherwise.

Two remarks are due. First, note that the messages exchanged between the process and the PPC do not have freshness guarantees. Therefore, the attacker can replay these messages, possibly misinforming the process about the freshness of the vaccination of the personnel. The second, more important, point is that the thread of the process that issues access tokens does not synchronize with the process thread that contacts the PPC. Therefore, race conditions can arise (or, can be caused by a malicious scheduler) where the access token is issued for a personnel whose vaccination is no longer effective. The first problem can be solved using the standard challenge-response message exchange patterns. The second problem can be solved, e.g., using a lock inside the process policy engine. This point is further discussed below.

**Synchronization.** In order to enforce synchronization inside the process, we can use a shared lock between the two threads. This can be modeled as:

**Thread1:**
\begin{align*}
rcv(U, \text{sig(sk(ppc), is\_vaccinated}(U))) \quad &\rightarrow \{\text{vaccinated}(U), \text{lock}(U)\}\updownarrow \\
\{\text{in\_ward}(U), \text{vaccinated}(U)\}, \text{lock}(U) \quad &\rightarrow \quad \text{snd(access\_token(U))}
\end{align*}

**Thread2:**
\begin{align*}
\{\text{lock}(U)\}\updownarrow \rightarrow \quad \text{snd}(v_{status}(U)) \\
rcv(\text{sig(sk(ppc), has\_expired}(X)))) \quad &\rightarrow \quad \{\{\text{vaccinated}(X), \text{lock}(U)\}\updownarrow
\end{align*}

Upon receiving a request from $U$, thread 1 of the process locks the user $U$ internally. The lock can be released only by thread 2 of the process who contacts the PPC to check the status of $U$’s vaccination. The execution of thread 1 can then resume, only if thread 2 has not retracted the statement “$U$ is vaccinated”.

Note that in this particular example, to ensure synchronization, the two threads can be linearized into one single thread. However, since threads execute their events concurrently and occasionally synchronize on critical data, linearizing concurrent threads would in general result in exponentially many interleavings.

**Propagating Revocations.** As mentioned above, retracting policy statements if used without the necessary synchronization mechanisms can be futile. This point is also relevant to how revocations are propagated in distributed settings. For instance, consider the file server of Section 3.1. Remember that Ann has the role admin in the file server. Ann may therefore access any public file. Suppose that Ann’s admin role is revoked after the file server has sent out the authorization code to RPS. After the revocation, Ann does not have the role user in the file server,
because her user role was due to her admin role, which is revoked. The semantics of retraction in our formalism is cascading [18]: if $a$ is entailed due to $b$, and only due to $b$, then retracting $b$ would mean that $a$ will not be entailed by the policy; see the semantics, Section 2.2.

In short, the revocation is cascaded locally on the file server: all that followed from Ann being an admin is retracted after the revocation. However, RPS may use the authorization code issued by the file server to communicate with a token endpoint, hence accessing the file *brochure.pdf* on behalf of Ann. Meanwhile, Ann herself can no longer access the file. That is, in distributed settings, such as the scenario of Section 3.1, revocations do not propagate automatically. To ensure that revocations take effect globally, synchronization mechanisms (e.g. through message passing, using shared objects) are in general necessary. ¹

Careless uses, and unintentional effects, of retraction can be detected by checking reachability. For instance, in this example one could check that after the revocation Ann cannot access *brochure.pdf*, while a state in which RPS can access the file is reachable. Deciding reachability is thus crucial for understanding communicating policies.

(Un)Ambiguous Policies. A form of careless use of retraction is when the policy becomes “ambiguous” about rights. This point is best explained with an example: Suppose that the policy of a library service states that any student can access the reading groups schedule file *schedule.pdf*, formalized as:

$$\text{can\_access}(U, \text{schedule.pdf}) \leftarrow \text{is\_student}(U).$$

Now, imagine upon receiving a complaint about Ann, who is a student, the library service retracts her permission to access *schedule.pdf*:

$$\text{rcv}((\text{complaint\_about}(U)) \triangleright \{} \{ \text{can\_access}(U, \text{schedule.pdf}) \}^{-}\$$

The retraction is meant to prevent Ann from accessing the file, while the fact that she is a student implies that she may access the file. This can be considered as an ambiguity in the policy engine. In the semantics of our formalism (Section 2.2), the infon rule trumps the retraction, because policies are seen as invariants of the process. That is, Ann would be able to access the file, despite “retracting” her access. For enforcing the retraction, either the fact *is\_student(U)* should be retracted (which would be too harsh a reaction to an unsigned complaint), or the guard for accessing the file should be refined to explicitly exclude those about whom a complaint has been received.

¹ This problem is closely related to the problem of attribute staleness in attribute-based authorization logics in distributed settings, analyzed in detail in [46]. In [46], a variety of properties are formulated in linear temporal logic to express different degrees of freshness of the attributes locally stored at a policy decision point. Our work does not focus on these properties, but allows one to prevent staleness altogether by means of synchronization mechanisms (as the example above shows), or to detect consequent authorization violations otherwise.
We remark that rejections in our formalism respect the semantics of Griffiths and Wade [37]: if a policy statement is added to the extensional knowledge of a process, and subsequently the same statement is retracted, the policy remains unchanged. Due to the semantics of infon clauses, however, retracting policy statements that do not belong to the extensional knowledge is inconsequential, as the example above shows. Adding and retracting policy statements, through updates, indeed affect only the extensional knowledge of the processes; see Section 2.2.

3.4 Variables originating in negative guards or updates, and SoD

An interesting feature of our formalism, not illustrated in the previous examples, is the variables that originate in negative guards or negative updates. This feature is described below.

Consider a hospital data center HDC that receives a message from the hospital’s human resources department HR stating that a doctor has retired. The following thread of HDC models this event:

\[
rcv(sig(sk(hr), retired(D))) \Rightarrow \{\} \{\text{is\_patient\_of}(P, D)\}^\sim
\]

That is, after receiving the message, HDC retracts all the statements of the form \(\text{is\_patient\_of}(P, D)\) for all patients \(P\). In other words, after \(D\) has retired, no patient can be considered as his/her patient. That is, the variables that originate in negative updates are interpreted universally (see the semantics, Section 2.2). It is therefore natural that such variables cannot be referred to in the rest of the thread; see the origination property, Section 2.1. We remark that variables cannot originate in positive updates.

Now we turn to the variables that originate in negative guards. Suppose that HDC receives a report \(R\) from a medical personnel \(P\). To review the report, the report may be forwarded to any doctor \(D\) who does not work in the same ward as \(P\). The following thread models this scenario.

\[
rcv(\{\text{sig(sk}(P), report(R))\}_{pk(hdc)}) \Rightarrow \{\} \{\text{in\_ward}(P, W), in\_ward(D, W)\}^\sim
\]

\[
snd(\{\text{please\_review}(R)\}_{pk(D)})
\]

Here, \(\text{in\_ward}(X, W)\) means that \(X\) works in ward \(W\), \(pk(X)\) is a public key of process \(X\). The variable \(W\) originates in a negative guard. Therefore, according to the semantics given in Section 2.2, this guard is satisfied for any \(D\) such that:

\[
\text{is\_doctor}(D) \land \neg \exists W. in\_ward(P, W) \land in\_ward(D, W)
\]

The guard is satisfied for \(D\) only when there is no instantiation for \(W\) where both \(D\) and \(P\) work in ward \(W\). Variables originating in negative guards can therefore not be referred to elsewhere in the thread; see the origination property.
Similarly, separation of duty (SoD) can be expressed in our formalism. Two tasks $T_1$ and $T_2$ are constrained under an SoD relation iff no single principal is allowed to perform them both. In general, SoD relations must be anti-reflexive and symmetric. Consider the following guarded send event:

$$\{\text{can\_do}(A, T_1)\}\{\text{has\_done}(A, T_2), \text{sod}(T_1, T_2)\} \triangleright \text{snd}(\text{auth\_token}(A, T_1))$$

An authorization token to perform task $T_1$ is issued for $A$ only if

- $A$ is allowed to perform $T_1$, denoted $\text{can\_do}(A, T_1)$.
- There exists no task $T_2$ such that $T_2$ has been performed by $A$, denoted $\text{has\_done}(A, T_2)$, and there is a SoD constraint on tasks $T_1$ and $T_2$, denoted $\text{sod}(T_1, T_2)$. That is,

$$\text{can\_do}(A, T_1) \land \neg \exists T_2. \text{has\_done}(A, T_2) \land \text{sod}(T_1, T_2)$$

In case the authorization code to perform $T_1$ is sent to $A$, the fact $\text{has\_done}(A, T_1)$ should be added (via update statements) to the extensional knowledge of the process that enforces SoD, after $A$ has performed the task. This is necessary to keep a complete record of the tasks that have been performed.

It is easy to observe that changes in the SoD relation can be expressed using positive and negative updates in our formalism.
Chapter 4

DC: A class of decidable CAPs

In this chapter, we identify a set of CAPs for which the REACH problem is decidable. We refer to this set as DC. Intuitively, a CAP belongs to DC iff the following conditions hold:

1. The attacker has the capabilities of the standard Dolev-Yao (DY) attacker [31]. The Dolev-Yao attacker intercepts and remembers all transmitted messages; he can encrypt, decrypt and sign messages whose corresponding keys are known to him, can compose and send new messages using the messages it has observed, and can remove or delay messages in favor of others being communicated. We give a formal description of the DY threat model below, in Section 4.1.

2. The policies of honest processes are written in a fragment of infon clauses, hereafter called AL, formally defined in Section 4.2. AL stands for authorization logic.

4.1 The Dolev-Yao attacker model

In the DC fragment we consider the most common threat model in the literature, namely the Dolev-Yao attacker [31]. Below, we list the deduction capabilities of the Dolev-Yao attacker, formalized as message clauses.

In Figure 4.1, the usual cryptographic primitives for security protocols are used: symmetric encryptions \(|\cdot|\), asymmetric encryptions \(\cdot\), digital signatures \(\text{sig}(\cdot, \cdot)\), public key constructors \(\text{pk}(\cdot)\) (and the corresponding secret key constructor \(\text{sk}(\cdot)\)), hash functions \(h(\cdot)\) and pairing \((\cdot, \cdot)\). When confusion is unlikely, we simply write \(x, y\) for the pair \((x, y)\). We assume these cryptographic operators are ideal à la Dolev and Yao [31].

It is a convention, in the literature, to partition the capabilities of the Dolev-Yao attacker into two sets: composition rules, and decomposition rules. Intuitively, the first set constitutes the set of capabilities that allow the attacker to synthesize new messages from the known messages, while the latter set consists of the rules that
allow the attacker to analyze the known messages. More in detail, the composition rules state that he can pair messages ($\phi_{\text{pair}}$), encrypt messages with a known key using both asymmetric ($\phi_{\text{penc}}$) and symmetric ($\phi_{\text{senc}}$) encryption algorithms, hash a message ($\phi_{\text{hash}}$) and sign a message with a known key ($\phi_{\text{sig}}$). Her decomposition capabilities include projecting a component of a pair ($\phi_{\text{proj}}^1$ and $\phi_{\text{proj}}^2$) and extracting the content of a (symmetric or asymmetric) encryption when the appropriate key is known to him.

### 4.2 The AL policy theories

Below, we introduce the AL theories in order to model the policies of honest processes in DC specifications. We recall that a set rewrite system is a finite set of rules of the form $l \Rightarrow r$, with $l$ and $r$ being finite sets of terms. The intuitive meaning of a rule $l \Rightarrow r$ is that a set of terms $S$ can be rewritten to $(S \setminus l\tau) \cup r\tau$, for any grounding substitution $\tau$ with $l\tau \subseteq S$. A set rewrite system is terminating iff for any given set of terms $S$ does not admit any reduction sequence $S \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \cdots$ of infinite length; cf. [7] for more on set rewrite systems. This property is closely related to acyclicity of logic programs in [21]. In the following we use the operator $\sqcup$ to denote disjoint union of sets.

**Definition 1.** Fix a signature $(S, \Sigma, V)$, and let $T$ be a finite set of infon clauses defined over $(S, \Sigma, V)$. $T$ is an AL theory iff $T = T_\leftarrow \sqcup T_\rightarrow$ where

(a) The set rewrite system $\mathcal{R}_{T_\leftarrow} = \{ \{ t \} \Rightarrow \{ t_1, \cdots, t_n \} \mid t \leftarrow t_1, \cdots, t_n \in T_\leftarrow \}$ induced by $T_\leftarrow$ is terminating.

(b) For each clause $t \leftarrow t_1, \cdots, t_\ell$ in $T_\rightarrow$, at least one of the $t_j$, with $j \in \{1, \cdots, \ell\}$, is an anchor for the clause, defined as:

- $\text{var}(t_j)$ contains all the variables appearing in the clause,
- all the variables in $t_j$ are infon variables (i.e. elements of $V_{\text{infon}}$),
- $t_j$ unifies with no term appearing in $T$, except for itself and $t$, and
• the set rewriting system \( \{ \{ t_j \} \Rightarrow \{ t \} \} \) is terminating.

Intuitively, we separate an AL theory \( T \) into two disjoint theories \( T_{\leftarrow} \) and \( T_{\rightarrow} \). Theory \( T_{\leftarrow} \) consists of all the rules in \( T \) on which a terminating backward search could be performed to derive a fact; that is, to derive a fact \( f \) it is possible to rewrite \( f \) as the premises of (an instance of) a rule in \( T_{\leftarrow} \) whose application would yield \( f \). Theory \( T_{\rightarrow} \), on the other hand, includes the rules in \( T \) for which a backward search would possibly not terminate but for which it is possible to identify one of the premises that is an anchor. The notion of anchor hints at terms that can be “safely” searched in the extensional knowledge of a participant (i.e. they are members of the extensional knowledge or can be inferred from it, in a finite number of steps, by means of rules in \( T_{\rightarrow} \)). Since all the variables of a \( T_{\rightarrow} \) rule \( R = h \leftarrow b_1, \cdots, b_n, a \) appear in its anchor \( a \), the search for a proof of \( a \) would bind the variables of the other premises \( b_1, \cdots, b_n \) of \( R \), whose proof can then found by means of a finite backward search. In contrast to Datalog, a fragment of Horn theories which has been extensively used in access control, AL allows for nested function symbols and variables that only appear in the heads of rules.

### 4.3 The DC class of CAPs

We are now ready to define the set DC of CAPs for which we prove REACH is decidable.

**Definition 2.** The set DC consists of all the CAPs \( \left( (S, \Sigma, V), \pi_1, \cdots, \pi_\ell, A \right) \) that satisfy the following conditions:

- \( (S, \Sigma, V) \) is a signature, and \( \Sigma \) contains the functions \( \{ \cdot \}, \{ | \cdot | \}, \text{sig}(\cdot, \cdot), \text{pk}(\cdot), \text{sk}(\cdot), h(\cdot) \) and \( (\cdot, \cdot) \), representing the usual cryptographic primitives. Namely, they represent respectively asymmetric and symmetric encryption, digital signature, public key and private key constructors, hash and pairing functions.
- For any process \( \pi = (\eta^0_\pi, \Omega^0_\pi, I_\pi) \):
  - \( I_\pi \) is an AL theory.
  - Terms in \( \Omega^0_\pi \), in guards and in updates are of type infon. That is, the knowledge of the process ranges over infons.
  - Variables in the guards and updates range over messages, i.e. they are always under the application of a wrapper function symbol.
  - In all the events \( g \triangleright snd(m) \) and \( rcv(m) \triangleright u, m \) is of type msg. This condition ensures that participants send and receive message terms, as opposed to infons.
- \( A = (\Omega^0_A, I^A) \), where \( \Omega^0_A \) is a finite subset of \( \mathcal{T}^\text{msg}_{\Sigma(0)} \), and \( I^A \) consists in the message clauses of Section 4.1, reflecting the capabilities of the standard Dolev-Yao attacker.
The following theorem states that the reachability problem $\text{REACH}$ is decidable for $\text{CAP}$s in $\text{DC}$.

**Theorem 1.** $\text{REACH}(\text{cap}, G)$ is decidable for any $\text{cap}$ in $\text{DC}$, and any goal $G$.

We prove Theorem 1 by giving a decision procedure for the reachability problem for $\text{CAP}$s in $\text{DC}$, and proving its correctness and termination, in Chapter 5.

### 4.4 Examples of DC specifications

It is easy to observe that all examples of Chapter 3, when considering the Dolev-Yao attacker, fall into $\text{DC}$. Indeed, all the policies belong to $\text{AL}$.

In particular, for the examples of Sections 3.1 and 3.3, we note that all the rules belong to $T_{\llcorner}$. For the example of Section 3.2, however, partitioning $T$ to $T_{\llcorner}$ and $T_{\lrcorner}$ is important:

- Let $T_{\llcorner}$ consist of all the rules $t \leftarrow t_1, \cdots, t_\ell$ where $t$ contains all the variables appearing in $t_1, \cdots, t_\ell$. We observe that the reduction system induced by $T_{\llcorner}$ is indeed terminating. In particular, for the trust delegation rule ($TD$)

  $$tdOn(P, tdOn(Q, X)) \leftarrow tdOn(P, X),$$

  we note that each reduction step decreases the number of $tdOn$ infon constructors by one, hence leading to termination.

- Let $T_{\lrcorner}$ contain all the other rules in $T$. That is, $T_{\lrcorner}$ consists of the trust application ($TA$) rule:

  $$X \leftarrow said(P, X), tdOn(P, X)$$

To find an anchor for each rule of $T_{\lrcorner}$ we notice that anchors can only be produced by the clause in which they appear. This is because anchors do not unify with the terms appearing in other rules of $T$. Therefore, anchors must either be present in the initial extensional knowledge of a process, or “initially” introduced when a (signed, and properly formatted) message is interpreted as a policy statement. Naturally, anchors can contain nested terms; hence the quotation mark around “initially”.

Following this intuition, we observe that the term that contains $said$ is the anchor for the ($TA$) rule. Indeed, $said(P, X)$ satisfies all the conditions of being an anchor, except that the variable $P$ appearing here is not an infon variable. Note that $P$ ranges over the names of the processes. Therefore, for each $P$, we can create a new $said_P$ (and the corresponding $tdOn_P$) infon constructor function, and omit $P$ from the rule. There are finitely many processes; therefore the trust application rule can be seen as a compact way of representing finitely many process-dependent trust application rules.
In the following subsections, we discuss two extended examples, respectively of a on-line car registration procedure and of an electronic health record system. We show that the examples indeed belong to DC and give a formalization as CAP specifications.

4.4.1 An extended example of a car registration procedure

We give here an informal description of a service-oriented architecture for an on-line car registration procedure, originated from the European initiative for points of single contact, see [5]. We refer to the examples as CRP. A formal model of CRP is given later in this section.

CRP involves a number of parties: Mike the new owner of a car, Piet the employee of the car registration office, Ann the head of the car registration office, the Human Resources department of the office, called hr, and the Central Repository server, referred to as cr.

Mike buys a car, but before he is allowed to drive it, he must register the car at the car registration office. The office provides an on-line registration service. After producing a document containing all the necessary data for registering the car, Mike sends the document to one of the employees of the office, Piet. If the document is valid, Piet sends it to the Central Repository server to be permanently stored. The cr allows only the employees of the car registration office to write on the server, and Piet is not initially known to the cr as an employee. The cr trusts Ann, the head of the office, on who is an employee of the office. Ann, however, has decided to delegate this task (or, right) to the Human Resources department hr, and communicates this decision to the cr with a certificate. Consequently, the cr inquires the hr on the status of Piet, to which the hr replies with a certificate stating that Piet is an employee of the office. Finally, the cr accepts Piet’s request to store the document. After storing the document, the cr sends back an acknowledgement to Piet. Piet, in turn, sends a token of successful completion of the registration to Mike.

Below, we present the message exchange pattern for the CRP. The usual primitives for security protocols are employed: asymmetric encryptions \{\cdot\}, digital signatures \textit{sig}(\cdot, \cdot), public and secret key constructors \textit{pk}(\cdot) and \textit{sk}(\cdot), hash functions \textit{h}(\cdot) and pairing (\cdot, \cdot). When confusion is unlikely, we write [x]a as a shorthand for \textit{sig}(\textit{sk}(a), x). We assume these cryptographic operators are ideal, à la Dolev and Yao [31]. Below, terms in sans-serif are flags, i.e. unique constants which denote the purpose of their accompanying messages.

\begin{align*}
\text{mike} &\rightarrow \text{piet} : \{\text{mike}, \text{doc}\}_{\text{pk}(\text{piet})}, [\text{h}(\text{doc})]_{\text{mike}} \\
\text{piet} &\rightarrow \text{cr} : \{\text{mike}, \text{doc}\}_{\text{pk}(\text{cr})}, [\text{ann}, \text{h}(\text{mike}, \text{doc})]_{\text{piet}} \\
\text{cr} &\rightarrow \text{piet} : [\text{h}(\text{mike}, \text{doc})]_{\text{cr}} \\
\text{piet} &\rightarrow \text{mike} : [\text{h}(\text{mike}, \text{doc}), \text{success\_token}]_{\text{piet}}
\end{align*}

The delegation of Ann’s right to the hr, and the hr’s attestation that Piet is an
employee, go over a separate exchange.

\[
\begin{align*}
    cr & \rightarrow ann : \text{piet, empl\_status} \\
    ann & \rightarrow cr : [\text{piet, is\_empl, delegated\_to, hr}]_{ann} \\
    cr & \rightarrow hr : \text{piet, empl\_status} \\
    hr & \rightarrow cr : [\text{piet, is\_empl}]_{hr}
\end{align*}
\]

This specification falls short of capturing the internal reasonings of the participants involved, as described informally above. For instance, it is not clear how the participants must interpret the messages they receive, and how the \text{cr} ascertains Piet’s right to store. These relations are often formalized in authorization logics, such as \text{DKAL}. However, specifying the internal reasoning in, e.g. \text{SECPAL} and \text{DKAL}, would fall short of determining the actual messages exchanged, or how the messages are related to policy statements.

**Formal specification**

We give a formal specification of the CRP scenario described above, by defining the \text{CAP} \text{crp} = ((S, \Sigma, V), \text{mike, piet, ann, hr, cr, A}). The signature \Sigma contains:

- The standard cryptographic primitives \{\cdot\}, \{\cdot\}, \text{pk}(), \text{sk}(), \text{h}(), \text{sig}() and (\cdot, \cdot) of type \text{msg}. These primitives denote, respectively, asymmetric and symmetric encryption, public and private key constructors, hashing, digital signatures and pairing.

- A set \Sigma_{\text{infon}} of infon constructors, namely \text{said}(\cdot, \cdot), \text{tdOn}(\cdot, \cdot), \text{can\_store}(\cdot), \text{empl}(\cdot) and \text{head}(\cdot). The functions \text{said}(\cdot, \cdot) and \text{tdOn}(\cdot, \cdot) are of type \text{msg} \times \text{infon} \rightarrow \text{infon}; they denote, for an agent \text{a} and a statement \text{x}, that \text{a} \text{said} \text{x} (\text{said}(\text{a}, \text{x})) and that \text{a} is trusted on \text{x} (\text{tdOn}(\text{a}, \text{x})), respectively. The functions \text{can\_store}(\cdot), \text{empl}(\cdot) and \text{head}(\cdot) are of type \text{msg} \rightarrow \text{infon}; for an agent \text{a}, they denote that \text{a} has the right to store data (\text{can\_store}(\text{a})), that \text{a} is an employee (\text{empl}(\text{a})) and that \text{a} is the head of the office (\text{head}(\text{a})), respectively.

- A set \Sigma_{C} of constants of type \text{msg}, consisting of the identities of the processes in \text{crp} (i.e., \text{mike, piet, ann, cr and hr}), the identity of the attacker \text{eve}, the document \text{doc} and the flags \text{empl\_status, is\_empl, delegated\_to} and \text{success\_token}.

The initial knowledge of the attacker \text{A} = (\Omega^{0}_{\text{A}}, I_{\text{A}}) contains all the constants in \Sigma_{C} except for the document \text{doc}, the public keys of all the honest agents in \text{crp}, and his public and private keys; that is,

\[
\Omega^{0}_{\text{A}} = \Sigma_{C} \setminus \{\text{doc}\} \cup \{\text{pk}(\text{A}) \mid \text{A} \in \{\text{mike, piet, ann, cr, hr, eve}\}\} \cup \{\text{sk}(\text{eve})\}.
\]

The policy \text{I}_{\text{A}} of the attacker reflects the usual DY message derivation rules; a formalization of the DY deduction capabilities is given in Section 4.1. Despite the
The attacker’s knowledge set is always finite, he can generate an infinite set of terms by pairing, hashing, signing and encrypting the terms that are known to her.

The policies $I_{mike}$, $I_{piet}$ and $I_{ann}$ of $mike$, $piet$ and $ann$ (respectively) are empty. That is, Mike, Piet and Ann have no policy. The policy $I_{hr}$ of the $hr$ is the theory $\{empl(X) \leftarrow head(X)\}$; the rule $empl(X) \leftarrow head(X)$ states that heads of the office inherit all rights of employees of the office. Observe that the policy $I_{hr}$ induces a terminating set rewrite system, and thus it falls into $\mathcal{AL}$ (cf. Definition 1).

The policy $I_{cr}$ of the $cr$ contains the rules $TD$ and $TA$:

$$
(\text{TD}) \quad tdOn(P, tdOn(Q, X)) \leftarrow tdOn(P, X)
$$

$$
(\text{TA}) \quad X \leftarrow said(P, X), tdOn(P, X)
$$

The rule $(\text{TD})$ models transitive trust delegation: If $P$ is trusted on $X$, then $P$ can delegate this to $Q$. The rule $(\text{TA})$ describes trust application: If a process $\pi$ knows that $P$ said $X$, and $\pi$ knows that $P$ is trusted on $X$, then $\pi$ can infer $X$; see Section 3.2. Furthermore, $I_{cr}$ includes also the rules

$$
(\text{CS}) \quad \text{can\_store}(X) \leftarrow empl(X)
$$

$$
(\text{TH}) \quad tdOn(H, empl(X)) \leftarrow head(H)
$$

$$
(\text{Hy}) \quad empl(X) \leftarrow head(X)
$$

which constitute a simple hierarchical RBAC system, where employees have the right to store documents in the $cr$, heads of the office are trusted on asserting who is an employee, and heads inherit all rights of employees. Also the policy $I_{cr}$ belongs to $\mathcal{AL}$: we have shown above in this section that the rule $(\text{TA})$ can be included in $T_{\rightarrow}$; the remaining rules induce a terminating set rewrite system, hence they can be included in $T_{\leftarrow}$. See Section 4.2.

Below, we describe the threads executed by the processes in $crp$, and their initial knowledge. Recall that capital letters denote variables. To avoid name clashes, variables appearing in different threads should be tagged with a thread identifier. The tagging is however omitted in the following to ease the presentation. Also, observe that in the CRP scenario no retraction of facts occurs nor negative queries are evaluated; therefore, in order to enhance readability, we omit negative guards and negative queries in events.

**Citizen ($mike$).** The initial knowledge of Mike is empty.

$$
\{\} \triangleright \text{snd}(\{mike, doc\}_{\text{pk}(piet)}, [h(doc)]_{mike})
$$

rcv([$h(mike, doc), success\_token]_{piet}) \triangleright \{\}

Mike sends document $doc$ to Piet, in order to be stored in the $cr$. He then waits to receive a “success” token from Piet. The names $mike$ and $piet$, and also $doc$ are constants in Mike’s specification.

37
Employee (piet). The initial knowledge of Piet is empty.

\[
\text{rcv}(\{C, D\} \text{pk}(\text{piet}), [h(D)]_C) \rightarrow \{\}
\]
\[
\{\} \rightarrow \text{snd}(\{C, D\} \text{pk}(\text{cr}), [h(C, D)]_\text{piet})
\]
\[
\text{rcv}([h(C, D)]_\text{cr}) \rightarrow \{\}
\]
\[
\{\} \rightarrow \text{snd}([h(C, D), \text{success_token}]_\text{piet})
\]

After Piet receives a request, from a citizen \(C\), to store a document \(D\), he sends a corresponding request to the \(\text{cr}\). Then he waits for confirmation (of successful storing) from the \(\text{cr}\), after which he notifies the citizen on the completed transaction.

Head (ann). The initial knowledge of Ann is empty.

\[
\text{rcv}(E, \text{empl_status}) \rightarrow \{\}
\]
\[
\{\} \rightarrow \text{snd}([E, \text{is_empl, delegated_to, hr}]_\text{ann})
\]

When Ann receives a request for information on the status of a principal \(E\), she replies that the task of providing such information has been delegated to the \(\text{hr}\). Ann would typically execute a few instances of this thread in parallel.

Human Resources (hr). The initial knowledge of \(hr\) contains all employees and heads of the office. That is, the \(hr\)’s initial knowledge is \{empl(piet), head(ann)\}.

\[
\text{rcv}(E, \text{empl_status}) \rightarrow \{\}
\]
\[
\{\text{empl}(E)\} \rightarrow \text{snd}([E, \text{is_empl}]_\text{hr})
\]

After receiving a request for the status of a principal \(E\), the \(hr\) confirms that \(E\) is an employee of the office by sending the message \([E, \text{is_empl}]_\text{hr}\), only if \(\text{empl}(E)\) can be inferred from the policy of the \(hr\).

Central Repository (cr). The \(cr\) process consists of two threads, executed in parallel. The initial knowledge of the \(cr\) is \{head(ann)\}.

Central Repository’s main thread.

\[
\text{rcv}([C, D] \text{pk}(\text{cr}), [h(C, D)]_E) \rightarrow \{\}
\]
\[
\{\} \rightarrow \text{snd}(E, \text{empl_status})
\]
\[
\text{rcv}([E, \text{is_empl}]_F) \rightarrow \{\text{said}(F, \text{empl}(E))\}
\]
\[
\{\text{can_store}(E)\} \rightarrow \text{snd}([h(C, D)]_\text{cr})
\]

After receiving the first message from \(E\), requesting to store a document \(D\) as an employee of the office, the \(cr\) sends a message \((E, \text{empl_status})\) inquiring on the employment status of \(E\). If the \(cr\), after receiving the third message, ascertains that \(E\) has the right to store, then the document is stored (not formalized here), and \(E\) is notified.
Central Repository’s delegation handler.

\[ \text{rcv} ([E, \text{is}_\text{empl}, \text{delegated}_\text{to}, HR]_H) \rightarrow \{ \text{said}(H, \text{tdOn}(HR, \text{empl}(E))) \} \]

This thread receives, independently of the other threads, messages from an office head \( H \) to delegate to the \( HR \) the right to declare the employment status of \( E \).

The formalization given above models the inquiries which the \( cr \) conducts via a “broadcast” send. Namely, when the \( cr \) sends the message \((E, \text{empl}_\text{status})\), the message is intended for any head of the office (e.g. \( ann \)), but it is received also by \( F \) (i.e. the \( hr \)). It is thus \( F \) who actually responds to the message \((E, \text{empl}_\text{status})\) in the \( cr \)’s main thread; \( F \)’s response is accepted by \( cr \) because \( ann \) replies to the message with a delegation certificate for \( hr \), which is consumed by the delegation handler thread of the \( cr \).

The derivation tree of Figure 4.2 below shows how the \( cr \) ascertains Piet’s right to write into the repository. The rules used in each derivation step are also shown in the figure. To avoid cluttering, we write \( a \) for \( ann \) and \( p \) for \( piet \).

\[
\begin{array}{c}
\text{head}(a) \\
\text{tdOn}(a, \text{empl}(p)) \quad \text{TH} \\
\text{tdOn}(a, \text{tdOn}(hr, \text{empl}(p))) \quad \text{TD} \\
\text{said}(a, \text{tdOn}(hr, \text{empl}(p))) \quad \text{TA} \\
\text{tdOn}(hr, \text{empl}(p)) \\
\text{empl}(p) \quad \text{CS} \\
\text{can_store}(p) \quad \text{TA} \\
\text{said}(hr, \text{empl}(p)) \quad \text{TA} \\
\end{array}
\]

Figure 4.2: A derivation tree for the CRP example

Remark 2. The formalization can be further extended by adding the following thread to the process of \( Ann \):

\[
\text{rcv}(E, \text{empl}_\text{status}) \rightarrow \{ \} \\
\{ \text{empl}(E) \} \rightarrow \text{snd}([E, \text{is}_\text{empl}])_{\text{ann}}
\]

We also add the fact \( \text{empl}(piet) \) to the initial knowledge of \( Ann \). In this extension \( Ann \) can decide whether to relegate the task to ascertain that Piet is an employee of the office to the Human Resources department or to carry out the task herself (or both). It is immediate that the main thread of \( cr \) remains the same in both these scenarios: it need not be a priori “aware” of whether the delegation takes place or not.

4.4.2 An extended example of an electronic health record system

In this section, we present a formalization of an extended examples on an electronic health record system, originally described in [5]. Medical practices often create,
collect, and manage electronic health records (eHRs, for short) of their patients, for the purposes of compiling the medical history of a patient and ease access to relevant information about the patient to authorized personnel. An eHR is a record of healthcare-related information, which can include information of a wide variety of types, including patient demographics, medical history, medications, emergency contact info, etc. Each record has an implicitly associated access control list, via some indirects which facilitate the understanding for patients and clinicians.

The scenario we present here concerns the electronic health record system of a hospital, and involves a number of parties: the doctors of the hospital Bob and Charlie, the Human Resources department of the hospital, called hr, and the Data Center of the hospital, referred to as dc, that stores the eHRs of the patients of the hospital. We start by giving an informal description of the scenario.

Bob wants to go on vacation. To this end, he requests a leave to the Human Resources department hr of the hospital. The hr, in order to grant Bob the leave, must ascertain that some conditions are met. In particular, the hr must find a substitute for Bob, to attend to the patients of Bob in his absence. Thus, the hr sends to Charlie a request to substitute for Bob. If Charlie is willing to do so, he confirms his availability to the hr by sending a signed hash of the received message.

Upon receiving Charlie’s confirmation, the hr notifies the replacement to the Data Center dc, which should henceforth provide Charlie with the access rights to the eHRs of Bob’s patients. The dc acknowledges receiving the message (and taking appropriate actions to enforce the delegation of rights) by sending back a signed hash of the message. Finally, the hr can grant Bob the leave to go on vacation.

Below, we present the message exchange pattern for the eHR scenario. We assume the usual primitives for security protocols, as in Section 4.4.1. Also here, we write $x, y$ for the pair $(x, y)$, and $[x]_a$ as a shorthand for $a, x, \text{sig}(sk(a), x)$. Recall that we use capital letters to denote variables; words starting with lower-case letters denote constants. In particular, terms in sans-serif are flags, i.e. unique constants which denote the purpose of their accompanying messages.

\[
\begin{align*}
\text{bob} \to \text{hr} : & \; [\text{bob}, \text{hol\_req}]_\text{bob} \\
\text{hr} \to \text{charlie} : & \; [\text{charlie}, \text{repl\_req}, \text{bob}]_\text{hr} \\
\text{charlie} \to \text{hr} : & \; \text{sig}(\text{sk}(\text{charlie}), h(\text{repl\_req}, \text{bob})) \\
\text{hr} \to \text{dc} : & \; [\text{del}, \text{bob}, \text{charlie}]_\text{hr} \\
\text{dc} \to \text{hr} : & \; \text{sig}(\text{sk}(\text{dc}), h(\text{del}, \text{bob}, \text{charlie})) \\
\text{hr} \to \text{bob} : & \; [\text{bob}, \text{hol\_granted}]_\text{hr}
\end{align*}
\]

At the end of this exchange, Charlie should be granted access to the eHRs of Bob’s patients. Below we describe the message exchange for accessing the eHR of a patient of Bob, i.e. Ann.

\[
\begin{align*}
\text{charlie} \to \text{dc} : & \; \text{charlie}, \text{ehr\_req}, \text{ann} \\
\text{dc} \to \text{charlie} : & \; \{\text{ehr}(\text{ann})\}^\text{pk(\text{charlie})}, \text{sig}(\text{sk}(\text{dc}), (\text{ehr}(\text{ann}), \text{ann}))
\end{align*}
\]
This specification falls short of capturing the internal reasonings of the participants involved, as described informally above. For instance, it is not clear why the hr chooses Charlie as substitute of Bob, or whether and how the dc grants Charlie access rights to eHRs of patients of Bob. In the following subsection, we give a formal specification of the eHR example in our formalism.

Formal specification

We give a formal specification of the eHR scenario described above, by defining the $\text{CAP} \ ehr = ((S, \Sigma, V), \ bob, \ charlie, \ hr, \ dc, \ A)$. The signature $\Sigma$ contains:

- The standard cryptographic primitives $\{\cdot\}$, $\{\cdot|\cdot\}$, $\text{pk}(\cdot)$, $\text{sk}(\cdot)$, $h(\cdot)$, $\text{sig}(\cdot, \cdot)$ and $(\cdot, \cdot)$ of type $\text{msg}$; cf. Section 4.4.1. Besides these functions, the signature $\Sigma$ contains the unary function $\text{ehr}(\cdot)$ of type $\text{msg}$, modeling the eHR of patients of the hospital; e.g., the eHR of a patient $p$ is denoted $\text{ehr}(p)$.

- A set $\Sigma_{\text{infon}}$ of infon constructors, consisting of:
  - The functions $\text{said}(\cdot, \cdot)$ and $\text{tdOn}(\cdot, \cdot)$ of type $\text{msg} \times \text{infon} \to \text{infon}$, denoting that an agent $a$ said $x$ ($\text{said}(a, x)$) and that an agent $a$ is trusted on $x$ ($\text{tdOn}(a, x)$), respectively; see Sections 3.2 and 4.4.1.
  - The functions $\text{hrd}(\cdot)$, $\text{wl}(\cdot)$ and $\text{avail}(\cdot)$, of type $\text{msg} \to \text{infon}$; for an agent $a$, they denote that $a$ is the Human Resources department of the hospital ($\text{hrd}(a)$), that $a$ is waiting to receive a vacation leave ($\text{wl}(a)$) and that $a$ is available to substitute for a colleague ($\text{avail}(a)$), respectively.
  - The functions $\text{doc.of}(\cdot, \cdot)$ and $\text{replaces}(\cdot, \cdot)$, of type $\text{msg} \times \text{msg} \to \text{infon}$; they denote, for two agents $a$ and $b$, that $a$ is the treating doctor of $b$ ($\text{doc.of}(a, b)$) and that $a$ is substituting for $b$ in $b$’s absence ($\text{replaces}(a, b)$), respectively.
  - The function $\text{can.read}(\cdot, \cdot)$ of type $\text{msg} \times \text{msg} \to \text{infon}$, which denotes that an agent $a$ has the right to view a document $x$ ($\text{can.read}(a, x)$).

- A set $\Sigma_{\text{C}}$ of constants of type $\text{msg}$ containing the identities of the processes in crp (i.e., $bob, charlie, dc, hr$), the identity of the attacker $eve$, the names of the patients of the hospital and the flags $\text{ehr}_{\text{req}}$, $\text{hol}_{\text{req}}$, $\text{hol}_{\text{granted}}$, $\text{repl}_{\text{req}}$ and $\text{del}$. The latter are used to denote the purpose of their accompanying messages; e.g. $\text{ehr}_{\text{req}}$ identifies a request to read the eHR of a patient.

The initial knowledge of the attacker $A = (\Omega^0_A, I_A)$ contains all the constants in $\Sigma_{\text{C}}$, the public keys of all the honest agents in $ehr$, and his public and private keys; that is,

$$\Omega^0_A = \Sigma_{\text{C}} \cup \{\text{pk}(A) \mid A \in \{bob, charlie, dc, hr, eve\} \cup \{\text{sk}(eve)\}.$$

41
The policy $I_A$ of the attacker reflects the usual DY message derivation rules, as formalized in Section 4.1.

Below, we describe the processes of ehr. Recall that capital letters denote variables, and that, in order to avoid name clashes, we assume variables appearing in different threads to be tagged with a unique thread identifier (the tagging is omitted here).

**Doctor (bob&charlie).** The initial knowledge and the policy of doctors are empty. The process of a doctor contains three threads executed in parallel. The first thread requests access to the eHR of a patient. The second thread requests a vacation leave the Human Resources department. The third thread responds to requests to substitute for a colleague. Below, we formalize and explain the threads for the doctor Bob; the formalization for Charlie is analogous.

**Doctor’s eHR request thread.**

\[
\begin{align*}
\text{rcv}(A) & \quad \{\{\}^\sim \} \\
\{\{}^\sim \} & \quad \text{snd}(\text{ehr req}, bob, A) \\
\text{rcv}([E]_{\text{pk(bob)}}, \text{sig}(\text{sk}(dc), (E, A))) & \quad \{\{\}^\sim \}
\end{align*}
\]

We let the attacker decide which eHR Bob requests to view, by receiving the identity of the patient $A$ in the first event. Bob then sends to the Data Center $dc$ a tuple $(\text{ehr req}, bob, A)$ consisting of the flag ehr req, his identity and the identity of the patient $A$; the tuple denotes a request to access the eHR of $A$. Finally, Bob waits to receive the eHR $E$ of $A$, in a properly formatted message in order to ensure secrecy and authenticity of $E$.

**Doctor’s vacation leave request thread.**

\[
\begin{align*}
\{\}^\sim & \quad \text{snd}([bob, \text{hol req}]_{bob}) \\
\text{rcv}([bob, \text{hol granted}]_{hr}) & \quad \{\{\}^\sim \}
\end{align*}
\]

Bob sends a message $[bob, \text{hol req}]_{bob}$ to the Human Resources department $hr$, denoting a request for a vacation leave. He then waits to receive the vacation leave (represented by the token hol granted) from the $hr$.

**Doctor’s replacement response thread.**

\[
\begin{align*}
\text{rcv}([bob, \text{repl req}, D]_{hr}) & \quad \{\{\}^\sim \} \\
\{\}^\sim & \quad \text{snd}(\text{sig}(\text{sk}(bob), h(\text{repl req}, D)))
\end{align*}
\]

Bob receives a message $[bob, \text{repl req}, D]_{hr}$ from the $hr$, representing a request to substitute for doctor $D$. He then sends back, as a confirmation message, a signed hash of the flag repl req and the identity of the doctor $D$.

Observe that the names $bob$, $hr$ and $dc$ are constants in Bob’s specification.
**Human Resources** (hr). The policy $I_{hr}$ of the hr is empty. The initial knowledge of the hr is the set $\{\text{avail}(X) \mid X \text{ is a doctor}\}$; that is, initially all doctors are available. In particular, the hr knows that $\text{avail}(\text{bob})$ and $\text{avail}(\text{charlie})$.

The hr process contains two threads that execute in parallel. The first thread carries out the exchange necessary to grant a vacation leave to a doctor who requests it. The second thread resolves deadlocks arising when all doctors either are on holidays or are waiting to receive a vacation leave.

**Human Resources’ vacation leave request thread.**

\[
\begin{align*}
\text{rcv}(\begin{array}{l}D, \text{hol}_{\text{req}}\end{array}) & \quad \Rightarrow \quad \{\text{wl}(D)\}\{\text{avail}(D)\}^-
\end{align*}
\]

\[
\begin{align*}
\{\text{wl}(D), \text{avail}(R)\} & \quad \Rightarrow \quad \text{snd}(\begin{array}{l}R, \text{repl}_{\text{req}}, D\end{array})_{hr}
\end{align*}
\]

\[
\begin{align*}
\text{rcv}(\text{sig}(\text{sk}(R), h(\text{repl}_{\text{req}}, D))) & \quad \Rightarrow \quad \{\}{\}\^-
\end{align*}
\]

\[
\begin{align*}
\{\}{\}\^- & \quad \Rightarrow \quad \text{snd}(\begin{array}{l}\text{del}, D, R\end{array})_{hr}
\end{align*}
\]

\[
\begin{align*}
\text{rcv}(\text{sig}(\text{sk}(Dc), h(\text{del}, D, R))) & \quad \Rightarrow \quad \{\}{\}\{\text{wl}(D)\}^-
\end{align*}
\]

The hr receives a request for a vacation leave from a doctor $D$. Subsequently, the process updates its knowledge in two ways: (1) it retracts the fact $\text{avail}(D)$, thus indicating that $D$ is not available to replace other doctors, and (2) it adds the fact $\text{wl}(D)$ denoting that $D$ is waiting to receive a vacation leave. Then, the hr sends a request to substitute for doctor $D$ to some available doctor $R$. Including in the guard the positive query $\text{wl}(D)$ allows the hr to avoid choosing $D$ as a replacement for himself, when his request has been discarded by the other thread of the process (see below for details). If such a replacement $R$ is found, a request to stand in for $D$ is sent to $R$, and a confirmation message from $R$ is awaited.

When $R$ confirms availability to substitute for $D$, the hr process sends a message $[\text{del}, D, R]_{hr}$ to the Data Center $Dc$; the message informs the $Dc$ of the replacement. The $Dc$ should then take actions to enforce the delegation of access rights of $D$ to $R$. When the hr process receives a confirmation message from $Dc$, ensuring that the necessary actions have been taken, the hr retracts the fact $\text{wl}(D)$ from its knowledge. Observe that the fact $\text{avail}(D)$ is not reintroduced in the knowledge of the hr process: this fact will be restored when doctor $D$ will come back from vacation (not modeled here). Finally, the hr sends the vacation leave to doctor $D$, represented by the token $\text{hol}_{\text{granted}}$.

**Human Resources’ deadlock detection thread.**

\[
\begin{align*}
\{\text{wl}(D)\}\{\text{avail}(R)\}^\rightarrow & \quad \{\text{avail}(D)\}\{\text{wl}(D)\}^-
\end{align*}
\]

This thread detects when a deadlock arises due to all the doctors of the hospital either being out for vacation or waiting for a leave to go on vacation. The system ensures that there is always at least a doctor in the hospital. It can not ensure, however, that the doctors that are in the hospital do not request to go on vacation simultaneously. Such cases are detected by the guard of the event; in particular, a doctor $D$ waiting for a vacation leave is chosen non-deterministically.
Then, the hr process discards the vacation leave request of $D$, and reintroduces the fact $avail(D)$ in the knowledge of the hr process. In between the evaluation of the guard and performing the update, a dummy message is sent to the network. Now, if any other doctor $D'$ is waiting for a vacation leave, doctor $D$ can substitute for $D'$. Observe that $D$ can not fill in for his own vacation leave request, due to the query $wl(D)$ in the event of the other thread of the hr process where a replacement is chosen.

**Data Center** ($dc$). The initial knowledge of the Data Center is the set

$$\{hrd(hr)\} \cup \{doc\_of(D,P) \mid D \text{ is the doctor of } P\};$$

that is, the $dc$ knows that $hr$ is the Human Resources department, and for all patients $P$ the $dc$ knows who is the treating doctor of $P$. The policy $I_{dc}$ of the Data Center contains the following rules:

$$(TA)\quad X \leftarrow said(P,X), tdOn(P,X)$$

$$(TH)\quad tdOn(X, replaces(R,D)) \leftarrow hrd(X)$$

$$(AD)\quad can\_read(D, ehr(P)) \leftarrow doc\_of(D,P)$$

$$(AR)\quad can\_read(R, ehr(P)) \leftarrow doc\_of(D,P), replaces(R,D)$$

$$(TR)\quad replaces(A,C) \leftarrow replaces(A,B), replaces(B,C)$$

The first rule is the trust application rule, which states that if $dc$ knows that $P$ said $X$, and $dc$ knows that $P$ is trusted on $X$, then $dc$ can infer $X$; see Section 3.2. The second rule states that the Human Resources department is trusted on the statement “$R$ replaces $D$”, for any $R$ and $D$. The third and fourth rule state, respectively, that the eHR of a patient $P$ can be viewed by a doctor $D$ that is the treating doctor of $P$, or by a doctor $R$ that replaces the treating doctor $D$ of $P$. Finally, the last rule states that if a doctor $A$ replaces a doctor $B$, and $B$ replaces a doctor $C$, then $A$ replaces $C$; the rule expresses the transitivity of the relation represented by the function $replaces$.

The policy $I_{dc}$ is not in $AL$: while the rule $(TA)$ is included in $T\rightarrow$, the remaining rules do not induce a terminating set rewrite system; cf. Definition 1. This is due to rule $(TR)$. We note, however, that under the reasonable assumption that a finite number of doctors work in the hospital, and their identities are known a priori, there are finitely many facts of the form $replaces(A,C)$ that can be inferred from a finite set of facts. We can use this observation to replace the rule $(TR)$ with all its ground instances $replaces(a,c) \leftarrow replaces(a,b), replaces(b,c)$, for any distinct doctors $a$, $b$, and $c$. Such a formulation induces a finite set rewriting system, hence it can be included in $T\rightarrow$.

The process of the Data Center contains two threads, executed in parallel; the first thread handles requests to view the eHR of a patient, while the second thread responds to notifications of delegation sent by the Human Resources department.
Data Center’s eHR request thread.

\[ \text{rcv(ehr}_{\text{req}}, D, A) \] \[ \Rightarrow \{ \} \] \[ \neg \{ \} \]
\[ \Rightarrow \{ \text{can}_{\text{read}}(D, ehr(A)) \} \neg \]
\[ \Rightarrow \text{snd}\{ehr(A)\}_{pk(D)}, \text{sig}(sk(dc), (ehr(A), A)) \]

The Data Center receives a message \((\text{ehr}_{\text{req}}, D, A)\) denoting that doctor \(D\) wishes to view the eHR of patient \(A\). If \(D\) has access rights to the eHR of \(A\), then the \(dc\) sends the eHR \(ehr(A)\) to \(D\).

Data Center’s delegation thread.

\[ \text{rcv([del, D, R]}_{H} \] \[ \Rightarrow \{ \text{said}(H, \text{replaces}(R, D)) \} \neg \]
\[ \Rightarrow \{ \} \neg \Rightarrow \text{snd}(\text{sig}(sk(dc), h([del, D, R])) \]

The \(dc\) process receives a message \([\text{del, D, R]}_{H}\), denoting a request to enforce a delegation of rights from doctor \(D\) to doctor \(R\). Subsequently, the \(dc\) adds the fact \(\text{said}(H, \text{replaces}(R, D))\) to its knowledge. Observe that this fact has an effect on the policy of the \(dc\) only if the \(dc\) trusts \(H\) on such statements. The process sends then, as a confirmation message, a signed hash of the previously received message.

The derivation tree of Figure 4.3 below shows how the Data Center \(dc\) ascertains Charlie’s right to view the eHR of a patient of Bob, Ann. The right is granted to Charlie after an execution of the CAP where Bob’s request for a vacation leave is satisfied, and Charlie is chosen as Bob’s replacement. The rules used in each derivation step are also shown in the figure.

Figure 4.3: A derivation tree for the eHR example
Chapter 5

A decision procedure for REACH for CAPs in DC

In this chapter we give a detailed description of our procedure for deciding the REACH problem for CAPs in DC. The procedure, called reach, is shown in Algorithm 1 below.

Algorithm 1 Procedure reach
REQUIRES: REACH(cap, G), cap = ((S, Σ, V), π₁, ··· , πℓ, A)

I := interleavings({π₁, ··· , πℓ})
for all ι ∈ I do
    repeat
        (res, σ, Ψ) := solve(cs(ι, G))
    until res = T or all reduction paths have been explored
    if res = T then
        return (T, (ι, σ, Ψ))
    return (F, (ε, ϵ, Φ, Φ))

We start by giving an informal overview of the procedure reach. Algorithm 1 takes as input a reachability problem instance REACH(cap, G), where cap is a CAP specification in DC (see Chapter 4), and G is a goal (see Section 2.3). The return value is a pair (V, W), where V is a truth value denoting whether or not there exists a reachable configuration in cap that satisfies G, and W is a witness of REACH(cap, G) = T. More precisely, W is a triple (ι, σ, Ψ), where ι is an interleaving of cap, σ is an instantiation of variables, and Ψ is a satisfiable provision formula (i.e., a formula that captures necessary restrictions on the instantiation of variables; provision formulas are formally introduced in Section 5.5); when V = T, then, for any substitution τ that satisfies Ψ, τστ is a trace of cap that reaches a configuration satisfying the goal G.
The algorithm hinges upon the notion of constraints (formally defined in Section 5.1). Constraints model “symbolic” deduction problems: that is, given a “knowledge” (a set of terms), a “query” (a literal, i.e. a possibly negated term) and an inference theory, whether there exists an instantiation of the variables under which the query can be inferred from the knowledge. We refer to such an instantiation of the variables as a solution of the constraint; there may be zero or more solutions to a constraint. A solution of a set of constraints is an instantiation of the variables that is a solution for all the constraints in the set simultaneously.

We differentiate and group constraints based on the criteria explained below. Firstly, we differentiate positive constraints (i.e. constraints whose query is a non-negated term) from negative constraints (i.e. constraints whose query is a negated term). Secondly, we differentiate constraints based on the inference theory they refer to. In the context of CAP specifications in the DC fragment, constraints whose query is of type msg are always solved under the Dolev-Yao inference theory (cf. Section 4.1), whereas all other constraints refer to theories in AL (see Chapter 4); we refer to the former as attacker constraints, and to the latter as policy constraints. We remark that all attacker constraints are positive, while policy constraints can be both positive and negative. Given a set of constraints $CS$ we will denote the (disjoint) subsets of attacker constraints, positive policy constraints and negative policy constraints in $CS$ as $att(CS)$, $pos(CS)$ and $neg(CS)$ respectively.

The algorithm reach generates a constraint sequence $cs(\iota, G)$ for each interleaving $\iota$ of the processes in cap, taking also into account the reachability goal $G$ under analysis; a precise definition of $cs(\iota, G)$ is given in Section 5.2. For each event in $\iota$ (respectively, reachability goal) a corresponding constraint is added to $cs(\iota, G)$ that models the event (respectively, the goal) in terms of a symbolic deduction problem. Constraints that model the reception of messages or the evaluation of positive queries are referred to as positive; constraints that model the evaluation of negative queries are referred to as negative. In the construction of the constraint set $cs(\iota, G)$ also a first step to reflect retraction in the CAP specification is taken: each constraint is coupled with a retraction record function $\Upsilon$. The function $\Upsilon$ associates to each term $t$ in the constraint a provision formula that captures the conditions on the instantiation of variables under which $t$ is not retracted. The symbolic treatment of retraction of facts is discussed in detail in Section 5.7. Each CAP includes a finite number of threads, each being a finite set of events, hence there are finitely many interleavings and finitely many constraint sets generated for any CAP.

The constraint set $cs(\iota, G)$ is then input to a constraint reduction system, called solve, shown in Algorithm 2 below. The reduction system solve tries to reduce the constraint set $cs(\iota, G)$ to a trivial form, i.e. immediately satisfiable. More precisely, a constraint set is trivial if for all constraints $c \in att(CS)$ the query of $c$ is a stand-alone variable, and $pos(CS) = neg(CS) = \emptyset$. The procedure solve reduces the constraint set until all of these conditions are met, or no further reduction is possible: at each iteration of the loop, one of the procedures reduceDY, solvePos and solveNeg is called to reduce $att(CS)$, $pos(CS)$ or $neg(CS)$, respectively. Each of
Algorithm 2 Procedure solve

REQUIRES: constraint set $CS$
\[
\tau = \emptyset, \Psi = \emptyset
\]
while $CS$ is not trivial && $CS$ is not maximally reduced do
  non-deterministically execute one of the following
  \begin{itemize}
    \item if $\text{att}(CS)$ is not trivial then $(CS', \sigma, \Delta) = \text{reduceDY}(CS)$;
    \item if $\text{pos}(CS) \neq \emptyset$ then $(CS', \sigma, \Delta) = \text{solvePos}(CS)$;
    \item if $\text{neg}(CS) \neq \emptyset$ then $(CS', \sigma, \Delta) = \text{solveNeg}(CS)$;
  \end{itemize}
\[
CS := CS', \sigma, \tau := \tau \cup \sigma, \Psi := (\Psi \land \Delta)\sigma
\]
if $\text{checkProv}(\Psi) = F$ then
  return $(F,\emptyset,\emptyset)$
if $CS$ is trivial then
  return $(T,\tau,\Delta)$
else
  return $(F,\emptyset,\emptyset)$

these procedures return a triple $(CS', \sigma, \Delta)$, where $CS'$ is the reduced constraint set, $\sigma$ is a solution for the subset of $CS$ reduced, and $\Delta$ is a provision formula. The formula $\Delta$ characterizes the instantiations of variables that (1) solve the negative constraints, if the reduction procedure invoked is $\text{solveNeg}$, or (2) ensure the soundness of the reduction performed by $\text{solvePos}$, when retraction of facts occurs; moreover, $\Delta$ is the empty formula (i.e., no restriction on the instantiation of variables is imposed) when the reduction procedure invoked is $\text{reduceDY}$.

If $\text{att}(CS)$ is not trivial, then procedure $\text{solve}$ may call the constraint reduction system $\text{reduceDY}$, to reduce the attacker constraints in $CS$. The reduction system $\text{reduceDY}$ is explained in detail in Section 5.3.

If $\text{pos}(CS) \neq \emptyset$, then the proof search algorithm $\text{solvePos}$ for $\mathbf{AL}$ theories is used to solve the positive policy constraints in $CS$. For an $\mathbf{AL}$ theory $T = T_\to \sqcup T_\leftarrow$, the algorithm $\text{solvePos}$ consists of a cut elimination step that accounts for the rules in $T_\to$ and a backward search algorithm that accounts for the rules in $T_\leftarrow$. Intuitively, $\text{solvePos}$ tries to guess a proof for the symbolic deduction problem represented by the constraint in input. Furthermore, $\text{solvePos}$ takes a second step in treating retraction of facts: when a proof $\Pi$ of the deduction problem is found, $\text{solvePos}$ returns a provision formula $\Delta$ that ensures the soundness of $\Pi$; that is, no fact used by $\Pi$ has been retracted. The provision formula $\Delta$ is constructed using the retraction record function $\Upsilon$ associated to the constraint in the generation of the constraint set. We describe procedure $\text{solvePos}$ in detail in Section 5.4, and elaborate on the treatment of retracted facts in Section 5.7.

If $\text{neg}(CS) \neq \emptyset$, the procedure $\text{solveNeg}$ is used to solve the negative constraints in $CS$. Procedure $\text{solveNeg}$ “complements” all the constraints in $\text{neg}(CS)$, i.e. turns each negative constraint into a positive constraint expressing the opposite (positive) symbolic deduction problem; this is done by removing the negation in the query term. The key idea is that the negative constraint is solvable when its com-
plement has no solutions; cf. [24]. To this end, `solveNeg` calls procedure `solvePos` to solve the complement of the negative constraint, and collects the set of all the solutions returned. The set of solutions is then used to construct a provision formula $\Delta$, whose models are solutions of the negative constraints in $\text{neg}(CS)$. A detailed description of `solveNeg` is given in Section 5.5.

After the reduction procedure invoked has returned the solution $(CS', \sigma, \Delta)$, the provision formula $\Delta$ is added to the provision formula $\Psi$ to be returned in case of success, and the substitution $\sigma$ is propagated to the reduced constraint set and to $\Psi$, and stored. The provision formula $\Psi$ must remain satisfiable at all times; in particular, after each reduction step, the procedure `checkProv` is invoked to ascertain the satisfiability of $\Psi$. If `checkProv` returns the truth value $T$, then $\Psi$ is satisfiable and the computation can continue; however, if `checkProv` returns $F$, then $\Psi$ is unsatisfiable and `solve` returns the truth value $F$ (along with a dummy substitution and provision formula).

Finally, if $CS$ is reduced to a trivial constraint set, then `solve` returns the truth value $T$, a substitution $\tau$ and a provision formula $\Psi$. The returned value denotes that the constraint set $CS$ in input is solved by any substitution $\tau\sigma$ with $\sigma$ being a substitution that satisfies $\Psi$. However, if $CS$ is reduced to a non-trivial constraint set and no further reduction is possible, then `solve` returns the truth value $F$ (and a dummy solution and provision formula). Observe that `solve` considers only one of the (finitely many) possible reduction paths; hence the truth value $F$ denotes that this reduction path did not solve the constraint set $CS$. Procedure `reach` repeatedly invokes `solve` to exhaustively search all possible reduction paths. We remark that there are finitely many reduction paths.

In the following sections we describe in detail the subroutines of procedures `reach` and `solve`. In Section 5.1 we introduce formally the notions needed for subsequent sections. In the sections that follow, we explain

- the generation of the constraint set, Section 5.2;
- the reduction procedure `reduceDY` for attacker constraints, Section 5.3;
- the reduction procedure `solvePos` for positive policy constraints, Section 5.4;
- the reduction procedure `solveNeg` for negative policy constraints, Section 5.5;
- the procedure `checkProv` to verify the satisfiability of the provision formula $\Psi$ constructed in the above reductions, Section 5.6.

In these sections, in order to facilitate understanding, we restrict our attention to CAP specifications where no negative update occurs; that is, for all receive events of the form $\text{rcv}(t) \triangleright u_+ u_-$ appearing in the specification $u_- = \emptyset$. The symbolic treatment of retraction of facts and its integration in the procedure `reach` is then explained in Section 5.7. Finally, in Section 5.8 we prove the correctness (i.e. termination, soundness and completeness) of Algorithm 1.
5.1 Preliminaries

In this section we formally define the notions of constraints, constraints solutions and several others that will be used in subsequent sections.

Given a CAP specification cap, we consider the following typing discipline: a subset $V^\exists \subseteq V_{msg}$ of positively originated variables is given; variables in $V^\exists$ appear for the first time in cap either in a message received from the network or in a positive guard. All other variables (that is, variables in $V \setminus V^\exists$) are called free; free variables originate in negative guards and in negative updates. The set of positively originated variables is partitioned in two sets $V_a$ and $V_g$ of, respectively, attacker originated variables (i.e. appearing for the first time in cap in a receive event) and guard originated variables (i.e. appearing for the first time in cap in a guard event). Obviously, the sets $V_a$ and $V_g$ are disjoint.

Consider a finite set of (message or infon) clauses $T$. A positive constraint over $T$ is of the form $S \vdash_T u$ where $S \subseteq \mathcal{T}_{\Sigma(V)}$, $S$ is finite, and $u \in \mathcal{T}_{\Sigma(V)}$. Intuitively, such constraint represents a “symbolic” deduction problem, namely the problem of finding an instantiation of the variables in $S \cup \{u\}$ under which $u$ can be inferred from $S$; we call such an instantiation a solution of the constraint. Formally, a solution of $S \vdash_T u$ is a grounding substitution $\sigma$ for $S \cup \{u\}$ such that $S\sigma \vdash\Sigma T u\sigma$.

A negative constraint over $T$ is of the form $S \not\vdash_T U$ where $S \subseteq \mathcal{T}_{\Sigma(V)}$, $U \subseteq \mathcal{T}_{\Sigma(V)}$, and $S$ and $U$ are finite. When $U$ is a singleton set $U = \{u\}$ we write $S \not\vdash_T u$ in place of $S \not\vdash_T \{u\}$. In contrast to positive constraints, a negative constraint represent the problem of finding an instantiation of the positively originated variables in $S \cup U$ under which $u$ can not be inferred from $S$, for some $u \in U$. A solution of $S \not\vdash_T U$ is a partial substitution $\sigma : V^\exists \rightarrow \Sigma_{\emptyset}$ (with $\text{dom}(\sigma) \supseteq \text{var}(S \cup U) \cap V^\exists$) such that for all ground substitutions $\rho : V \setminus V^\exists \rightarrow \mathcal{T}_{\Sigma(\emptyset)}$ there exists $u \in U$ such that $S\sigma\rho \not\vdash_T u\sigma\rho$.

Positive constraints are generally used to model the reception of messages from the network or the evaluation of positive guards against the policy of a process; hence, for a positive constraint $S \vdash_T u$, $\text{var}(S \cup \{u\}) \subseteq V^\exists$, i.e. the constraint contains only positively originated variables. A notable exception is the use of positive constraints as “complements” of negative constraints in solving negative constraints; cf. Section 5.5. Negative constraints, on the other hand, are used to model the evaluation of negative guards against the policy of a process; therefore a negative constraint $S \not\vdash_T U$ might contain also free variables, which are interpreted as universally quantified.

Example 2. Let $\pi$ be a process with policy $I_\pi$, and the following be a guarded send event executed by a thread of $\pi$ at configuration $\varepsilon$:

$\{f(X)\}\{g(X,Y), h(Y)\} \triangleright \text{snd}(m),$

with $X \in V^\exists$ being a positively originated variable and $Y$ being a free variable. The term $m$ is sent to the network if the guard evaluates to true; that is, if there exists a value of $X$ such that $f(X)$ can be inferred from $\Omega$ and there exists no
value of $Y$ such that $g(X,Y)$ and $h(Y)$ can be inferred from $\Omega$, with $\Omega$ being the extensional knowledge of $\pi$ at configuration $z$. The evaluation of the guard can be modeled with the following two constraints:

$$\Omega \models^\pi f(X)$$
$$\Omega \not\models^\pi \{g(X,Y), h(Y)\}$$

The constraint are (simultaneously) solved by a substitution $\tau$: $
\exists\rightarrow T$ $\Sigma(\emptyset)$ such that $\Omega \tau \models^\pi f(X)\tau$ and for all substitutions $\rho: \emptyset \rightarrow T$ $\Sigma(\emptyset)$ either $\Omega \tau \rho \not\models^\pi g(X,Y)\tau \rho$ or $\Omega \tau \rho \not\models^\pi h(Y)\tau \rho$.

(end of example)

A constraint system $C$ is a finite set of constraints. A solution of a constraint system $C$ is a substitution $\tau$: $\exists\rightarrow T$ $\Sigma(\emptyset)$ grounding the positively originated variables in $C$ that solves all constraints in $C$ simultaneously.

We define the refinement partial order $\sqsubseteq$ over the set of all the substitutions $\emptyset \rightarrow T$ $\Sigma(\emptyset)$ as: $\sigma'$ refines $\sigma$, denoted $\sigma \sqsubseteq \sigma'$, if there exists a mapping $\sigma''$ such that $\sigma' = \sigma \sigma''$. Two substitution functions $\sigma$ and $\sigma'$ are equivalent, denoted $\sigma \equiv \sigma'$, iff $\sigma \sqsubseteq \sigma'$ and $\sigma' \sqsubseteq \sigma$; i.e. they are the same substitution modulo $\alpha$-renaming of the variables.

We extend the notion of solution of a constraint to include non-ground substitution functions whose ground refinements are solutions of the constraint. Formally, a symbolic solution of a constraint $c$ is a partial substitution function $\sigma: \emptyset \rightarrow T$ $\Sigma(\emptyset)$ such that $\sigma \sigma'$ is a solution of $c$ for any ground total substitution $\sigma': \emptyset \rightarrow T$ $\Sigma(\emptyset)$. We use symbolic solutions in order to instantiate only the variables that are necessary to solve a constraint, without binding the remaining ones. In the remainder of the chapter, when confusion is unlikely, we omit the adjective “symbolic” from symbolic solutions.

A constraint $S \models^T V$ is simple if $V \in \mathcal{V}_{msg}$, and $V$ is not a subterm of any term in $T$. This definition implies that $V$ can be instantiated with any term that can be inferred from $S$ under theory $T$. We remark that a policy constraint $S' \models^{T'} q'$, for CAP specifications in DC, can never be simple; the reason lies in the condition that variables appearing in a CAP specification are of type $msg$, while the knowledge of the processes of the CAP, ranges over terms of type $infon$; cf. Definition 1 and Section 5.2 on the construction of the constraint system. Hence this definition is applicable only to attacker constraints. A constraint system $C$ is trivial if and only if all constraints in $C$ are simple.

5.2 Generating constraint systems from interleavings

Let $\text{cap} = ((S, \Sigma, \mathcal{V}), \pi_1, \cdots, \pi_\ell, A)$ be a CAP in DC, with $\pi_i = (\eta_i, \Omega_i, I_i)$ for all $i \in \{1, \cdots, \ell\}$. A symbolic (i.e. non-ground) configuration of $\text{cap}$ is a tuple

$$((\eta_1, \Omega_1, I_1), \cdots, (\eta_\ell, \Omega_\ell, I_\ell), (\Omega_A, I_A)),$$
where \( \eta_1, \ldots, \eta_\ell \) are finite sets of threads, \( \Omega_1, \ldots, \Omega_\ell, \Omega^A \) are finite sets of terms and \( I_1, \ldots, I_\ell \) are finite sets of infon clauses and \( I^A \) is finite sets of message clauses. A symbolic configuration represents a state in an “abstract” execution of \( \text{cap} \). With respect to concrete configurations (see Section 2.2) symbolic configurations might contain variables; they can represent a (possibly infinite) set of concrete configurations, all those obtainable by grounding the variables in the symbolic configuration. In the following we omit the theories \( I_1, \ldots, I_\ell, I^A \) (i.e. the policies of the processes and the capabilities of the attacker), when they are clear from the context.

Let \( \mathcal{Z} \) be the set of all symbolic configurations of \( \text{cap} \), i.e. the set of all tuples of the form \( z = (\pi_1, \ldots, \pi_\ell, A) \). We define the relation \( \rightarrow \subseteq \mathcal{Z} \times E \times \mathcal{Z} \) as:

\[
(z, e, z') \in \rightarrow \iff z = (\pi_1, \ldots, \pi_\ell, A), z' = (\pi_1', \ldots, \pi_\ell', A'), A = (\Omega^A, I^A), \pi_i = (\eta, \Omega, I) \quad \text{and} \quad \gamma = e \cdot \gamma' \in \eta \quad \text{for some thread} \ \gamma' \text{ and } i \in \{1, \ldots, \ell\}, \text{and one of the following conditions hold:}
\]

- \( e = g_3 g_2 \triangleright \text{snd}(t), A' = (\Omega^A \cup \{t\}, I^A) \) and \( \pi_i' = (\eta \setminus \{\gamma\} \cup \{\gamma'\}, \Omega, I) \);
- \( e = \text{rcv}(t) \triangleright u + u_, A' = A \) and \( \pi_i' = (\eta \setminus \{\gamma\} \cup \{\gamma'\}, \Omega \cup u_, I) \).

Recall that negative updates are here deliberately ignored; that is, for all events of the form \( \text{rcv}(t) \triangleright u + u_- \) we assume \( u_- = \emptyset \). Negative updates are treated in detail in Section 5.7.

We write \( z \rightarrow z' \) for \( (z, e, z') \in \rightarrow \). An interleaving of \( \text{cap} \) is an alternating sequence of symbolic configurations and events \( \iota = z_0 e_1 z_1 \cdots z_{n-1} e_n z_n \) such that \( z_0 \) is the initial configuration of \( \text{cap} \) and, for all \( i \in \{1, \ldots, n\}, z_{i-1} \xrightarrow{e_i} z_i \). As opposed to traces, in an interleaving the variables introduced by an event are not instantiated, but carried over to the subsequent symbolic configurations, thus symbolically representing a (possibly infinite) set of concrete traces. Also in contrast to traces, interleavings are finitely many, because \( \text{cap} \) specifications contain finitely many threads, each of finite length.

A substitution \( \sigma : V \rightarrow \mathcal{T}_{E(0)} \) realizes an interleaving \( \iota \) of \( \text{cap} \) if \( i \sigma \) is a trace of \( \text{cap} \). We say that an interleaving \( \iota \) is realizable iff there exists at least one substitution \( \sigma \) that realizes \( \iota \).

Let \( \iota = z_0 e_1 \cdots z_{i-1} e_i z_i \cdots e_n z_n \) be an interleaving of \( \text{cap} \). To simplify the presentation, in the remainder of the chapter we assume that for all events \( e_i \) in \( \iota \) (with \( i \in \{1, \ldots, n\} \)), if \( e_i = g_3 g_2 \triangleright \text{snd}(m) \) then either \( g_3 = \{a\} \) and \( g_2 = \emptyset \), or \( g_3 = \emptyset \) and \( g_2 = \{a\} \); that is, only one of \( g_3 \) and \( g_2 \) is non-empty, and it is a singleton set. Similarly, we will assume goals of the form \( G = \pi : f \), for \( \pi \in \{\pi_1, \ldots, \pi_\ell\} \) and \( f \in \mathcal{T}_{E(0)}^{\text{infon}} \) or \( f = \mathcal{T}_{E(0)}^{\text{msg}} \). We discuss how more complex guards and goals can also be represented in the constraint system \( \text{cs}(\iota, G) \) in Remark 3, below.

Now we describe how \( \text{cs}(\iota, G) \) is generated for interleaving \( \iota \) and goal \( G \). Intuitively, each event of the interleaving \( \iota \) gives rise to one constraint in the constraint system \( \text{cs}(\iota) \). Let \( e \) be a guarded send \( e = g_3 g_2 \triangleright \text{snd}(t) \), executed at configuration \( z \) by thread \( p \) belonging to process \( \pi \). If \( g_3 = \{a\} \) then the positive constraint
\[ \Omega_\pi \vdash \pi \] a over the policy of \( \pi \) is added to the constraint system, checking that term a evaluates to true against the policy of \( \pi \) at configuration \( z \). If \( g_2 = \{ a \} \) then a negative constraint \( \Omega_\pi \not\vdash \pi \) over the policy of \( \pi \) is added to the constraint system, checking that the term a evaluates to false against the policy of \( \pi \) at configuration \( z \). We refer to such constraints as (respectively positive and negative) policy constraints, since they ascertain the satisfaction of policy requirements. For a receive event \( e = \text{rcv}(t) \rightarrow u \), executed at configuration \( z \), the constraint \( \Omega_A \vdash_\pi^G t \) over the deduction theory of the attacker is added to the constraint system, checking that the term \( t \) can be generated from the attacker’s knowledge set \( \Omega_A \) at configuration \( z \). We refer to such constraints as attacker constraints, as they model the messages that the attacker should construct from the ones she knows. Notice that the attacker constraints in \( cs(i, G) \) are always positive.

Additionally, one constraint is added to the constraint system to model the goal \( G \). If \( G = \pi : f \), with \( \pi \in \{ \pi_1, \cdots, \pi_\ell \} \) and \( f \in T_{\Sigma(\emptyset)}^{\text{info}} \), we add the policy constraint \( \Omega_\pi \vdash \pi f \), where \( \Omega_\pi \) is the extensional knowledge of process \( \pi \) at configuration \( z_n \) (i.e., the configuration reached after full execution of the interleaving). If \( G = A : m \) with \( m \in T_{\Sigma(\emptyset)}^{\text{msg}} \), we add the attacker constraint \( \Omega_A \vdash_\pi^G m \), where \( \Omega_A \) is the knowledge set of the attacker at configuration \( z_n \).

Formally, for an interleaving \( \iota = z_0 e_1 z_1 \cdots z_{n-1} e_n z_n \) of \( \text{cap} \) and a goal \( G \), the constraint system \( cs(i, G) \) generated is \( cs(i) \cup cs(G) \), that is, the union of the constraints generated to model the interleaving (\( cs(i) \)) and the constraints generated to model the goal (\( cs(G) \)); both are defined below:

\[
\begin{align*}
\text{cs}(i) &= \begin{cases}
\emptyset & \text{if } i = \epsilon \text{ (empty interleaving)} \\
\{ \Omega_\pi \vdash_\pi^a \} \cup \text{cs}(i') & \text{if } i = ze', e = \{ a \} \vdash \text{snd}(t), e \text{ is an event of process } \pi \\
\{ \Omega_\pi \vdash_\pi^a \} \cup \text{cs}(i') & \text{if } i = ze', e = \{ a \} \vdash \text{snd}(t), e \text{ is an event of process } \pi \\
\{ \Omega_A \vdash_\pi^G t \} \cup \text{cs}(i') & \text{if } i = ze' \text{ and } e = \text{rcv}(t) \rightarrow u
\end{cases}
\end{align*}
\]

\[
\text{cs}(G) = \begin{cases}
\{ \Omega_\pi \vdash_\pi^f \} & \text{if } G = \pi : f \\
\{ \Omega_A \vdash_\pi^G m \} & \text{if } G = A : m
\end{cases}
\]

where \( \Omega_i \) denotes the symbolic extensional knowledge of \( i \in \{ \pi_1, \cdots, \pi_\ell, A \} \) at configuration \( z \). Since each CAP results in a finite number of interleavings, there are finitely many constraint systems generated for each CAP.

Observe that the constraint system \( cs(i, G) \) has the following properties (cf. [23]):

- **Attacker monotonicity.** Let \( \text{att}(cs(i, G)) = \{ T_1 \vdash_\pi^A u_1, \cdots, T_\ell \vdash_\pi^A u_\ell \} \).
  The term sets \( T_1, \cdots, T_\ell \) are totally ordered by set inclusion.

- **Attacker origination.** Let \( T \vdash_\pi^A u \) be a constraint in \( \text{att}(cs(i, G)) \), and \( V \in \text{var}(T) \cap \mathcal{V}_a \). Then there exists \( T_V \vdash_\pi^A u_V \in \text{att}(cs(i, G)) \) such that \( T_V = \min(\{ T' | T' \vdash_\pi^A u' \in \text{att}(cs(i, G)), V \in \text{var}(u') \}) \) and \( T_V \subset T \).

\[ ^1 \text{We remark that even though we do not consider partial executions here (e.g. as in [24]) it is immediate to account for them: one possibility is to consider variants of the constraint system generated where the constraint } \Omega \vdash \pi \text{ that models the goal is replaced with a constraint } \Omega' \vdash \pi \text{, where } \Omega' \text{ is the extensional knowledge of } \pi \text{ at configuration } z_i, \text{ with } i \in \{ 0, \cdots, n \}.\]
Remark 3. In the construction of the constraint system $cs(\iota, G)$, to simplify presentation, we have assumed for all guarded send events occurring in the CAP specification (and, consequently, in the interleaving $\iota$) are of the form $g_3 \triangleright \text{snd}(m)$ where exactly one of $g_3$ and $\neg g_3$ is non-empty, and it is a singleton. Adapting $cs(\iota, G)$ to general guards is straightforward: let

\[ \{p_1, \ldots, p_n\} \{q_1, \ldots, q_m\} \triangleright \text{snd}(m) \]

be a guarded send event performed by a thread of process $\pi$, and $\Omega$ be the extensional knowledge of $\pi$ at the configuration in which the guard is to be evaluated. Then the evaluation of the guard can modeled in the constraint system $cs(\iota, G)$ by including the constraints

\[
\begin{align*}
\Omega & \models^{\pi} p_1 \\
& \quad \vdots \\
\Omega & \models^{\pi} p_n \\
\Omega & \not\models^{\pi} \{q_1, \ldots q_m\}
\end{align*}
\]

Observe also that for a guarded send event $\{\} \{\} \triangleright \text{snd}(m)$ procedure $cs$ generates no constraint, reflecting the unconditional transmission of term $t$ to the network.

(end of remark)

The constraint system $cs(\iota, G)$ is input to solve, shown in Algorithm 2. Procedure solve tries to reduce $cs(\iota, G)$ into a trivial constraint system by invoking the reduction procedures $\text{reduceDY}$, $\text{solvePos}$ and $\text{solveNeg}$. In the following sections, we discuss in detail these procedures.

5.3 Solving attacker constraints

The set $\text{att}(cs(\iota, G))$ of attacker constraints in $cs(\iota, G)$ is solved by $\text{reduceDY}$, shown in Figure 5.1. Procedure $\text{reduceDY}$ is an adaptation of the non-deterministic constraint reduction system formerly presented in [23] to decide the problem of reachability in security protocols under the usual Dolev-Yao threat model (see Section 4.1).

Intuitively, $\text{reduceDY}$ non-deterministically applies the rules of Figure 5.1 repeatedly on a constraint system $CS$ containing only attacker constraints until a maximally reduced constraint system $CS'$ is reached. A constraint system $CS'$ is maximally reduced iff no reduction rule is applicable to $CS'$; that is, there exist no substitution $\tau$ such that $CS' \sim_{\tau} CS''$, for some constraint system $CS''$. A maximally reduced constraint system is trivial iff all its constraints are of the form $T \models^{A} X$, with $X \in V_{\text{msg}}$. A trivial constraint system can be immediately solved when the constraint system has the monotonicity and origination properties (cf. Section 5.2); this is always the case for attacker constraint systems stemming from the constraint system generation procedure $cs$. If $CS'$ is trivial then $CS$ is
solvable. However, if $CS'$ is not trivial, then $reduceDY$ backtracks, i.e. it tries to apply on $CS$ another sequence of the rules of Figure 5.1. There are finitely many possible reduction paths.

$$(R_1) \ C \cup \{T \parallel^A u\} \leadsto C \quad \text{if } T \cup \{x \mid T' \parallel^A x \in C, T' \subset T\} \vdash^A u$$

$$(D_1) \ C \cup \{T \parallel^A u\} \leadsto C \cup \{T \parallel^A t, T' \parallel^A v\} \quad \text{if } u = f(t, v) \text{ and } f \in \{(,),\{,\},\{\},\}\text{, } \text{sig}(\cdot,\cdot)$$

$$(D_2) \ C \cup \{T \parallel^A u\} \leadsto C \cup \{T \parallel^A v\} \quad \text{if } u = h(v)$$

$$(U_1) \ C \cup \{T \parallel^A u\} \leadsto C \cup \{T \parallel^A ut\} \quad \text{if } \tau = mgu(u, t), t \in \text{sub}(T), t \neq u, t, u \notin V_a$$

$$(U_2) \ C \cup \{T \parallel^A u\} \leadsto C \cup \{T \parallel^A ur\} \quad \text{if } \tau = mgu(t, v), t, v \in \text{sub}(T), t \neq v, t, v \notin V_a$$

$$(U_3) \ C \cup \{T \parallel^A u\} \leadsto C \cup \{T \parallel^A ur\} \quad \text{if } \tau = mgu(t, v), \{w\}_{sk(i)}, \text{sk}(v) \in \text{sub}(T), t \neq v, t \in V_a \vee v \in V_a$$

Below, we briefly explain each of the reduction rules of Figure 5.1. Rule $R_1$ removes a constraint $T \parallel^A u$. It assumes a decision procedure for the DY ground deduction problem $\vdash^A$, reflecting the DY attacker capabilities (see Figure 4.1). It is well-known that $\vdash^A$ is decidable; see, e.g., [13]. The constraint is removed if $T \cup \{x \mid T' \parallel^A x \in C, T' \subset T\} \vdash^A u$. Observe that here and in the remainder of the section we write $T \vdash^A u$, for a set of terms $T$ and a term $u$ not necessarily ground, iff $T \vdash u \sigma$ (under the DY attacker theory) for any substitution $\sigma$ grounding for $T \cup \{u\}$; in other words, when variables in $T$ and in $u$ are seen as constants. Rules $D_1$ and $D_2$ decompose a constraint $T \parallel^A u$ into simpler constraints, when the most external constructor used in $u$ is a function available to the attacker. Rules $U_1$ and $U_2$, given a constraint $T \parallel^A u$, unify some subterm of $T$ with, respectively, $u$ or another subterm of $T$. Finally, rule $U_3$ unifies two subterms of $T$, where the subterms represent a secret key known to the attacker and a corresponding public key used for asymmetric encryption.

The constraint reduction presented in [23] reduces attacker constraint systems that satisfy the following properties:

- **monotonicity.** A constraint system $\{T_1 \parallel^A u_1, \ldots, T_\ell \parallel^A u_\ell\}$ has the monotonicity property iff the term sets $T_1, \ldots, T_\ell$ are totally ordered by set inclusion.

- **origination.** A constraint system $CS = \{T_1 \parallel^A u_1, \ldots, T_\ell \parallel^A u_\ell\}$ has the origination property iff for any constraint $T \parallel^A u$ in $CS$ if $V \in \text{var}(T)$ then there exists $T_V \parallel^A u_V \in CS$ such that $T_V = \min(\{T' \mid T' \parallel^A u' \in CS, V \in \text{var}(u')\})$ and $T_V \subset T$.

Let $CS$ be a constraint system generated by the procedure $cs$ (see Section 5.2). Then $\text{att}(CS)$ has the monotonicity property (immediate implication of the attacker monotonicity property, see Section 5.2), but $\text{att}(CS)$ does not have the origination property, because guard originated variables can appear on the right.
hand side of constraints. In contrast to the procedure of [23], procedure \textit{reduceDY} reduces attacker constraint systems that satisfy the attacker monotonicity property and the attacker origination property; cf. Section 5.2.

In the following sections we prove the termination and correctness of the reduction procedure \textit{reduceDY}. A proof \( \Pi \) of \( S \vdash a \), with \( S \subseteq \mathcal{T}_{\Sigma(\emptyset)} \), \( S \) finite, \( a \in \mathcal{T}_{\Sigma(\emptyset)} \) and \( \vdash; 2^{\mathcal{T}_{\Sigma(\emptyset)}} \times \mathcal{T}_{\Sigma(\emptyset)} \) being a ground deduction relation, is called normal if no label \( S' \vdash a' \) appears more than once on the same branch of \( \Pi \). Given a sequence \( T_1 \subseteq T_2 \cdots T_{n-1} \subseteq T_n \) (totally ordered with respect to set inclusion) of finite subsets of \( \mathcal{T}_{\Sigma(\emptyset)} \), we say that a proof \( \Pi \) of \( T_i \vdash a \), with \( 1 \leq i \leq n \), is left-minimal if \( \Pi \) does not hold when substituting \( T_i \) with \( T_j \), for any \( 1 \leq j < i \), in all labels of \( \Pi \).

We recall Tarski’s conditions for consequence relations, which hold in particular for \( \vdash^A \):

1. \textit{Set membership:} if \( u \in S \), then \( S \vdash^T u \);
2. \textit{Monotonicity:} if \( S \vdash^T u \) and \( S \subseteq S' \), then \( S' \vdash^T u \);
3. \textit{Consequence:} if \( S \vdash^T u \) and \( S \cup \{ u \} \vdash^T v \), then \( S \vdash^T v \).

In the following we will write \( \text{sub}(t) \) to denote the set of subterms of a term \( t \).

### 5.3.1 Termination

**Theorem 2.** The procedure \textit{reduceDY} terminates for all inputs generated from CAP specifications in the DC fragment.

**Proof.** Associate to any positive constraint system \( C \) a pair of non-negative integers \( \nu(C) = (n, m) \) where \( n \) is the number of variables appearing in \( C \) and \( m \) is the number of function applications and constants in the right hand sides of the constraints in \( C \) (also counting repetitions). Consider a lexicographic ordering on pairs \( \leq_{\text{lex}} \). We show that, for any two constraint systems \( C \) and \( C' \), \( C \simbowtie_{\tau} C' \) implies \( \nu(C) >_{\text{lex}} \nu(C') \). In fact, rules \( U_1 \), \( U_2 \) and \( U_3 \) decrease \( n \) strictly. All other rules, while not increasing \( n \), decrease \( m \) strictly.

### 5.3.2 Soundness

**Theorem 3.** The procedure \textit{reduceDY} is sound. That is, for two positive constraint systems \( C \) and \( C' \) such that \( C \simbowtie_{\tau} C' \) and a solution \( \rho \) of \( C' \), \( \tau \rho \) is a solution of \( C \).

**Proof.** Let \( C \) and \( C' \) be two constraint systems such that \( C \simbowtie_{\tau} C' \), and \( \rho \) be a solution of \( C' \). We show that \( \tau \rho \) is then a solution of \( C \).

We condition on the rule applied:
• \((R_1)\). Let \(T \vdash^A u\) be the constraint eliminated from \(C\). Hence, \(C' = C \setminus \{ T \vdash^A u \}\). Since \(\tau = \emptyset\), \(\rho\) is a solution for all constraints \((T' \vdash^A u') \neq (T \vdash^A u)\) in \(C\). We show then that \(T \rho \vdash^A u\). Recall that \(T \cup \{ x \mid T' \vdash^A x \in C, T' \subset T \} \vdash^A u\), and consequently also \(T \rho \cup \{ x_\rho \mid T' \vdash^A x, T' \subset T \} \vdash^A u\). By Tarski’s monotonicity property, \(T' \rho \vdash^A x \rho\) (with \(T' \subset T\)) implies \(T \rho \vdash^A x \rho\). This is the case for all constraints in \(C\) whose left hand side is smaller than \(T\). Then, by Tarski’s consequence property, \(T \rho \cup \{ x_\rho \mid T' \vdash^A x, T' \subset T \} \vdash^A u\) and \(T \rho \vdash^A u\) (for all \(T' \vdash^A x\) in \(C\) with \(T' \subset T\)) implies \(T \rho \vdash u\).

• \((D_1)\) and \((D_2)\). Let \(T \vdash^A f(u, v)\) be the constraint in \(C\), replaced in \(C'\) by \(T \vdash^A u\) and \(T \vdash^A v\). Since \(T \rho \vdash^A u\) and \(T \rho \vdash^A v\), \(T \rho \vdash^1 u\). \(T \rho \vdash^1 u\) follows from the corresponding inference rule in the attacker model (cf. Figure 4.1).

• \((U_1)\), \((U_2)\) and \((U_3)\). For any constraint \(T \vdash^A u \in C\) there is a constraint \(T \tau \vdash^A u \tau \in C'\). Since \(\rho\) is a solution of \(C'\), \(T \tau \rho \vdash^\pi (u \tau)\rho\), that is \(T (\tau \rho) \vdash^\pi u (\tau \rho)\).

\[\square\]

5.3.3 Completeness

Theorem 4. The procedure reduceDY is complete. That is, for a positive constraint system \(C\) that is not maximally reduced and a solution \(\omega\) of \(C\), there exist a constraint system \(C'\) and a solution \(\rho\) of \(C'\) such that \(C \leadsto^\pi C'\) and \(\omega = \tau \rho\).

Proof. Let \(C\) be a constraint system that is not maximally reduced, and \(\omega\) be a solution of \(C\). Then, there exists a constraint \(T \vdash^A u \in C\) whose right hand side is a non-variable term. Let \(T \vdash^A u\) be such that for all \(T' \vdash^A u' \in C\), with \(T' \subset T\), \(u' \in V\). Since \(\omega\) is a solution of \(C\), then \(T \omega \vdash^A u\omega\). Let \(\Pi\) be a normal left-minimal proof of \(T \omega \vdash^A u\omega\). We reason inductively on the structure of \(\Pi\), conditioning on the last rule applied. We consider the following cases:

• Composition. If the last rule applied in \(\Pi\) is a composition rule (cf. Table 4.1), then \(u\) is of the form \(u = f(x_1, \ldots, x_n)\), with \(f\) being a function available to the attacker (of arity \(n\)), and \(T \omega \vdash^A x_1 \omega, \ldots, T \omega \vdash^A x_n \omega\). Then rule \(D_1\) or \(D_2\) are applicable with \(\tau = \emptyset\), resulting in removing the constraint \(T \vdash^A u\) and introducing the constraints \(T \vdash^A x_1, \ldots, T \vdash^A x_n\). We need to show that \(\omega\) is a solution for the newly introduced constraints. This is immediate, because in \(\Pi\) we have \(T \omega \vdash^A x_1 \omega, \ldots, T \omega \vdash^A x_n \omega\).

• Decomposition or axiom. If the last rule applied in \(\Pi\) is the axiom or a decomposition rule (cf. Figure 4.1), then due to Lemma 1 there exists \(t \in \text{sub}(T), t \notin V_n\), such that \(t \omega = u\omega\).
If \( t \neq u \) then rule \( U_1 \) is applicable. Since \( \tau = \text{mgu}(u, t) \), by definition of \( \text{mgu} \) for any substitution \( \omega \) such that \( t\omega = u\omega \) there exists a substitution \( \rho \) such that \( \omega = \tau \rho \).

If \( t = u \), then \( u \in \text{sub}(T) \).

If there exists \( t' \in \text{sub}(T) \) such \( t' \notin \mathcal{V}_a \) and \( t' \neq t \), then rule \( U_2 \) is applicable. Since \( \tau = \text{mgu}(t, t') \), by definition of \( \text{mgu} \) there exists \( \rho \) such that \( \omega = \tau \rho \).

If there exist \( \{t_1, ...\} \text{pk}(t_2), \text{sk}(t_3) \in \text{sub}(T) \), with either \( t_2 \) or \( t_3 \) being an attacker originated variable, then rule \( U_3 \) is applicable and again by definition of \( \text{mgu} \) there exists \( \rho \) that solves \( C' \) and such that \( \omega = \tau \rho \).

If none of the above applies, then the conditions for Lemma 2 apply, from which \( T \cup \{x \mid (T' \models^A x) \in C, T' \subset T\} \vdash^A u \) follows. Consequently, rule \( R_1 \) is applicable.

Lemma 1. Let \( C \) be a constraint system that is not maximally reduced, \( \omega \) be a solution of \( C \), and \( T \) be a set of facts such that for all constraints \( T' \models^A x \in C \), with \( T' \subset T \), \( x \in \mathcal{V} \). If there exists a normal left-minimal proof of \( T\omega \vdash^A u\omega \) whose last rule is a decomposition rule or the axiom rule, then there exists \( t \in \text{sub}(T) \) such that \( t\omega = u\omega \) and \( t \notin \mathcal{V}_a \).

Proof. It is obvious that if the last rule in a normal left-minimal proof of \( T\omega \vdash^A u\omega \) is a decomposition rule then there exists \( t \in \text{sub}(T) \) such that \( t\omega = u\omega \).

It remains to prove that \( t \notin \mathcal{V}_a \). By contradiction, assume \( t \in \mathcal{V}_a \). Then, by attacker origination property (cf. Section 5.2), there exists \( T_i \models^A t \in C \) such that \( t \notin \text{sub}(T_i) \) and \( T_i \subset T \). But then \( T_i\omega \vdash^A t\omega = u\omega \), which contradicts the left-minimality of \( T\omega \vdash^A u\omega \).

In the following we write \( \text{pv}_C(T) \) (standing for “previous variables”), for a constraint system \( C \) and a finite set of terms \( T \), to denote the set \( \{x \mid (T' \models^A x) \in C, T' \subset T\} \). Lemma 2 has been proved in [23]. We give a slightly modified (and simpler) proof here, that also accounts for the less restrictive property of attacker origination.

Lemma 2. Let \( C \) be a constraint system, \( \omega \) be a solution of \( C \), \( u \) be a term and \( T \) be the left hand side of a constraint in \( C \), such that:

1. for any constraint \( T' \models^A x \) in \( C \), with \( T' \subset T \), \( x \) is a variable
2. there are no two distinct terms \( t_1, t_2 \in \text{sub}(T) \) such that \( t_1, t_2 \notin \mathcal{V}_a \) and \( t_1 \omega = t_2 \omega \)
3. there are no two terms \( \{t_1, ...\} \text{pk}(t_2), \text{sk}(t_3) \in T \) such that either \( t_2 \in \mathcal{V}_a \) or \( t_3 \in \mathcal{V}_a \) and \( t_2 \neq t_3 \)
4. $u$ is a subterm of $T$ such that $u \notin V_a$

5. $T \omega \vdash^K u \omega$

Then $T \cup pv_C(T) \vdash^K u$.

**Proof.** Firstly, we remark that due to condition (2) no guard originated variable appears in $T$ (and, as a consequence of (4), neither in $u$); that is $\text{var}(T \cup \{u\}) \cap V_g = \emptyset$. In absence of ambiguity, in the remainder of the proof we will refer to variables instead of attacker originated variables.

Consider a normal left-minimal proof $\Pi$ of $T \omega \vdash^K A u \omega$. Without loss of generality we consider $T$ as the minimal left hand side of the constraints in $C$ for $\Pi$ to be left-minimal (alternatively $T$ can be replaced in $\Pi$ with the minimal $T'$ for which $\Pi$ still holds). We prove the lemma by induction on the depth of $\Pi$, conditioning on the nature of the last rule applied in $\Pi$.

**Axiom or decomposition rule.** If $\Pi$ ends with a decomposition rule or an axiom, then there is a term $t$ such that $T \omega \vdash^K t \omega$ and possibly another term $v$ (again such that $T \omega \vdash^K v \omega$) that are necessary for the application of the last rule in $\Pi$. Namely,

- $t = t'$, if the last rule applied is an axiom
- $t = (t', t'')$ or $t = (t'', t')$, if the last rule applied is unpairing
- $t = \{t'_p\}_{pk(v')}$ and $v = \text{sk}(v'')$, with $v' \omega = v'' \omega$, if the last rule applied is asymmetric decryption
- $t = \{|t'_p|\}_{v'}$ and $v = v''$, with $v' \omega = v'' \omega$, if the last rule applied is symmetric decryption

where $t' \omega = u \omega$. Due to normality, $t$ cannot be the result of a compositional rule in $\Pi$; hence $t \in \text{sub}(T)$.

Assume, by contradiction, that $t' \in \mathcal{V}$. Then by attacker origination there exists $T_v \vdash^K t'$ in $C$, with $T_v \subset T$, of which $\omega$ is a solution. But then $T_v \omega \vdash^K t' \omega = u \omega$, contradicting the left-minimality of $\Pi$. Hence $t'$ can not be a variable. On the other hand, due to conditions (2) and (4), $t'$ can not be a non-variable term distinct from $u$. This implies that $t' = u$. In the case of an axiom or of unpairing, we can then conclude that $T \cup pv_C(T) \vdash^K u$.

In the case of asymmetric decryption also term $v = \text{sk}(v'')$ is needed. Since $\text{sk}(v'')$ cannot be the result of the application of a compositional rule in $\Pi$, it must be a subterm of $T$. Due to condition (3) there can not exist $t = \{t'_p\}_{pk(v')}$ and $v = \text{sk}(v'')$ with at least one of $v'$ and $v''$ being a variable distinct from the other. It follows that $v' = v''$, and therefore that the asymmetric decryption rule is applicable to $t$ and $v$ thus obtaining $u$. Hence, $T \cup pv_C(T) \vdash^K u$.

In the case of symmetric decryption also term $v''$ is needed (the key used for encryption in $t$), with $T \omega \vdash^K v'' \omega$. Due to condition (2) either $v' = v''$, or one of
the two is a variable. In the former case, the claim of the lemma is immediate. We first consider the case \( v' \) is a variable, and then turn to the case \( v' \) is not a variable (thus \( v'' \) must be a variable).

- If \( v' \) is a variable then, by attacker origination property (see Section 5.2), there exists \( T_{v'} \vdash^A v' \) in the constraint system \( C \) where \( T_{v'} \subseteq T \). Therefore, \( T \cup pv_C(T) \vdash^A v' \). Then by applying the symmetric decryption rule we conclude that \( T \cup pv_C(T) \vdash^A u \).

- Now, suppose \( v'' \) is a variable and \( v' \) is not a variable. By the proof \( \Pi \) of \( T \omega \vdash^A u \omega \) we have \( T \omega \vdash^A v'' \omega \). Then, \( v' \omega = v'' \omega \) implies \( T \omega \vdash^A v' \omega \). We remark that \( v' \) is not a variable. Therefore, by induction hypothesis, we have \( T \cup pv_C(T) \vdash^A v' \). Then applying the symmetric decryption rule yields \( T \cup pv_C(T) \vdash^A u \).

**Composition rule.** If \( \Pi \) ends with a composition rule, then \( u \) (being a non-variable term) has a definite form. Namely,

- \( u = (v_1, v_2) \), if the last rule applied is pairing
- \( u = h(v_1) \), if the last rule applied is hashing
- \( u = \{ v_1 \}_{v_2} \), if the last rule applied is asymmetric encryption
- \( u = \{| v_1 |\}_{v_2} \), if the last rule applied is symmetric encryption

Consider any term \( v \) needed to construct \( u \). If \( v \in V_a \) then \( v \) appears in \( T \) due to condition (4). Then, by attacker origination, there exists \( T_v \vdash^A v \) in \( C \), with \( T_v \subseteq T \), hence \( T \cup pv_C(T) \vdash^A v \). If \( v \) is not a variable, then by induction hypothesis \( T \cup pv_C(T) \vdash^A v \). Therefore, applying the corresponding composition rule, \( T \cup pv_C(T) \vdash^A u \).

\[ \square \]

### 5.4 Solving positive policy constraints

In this section, we present a proof search procedure for AL theories. The procedure, called \( ps \) (standing for proof search), is shown in Figure 5.2. The procedure \( ps \) is the heart of the procedure \( solvePos \) used by procedure \( solve \) to solve positive policy constraints: \( solvePos \) calls repeatedly \( ps \) for each constraint in \( pos(cs(t, G)) \), to find a solution \( \tau \) that solves all of them simultaneously.

Consider an AL theory \( T = T_w \cup T_\rightarrow \) (see Definition 1), and write \( \vdash \) for the ground deduction relation induced by \( T \). As a convention, we write rules in \( T_\rightarrow \) as \( t \leftarrow t_1, \cdots, t_n, a \) so that the anchor is the last premise (in this case, \( a \)) of the rule. The procedure \( ps \) is a non-deterministic constraint reduction system. The input to \( ps \) is a finite set of positive constraints over \( T \); typically, a singleton constraint system \( C_0 = \{ \Omega \vdash^A u \} \). Intuitively, \( ps \) guesses tentative proof trees for
Ωτ ⊢ uτ, with τ being a solution of Ω ⊨ u. The constraint system created and further reduced by ps represents the set of (not yet discharged) leaves for the tentative proof tree. If ps can discharge all the leaves, then ps outputs the solution τ. However, if ps fails to discharge all the leaves of the tentative proof, then ps backtracks. There are finitely many guesses that ps can make (this is stated by Theorem 5 below). If ps fails to discharge the leaves (i.e. assumptions) for all the (guessed) tentative proofs, then ps returns a failure, indicating ¬∃τ. Ωτ ⊢ uτ.

1. \( C \cup \{ S \sqcup \{ t \} \mid \vdash u \} \sim_\tau C' \) if \( \tau = \text{mgu}(u, t) \)

2. \( C \cup \{ S \sqcup \{ z \} \mid \vdash u \} \sim_\tau (C \cup \{ S \mid \vdash t_1, \ldots, S \mid \vdash t_n, S \sqcup \{ t \} \mid \vdash u \})' \)
   
   if \( \tau = \text{mgu}(z, a) \) for some \( t \leftarrow t_1, \ldots, t_n, a \in T \rightarrow \)

3. \( C \cup \{ S \mid \vdash u \} \sim_\tau (C \cup \{ S \mid \vdash t_1, \ldots, S \mid \vdash t_n \})' \)
   
   if \( \tau = \text{mgu}(u, t) \) for some \( t \leftarrow t_1, \ldots, t_n \in T \leftarrow \)

Figure 5.2: Constraint reduction system for AL proof search

The procedure ps applies non-deterministically one of the constraint reduction rules (shown in Figure 5.2) as long as the constraint system is not empty. The reduction rules reflect the infon clauses in \( \mathcal{T} \) and the axiom rule \( \mathcal{T} \cup \{ u \} \vdash u \). We write \( C \sim_\tau C' \) to denote that the constraint system \( C \) is reduced to the constraint system \( C' \) by applying the partial substitution \( \tau : \text{var}(C) \rightarrow T_{\Sigma(V)} \).

- Rule (1) discharges one of the leaves using unification. For a constraint \( S \sqcup \{ t \} \mid \vdash u \in C \), if \( u \) can be unified with a term \( t \) in the left hand side of the constraint, then the constraint is removed, and their most general unifier \( \tau \) is applied to the remaining (if any) constraints in \( C \).

- Rule (2) applies a rule \( R : t \leftarrow t_1, \ldots, t_n, a \in T \rightarrow \) to a constraint \( (S \sqcup \{ z \} \mid \vdash u) \in C \). The intuition behind the rule lies in the fact that the anchor \( a \) can only be found in the extensional knowledge or derived with another application of \( R \). If \( z \) unifies with \( a \), and \( \tau \) is the most general unifier of \( z \) and \( a \), then the constraint is replaced by \( n + 1 \) new constraints: the first \( n \) constraints, \( (S \mid \vdash t_1, \ldots, S \mid \vdash t_n) \tau \), check that the premises of the rule (except for \( a \)) can be proved from \( S \); the last constraint, \( (S \cup \{ t \} \mid \vdash u) \tau \), checks whether \( u \) can be proved knowing \( S \) and \( t \). Notice that \( z \) does no longer appear in the left hand side of the new constraints: this ensures termination of ps, and does not compromise its completeness (cf. Theorems 5 and 7).

- Finally, rule (3) applies a rule \( t \leftarrow t_1, \ldots, t_n \in T \leftarrow \) to a constraint \( (S \mid \vdash u) \in C \). If there exists a most general unifier \( \tau \) of \( u \) and \( t \), then the constraint is replaced by \( n \) constraints \( (S \mid \vdash t_1, \ldots, S \mid \vdash t_n) \tau \). The properties of \( T \leftarrow \) ensure that rule (3) can be applied only finitely many times (cf. Theorem 5).
The partial substitution $\tau$ is also applied to the remaining (if any) constraints in $C$.

Let $\sim^*$ be the reflexive and transitive closure of $\sim$. We write $C \sim^*_\tau C'$ to denote that there exist constraint systems $C_1, \ldots, C_k$ and substitutions $\tau_1, \ldots, \tau_k$ (for $k \geq 0$) such that $C \sim_{\tau_1} C_1 \sim_{\tau_2} \cdots \sim_{\tau_k} C_k, C_k = C'$ and $\tau = \tau_1 \cdots \tau_k$. If $C_0 \sim^*_\tau \emptyset$ then all leaves of a tentative proof tree have been discharged; hence $ps$ returns the solution $\tau$. If $C_0 \sim^*_\tau C'$, with $C'$ being a non-empty constraint system not further reducible, then the tentative proof at hand fails, and $ps$ backtracks (i.e. $ps$ tries different sequences of applications of the rules of Figure 5.2). If no sequence of rule applications leads to an empty constraint system, then $ps$ reports that the constraint system $C_0$ has no solutions.

Observe that $ps$ is non-deterministic, as there can be multiple solutions for the input constraint $\Omega \vdash u$. Intuitively, for any possible “proof skeleton” $\Pi$ of $\Omega \vdash u$, the procedure $ps$ returns a “maximal” solution $\tau$: if $\tau'$ is a solution such that $\Pi \tau'$ is a valid proof tree, and $\tau' \sqsubseteq \tau$, then $\tau \equiv \tau'$; that is, $\tau$ and $\tau'$ are equivalent modulo $\alpha$-renaming of the variables. There are finitely many such maximal solutions for any constraint $\Omega \vdash u$ (namely one for each possible proof skeleton).

In the following sections, we prove that the proof search procedure $ps$ is terminating, sound and complete. The proofs exploit the fact that every ground deduction relation $\vdash_T$ induced by a set $T$ of infon clauses respects Tarski’s conditions for consequence relations, namely:

1. **Set membership**: if $u \in S$, then $S \vdash_T u$;
2. **Monotonicity**: if $S \vdash_T u$ and $S \subseteq S'$, then $S' \vdash_T u$;
3. **Consequence**: if $S \vdash_T u$ and $S \cup \{u\} \vdash_T v$, then $S \vdash_T v$.

### 5.4.1 Termination

**Theorem 5.** The proof search procedure $ps$ terminates for all deduction problems in AL.

**Proof.** To prove termination of the $ps$ procedure we define two measures, $\omega_{\rightarrow}$ and $\omega_{\leftarrow}$, for terms of type infon containing only variables of type $msg$, as follows:

$$
\omega_{\rightarrow}(u) = \begin{cases}
\max_{R \in T_{\rightarrow}} \omega_{\rightarrow}(u) + 1 & \text{if } T_{\rightarrow} \neq \emptyset \\
0 & \text{otherwise}
\end{cases}
$$

$$
\omega_{\leftarrow}(u) = \begin{cases}
\omega_{\rightarrow}(t\tau) & \text{if } R = t \leftarrow t_1, \cdots, t_n, a, \tau = \text{mgu}(u, a) \\
0 & \text{otherwise}
\end{cases}
$$

$$
\omega_{\leftarrow}(u) = \begin{cases}
\max_{R \in T_{\leftarrow}} \omega_{\leftarrow}(u) + 1 & \text{if } T_{\leftarrow} \neq \emptyset \\
0 & \text{otherwise}
\end{cases}
$$

63
\[
\omega_R(u) = \begin{cases} 
\max_{i \in \{1, \ldots, n\}} \omega_R(t_i \tau) & \text{if } R = t \leftarrow t_1, \ldots, t_n, \tau = \text{mgu}(t, u) \\
0 & \text{otherwise}
\end{cases}
\]

Intuitively, \( \omega_R(u) \) is a measure of how many applications of rules in \( T_\rightarrow \) can stem from unifying a term \( u \) with the anchor \( a \) of a rule \( R \in T_\rightarrow \); we informally refer to it as the anchor potential of \( u \). We extend \( \omega_\rightarrow \) to sets of terms, so that \( \omega_\rightarrow \{u_1, \ldots, u_n\}\) = \( \sum_{i \in \{1, \ldots, n\}} \omega_\rightarrow(u_i) \).

The function \( \omega_\rightarrow \) is a measure of the length of the longest reduction sequence induced by \( u \), under theory \( T_\rightarrow \); we informally refer to it as the rewriting potential of \( u \).

The function \( \omega_\rightarrow \) is well defined due to the conditions on \( \text{AL} \) theories expressed in Definition 1: since the rewrite system \( \{a \Rightarrow t\} \) is terminating (for each \( t \leftarrow t_1, \ldots, t_n, a \in T_\rightarrow \)), there is no infinite sequence of rewriting steps. Observe that considering such rewrite system is sufficient as the anchors of other rules in \( T_\rightarrow \) can not unify with \( t \) (cf. Chapter 4). Similarly for function \( \omega_\rightarrow \), the rewrite system \( \text{R}_\rightarrow = \{t \Rightarrow t_1, \ldots, t_n \mid t \leftarrow t_1, \ldots, t_n \in T_\rightarrow \} \) is terminating, hence there is no infinite sequence of rewriting steps.

We now define a weight function \( \omega \) for constraints. For a constraint \( S \models u \), \( \omega(S \models u) = (\omega_\rightarrow(S), \omega_\rightarrow(u)) \); that is, the weight of \( S \models u \) is the pair whose first element is the anchor potential of the set \( S \) and whose second element is the rewriting potential of the term \( u \). We show that at each derivation step, if a constraint \( C \) is removed and constraints \( C_1, \ldots, C_k \) are introduced, \( \omega(C) > \omega(C_i) \) for all \( i \in \{1, \ldots, k\} \); according to the usual lexicographical order. We condition on the reduction rule applied:

1. \( S \cup \{t\} \models u \sim_\tau \emptyset \), with \( \tau = \text{mgu}(u, t) \). In this case the claim is trivially satisfied.

2. \( S \cup \{z\} \models u \sim_\tau S \tau \models t_1 \tau, \ldots, S \models t_n \tau, S \tau \cup \{t\} \models u \tau \), with \( \tau = \text{mgu}(z, u) \), for some \( R : t \leftarrow t_1, \ldots, t_n, a \in T_\rightarrow \). For any constraint \( S \tau \models t_i \tau \), with \( i \in \{1, \ldots, n\} \), by Lemma 3 \( \omega_\rightarrow(S \tau) \leq \omega_\rightarrow(S) \), and \( \omega_\rightarrow(S \cup \{z\}) > \omega_\rightarrow(S) \) because \( \omega_\rightarrow(z) > 0 \), hence \( \omega(S \cup \{z\} \models u) > \omega(S \tau \models u \tau) \). Similarly for the constraint \( S \tau \cup \{t\} \models u \tau \), \( \omega_\rightarrow(S \tau \cup \{t\} \tau) \) = \( \omega_\rightarrow(S \tau) \), and since \( \omega_\rightarrow(z) = 0 \) it is 1 by definition of \( \omega_\rightarrow \) then \( \omega(S \cup \{z\} \models u) > \omega(S \tau \models u \tau) \).

3. \( S \models u \sim_\tau S \tau \models t_1 \tau, \ldots, S \models t_n \tau \), with \( \tau = \text{mgu}(u, t) \) for some \( t \leftarrow t_1, \ldots, t_n \in T_\rightarrow \). Consider any constraint \( S \tau \models t_i \tau \), with \( i \in \{1, \ldots, n\} \). By Lemma 3 \( \omega_\rightarrow(S) \geq \omega_\rightarrow(S \tau) \). Also, by definition of \( \omega_\rightarrow \) and by Lemma 4, \( \omega_\rightarrow(u) > \omega_\rightarrow(t_i) \geq \omega_\rightarrow(t_i \tau) \). Therefore \( \omega(S \models u) > \omega(S \tau \models t_i \tau) \).

We consider now a set of constraints. We make the following observations to assert that the \( ps \) procedure terminates on the constraint set. Consider the derivation step \( CS \cup \{C\} \sim_\tau CS \tau \cup C' \):
• Following from Lemmas 3 and 4, \( \omega(CS) \geq \omega(CS\tau) \); that is, when a reduction step is performed on a constraint in the constraint system, and the resulting solution \( \tau \) applied to the remainder of the constraint system, the weight of the remainder of the constraint system is not increased.

• \( C' \) is always a finite set of constraints (obvious because of the finite number of premises of the rules in \( T \)), hence the derivation tree stemming from \( C \) is finitely branching.

• The derivation tree stemming from \( C \) has finite depth, which is implied by the fact that \( \omega(C) > \omega(c) \) for any constraint \( c \in C' \), shown above.

The considerations above show that all possible derivations of the constraint system \( CS \) terminate.

For the following lemmas, we observe that every application of a rule in \( T \) uses fresh variables, that is, they do not appear elsewhere. Consequently, they do not appear in the domain of any substitution \( \sigma \) prior to consideration of that instance of the rule; in particular, if \( t \) is a term appearing in a rule, \( t\sigma = t \).

**Lemma 3.** Let \( u \in T_{\Sigma}^{\text{Infon}}_{\Sigma(N_{\text{msg}})} \) be a term of type infon containing only variables of type msg, and \( \sigma \) be any well typed substitution. Then \( \omega_\rightarrow(u) \geq \omega_\rightarrow(u\sigma) \).

**Proof.** We prove the claim by induction on the value of \( \omega_\rightarrow(u) \). If \( \omega_\rightarrow(u) = 0 \), then there is no rule \( t \leftarrow t_1, \ldots, t_n, a \in T_\rightarrow \) and no substitution \( \tau \) such that \( u\tau = a\tau \). Assume, by contradiction, that \( \omega_\rightarrow(u\sigma) > 0 \). Then there should exist \( \tau \) such that \( u\sigma\tau = a\tau = a\sigma\tau \), hence \( \sigma\tau \) would be a unifier for \( u \) and \( a \), contradicting the hypothesis.

If \( \omega_\rightarrow(u) > 0 \), then \( \omega_\rightarrow(u) = \omega_\rightarrow(t\tau) + 1 \), for some rule \( t \leftarrow t_1, \ldots, t_n, a \in T_\rightarrow \) and \( \tau = \text{mgu}(u, a) \). We condition on whether \( u\sigma \) unifies with \( a \), for some \( t \leftarrow t_1, \ldots, t_n, a \in T_\rightarrow \). If there exists no \( \tau' \) such that \( u\sigma\tau' = a\tau' \), then \( \omega_\rightarrow(u\sigma) = 0 \leq \omega_\rightarrow(u) \). If there exists \( \tau' \) such that \( u\sigma\tau' = a\tau' \), then by the consideration above \( u\sigma\tau' = a\sigma\tau' \). Then \( \sigma\tau' \) is a unifier for \( u \) and \( a \), and therefore there exists \( \sigma' \) such that \( \sigma\tau' = \tau\sigma' \). Then \( \omega_\rightarrow(u\sigma) = \omega_\rightarrow(t\tau\sigma') + 1 = \omega_\rightarrow(t\tau\sigma') + 1 \).

By Lemma 5, \( \tau \) contains all variables of \( a \) and maps them to terms containing only variables of type msg; moreover, \( \text{var}(t) \subseteq \text{var}(a) \) (by definition of \( \text{AL} \), cf. Chapter 4). Then \( t\tau \) is a term containing only variables of type msg. By induction hypothesis, \( \omega_\rightarrow(t\tau\sigma') \leq \omega_\rightarrow(t\tau) \), so \( \omega_\rightarrow(u\sigma) = \omega_\rightarrow(t\tau\sigma') + 1 \leq \omega_\rightarrow(t\tau) + 1 = \omega_\rightarrow(u) \).

**Lemma 4.** Let \( u \in T_{\Sigma}^{\text{Infon}_{\Sigma(N_{\text{var}})}} \) be a term of type infon, and \( \sigma \) be any well typed substitution. Then \( \omega_\rightarrow(u) \geq \omega_\rightarrow(u\tau) \).

**Proof.** We prove the claim by induction on the value of \( \omega_\rightarrow(u) \). If \( \omega_\rightarrow(u) = 0 \), then there is no rule \( t \leftarrow t_1, \ldots, t_n \in T_\rightarrow \) and no substitution \( \tau \) such that \( u\tau = t\tau \).
Assume, by contradiction, that $\omega_\omega(u_\sigma) > 0$. Then there should exists $\tau$ such that $u_\sigma \tau = t_\tau = t_\sigma \tau$, hence $\sigma \tau$ would be a unifier for $u$ and $t$, contradicting the hypothesis.

Assume now that $\omega_\omega(u) > 0$. We condition on whether $u_\sigma$ unifies with $a$, for some $t \leftarrow t_1, \cdots, t_n \in T_\omega$. If there exists no $\tau'$ such that $u_\sigma \tau' = t_\tau'$, then $\omega_\omega(u_\sigma) = 0 \leq \omega_\omega(u)$. If there exists $\tau'$ such that $u_\sigma \tau' = t_\tau'$ for some rule $t \leftarrow t_1, \cdots, t_n \in T_\omega$; we assume, without loss of generality, that $t \leftarrow t_1, \cdots, t_n \in T_\omega$ is the rule with the premise $t_i$ for which $\omega_\omega(t_i \sigma_\tau')$ is maximal. By the consideration above $u_\sigma \tau' = t_\tau' = t_\sigma \tau'$, hence $\sigma \tau'$ is a unifier for $u$ and $a$; and there exists $\tau = \text{mgu}(u, t)$ and $\sigma'$ such that $\sigma \tau' = \tau_\sigma'$. By definition of $\omega_\omega$, $\omega_\omega(u) \geq \omega_\omega(t_i \tau) + 1$. Then $\omega_\omega(u_\sigma) = \omega_\omega(t_i \sigma \tau') + 1 = \omega_\omega(t_i \sigma') + 1$, and by induction hypothesis $\omega_\omega(t_i \sigma') \leq \omega_\omega(t_i \tau)$, hence $\omega_\omega(u_\sigma) \leq \omega_\omega(t_i \tau) + 1 \leq \omega_\omega(u)$. 

Lemma 5. Let $a \in T_{\text{infon}}^{\Sigma(V)}$ be a term of type infon and $z \in T_{\text{infon}}^{\Sigma(V_{\text{msg}})}$ be a term of type infon containing only variables of type msg. If there exists a substitution $\tau = \text{mgu}(a, z) = \{X_1 \mapsto t_1, \cdots, X_n \mapsto t_n\}$ then:

1. for all $X \in \text{var}(z) \cap \{X_1, \cdots, X_n\}$ $X \tau$ is ground;
2. $\text{var}(a) \subseteq \{X_1, \cdots, X_n\}$;
3. for all $X \in \text{var}(a)$, $\text{var}(X \tau) \cap V_{\text{infon}} = \emptyset$.

Proof. We start by proving (1). Let $X \in \text{var}(z) \cap \{X_1, \cdots, X_n\}$; then $X$ maps to a subterm of $a$ of type msg. $t$ can not be a variable, because variables in $a$ are of type infon; also, $t$ can not contain any variables, because terms of type msg can not contain variables of type infon (cf. Definition 1).

We now prove (2). Assume, by contradiction, that there exists $X \in \text{var}(a)$ such that $X \notin \{X_1, \cdots, X_n\}$. But then there must exist $Y \in \{X_1, \cdots, X_n\} \cap \text{var}(z)$ such that $X \in \text{var}(Y \tau)$ (i.e. $X$ is a subterm of the subterm of $z$ that $Y$ maps to); this is not possible because (1) shows that $Y$ must map to a ground term. Therefore, $X \in \{X_1, \cdots, X_n\}$.

Finally, (3) follows trivially from (2) and the assumption that $z$ does not contain any variable of type infon.

5.4.2 Soundness

Theorem 6. The proof search procedure $ps$ is sound. That is, for two constraint systems $C$ and $C'$ and a substitution $\rho$ that is a solution for $C'$, if $C \sim_\tau C'$ then $\tau \rho$ is a solution for $C$.

Proof. We condition on the rule applied:
The proof goes by induction on the depth of the constraint system $C$.

The proof search procedure:

**Theorem 7.** The proof search procedure $ps$ is complete. That is, for a (reducible) constraint system $C$ and a substitution $\omega$ that is a solution for $C$, there exists a constraint system $C'$ such that $C \leadsto C'$ and $\tau \subseteq \omega$.

**Proof.** Consider a constraint $S \vdash u \in C$, and a proof $\Pi$ of $S\omega \vdash u\omega$. The proof goes by induction on the depth of $\Pi$. We condition on the last rule applied in $\Pi$:

- If the last rule applied is the axiom (i.e. the conclusion $u\omega$ belongs to the set of axioms $S\omega$), then $u\omega = u\omega \in S\omega$. Therefore rule (1) of $ps$ is applicable, and since $\tau = mgu(u, t)$ by definition of $mgu$ there exists a substitution $\rho$ such that $\omega = \tau \rho$.

- If the last rule applied is $t \leftarrow t_1, \ldots, t_n, a \in T_\rightarrow$ (with $a$ being the anchor of the rule), then due to Lemma 6 rule (2) of $ps$ is applicable for a term $z \in S$ such that $z\omega = a\omega$. In this case the constraint $S \vdash u$ is replaced by the constraints $S'\tau \vdash t_1\tau, \ldots, S'\tau \vdash t_n\tau, S'\tau \vdash \{t\} \vdash u\tau$, with $S' = S \setminus \{z\}$. Since $\tau = mgu(a, z)$ a solution $\omega$ refines $\tau$. Again due to Lemma 6 $S'\omega \vdash t_1\omega, \ldots, S'\omega \vdash t_n\omega$ and $S'\omega \vdash \{t\} \vdash u\omega$, with $S' = S \setminus \{z\}$.

- If the last rule applied is a rule $p \leftarrow p_1, \ldots, p_n \in T_\rightarrow$, then $S\omega \vdash p_1\omega, \ldots, S\omega \vdash p_n\omega$ and $u\omega = p\omega$. Rule (3) is then applicable. Since $\tau = mgu(u, p)$,
by definition of \( mgu \) there exists \( \rho \) such that \( \omega = \tau \rho \). Since \( S\omega \vdash p_1 \tau \rho, \ldots, S\omega \vdash p_n \tau \rho \) then \( \rho \) is a solution for the constraints \( S\tau \vdash p_1 \tau, \ldots, S\tau \vdash p_n \tau \).

\[
\]

For the following lemmas, we recall that a proof \( \Pi \) is normal if a label \( A \vdash c \) never appears more than once on the same branch of \( \Pi \).

**Lemma 6.** Let \( T \) be an \textit{AL} theory, \( \vdash \) the ground deduction relation induced by \( T \), \( S \) a set of terms, \( u \) a term, and \( \omega \) a substitution such that \( S\omega \vdash u\omega \). If there exists a normal proof of \( S\omega \vdash u\omega \) ending with an instance \( t' \leftarrow t_1', \ldots, t_n', a' \) of rule \( R : t \leftarrow t_1, \ldots, t_n, a \in T_a \), then there exists an instance \( t'' \leftarrow t_1'', \ldots, t_n'', a'' \) of rule \( R \) and a term \( z \in S \) such that \( z\omega = a''\omega \), \( S'\omega \vdash t_1''\omega, \ldots, S'\omega \vdash t_n''\omega \), and \( S'\omega \cup \{t''\omega\} \vdash u\omega \), with \( S' = S \setminus \{z\} \).

**Proof.** Let \( \Pi \) be a normal proof of \( S\omega \vdash u\omega \) that ends with an instance \( t' \leftarrow t_1', \ldots, t_n', a' \) of rule \( R \). Then \( u\omega = t'\omega \) and \( S\omega \vdash t_1'\omega, \ldots, S\omega \vdash t_n'\omega, S\omega \vdash a'\omega \), which by property (3) of Tarski’s consequence relations implies \( S\omega \vdash t'\omega = u\omega \).

Let there be \( z \in S \) such that \( z\omega = a'\omega \). Then, by Lemma 7, \( S'\omega \vdash t_1'\omega, \ldots, S'\omega \vdash t_n'\omega \), with \( S' = S \setminus \{z\} \). Also, \( S\omega = (S' \cup \{a'\})\omega \vdash t'\omega = u\omega \). Hence the statement of the lemma holds for \( a' = a'' \), \( t' = t'' \), \( t_1' = t_1'' \), \ldots, \( t_n' = t_n'' \).

Assume now there is no \( z \in S \) such that \( z\omega = a'\omega \), and \( \Pi' \) be a proof of \( S\omega \vdash a'\omega \). It is immediate to see that \( \Pi' \) can not end with the axiom rule. Furthermore, by definition of \textit{AL} (cf. Definition 2) \( a' \) can only unify with the head of rule \( R \). Then by induction hypothesis there exists an instance \( t'' \leftarrow t_1'', \ldots, t_n'', a'' \) of \( R \) and a term \( z \in S \) such that \( z\omega = a''\omega \), \( S'\omega \vdash t_1''\omega, \ldots, S'\omega \vdash t_n''\omega \), and \( S'\omega \cup \{t''\omega\} \vdash a'\omega \), with \( S' = S \setminus \{z\} \).

To conclude the proof it remains to show that \( S'\omega \vdash t_1''\omega, \ldots, S'\omega \vdash t_n''\omega \). Consider \( t_i' \) among \( t_1', \ldots, t_n' \). Assume by contradiction that \( S\omega \vdash t_i'\omega \) but \( (S \setminus \{z\})\omega \not\vdash t_i'\omega \). Then the subproof \( \Pi' \) of \( S\omega \vdash t_i'\omega \) has in its leaves (at least) an application of the axiom rule unifying \( z \) with a premise \( p \) of a rule \( R' \) (in the context of proof \( \Pi \)). By definition of \textit{AL}, \( p \) can not be a premise of any rule in \( T \setminus \{R\} \), nor any of the premises \( t_1, \ldots, t_n \) in rule \( R \). Then \( p \) is the anchor of an instance of rule \( R \). Observe that the head \( t \) of rule \( R \) unifies with anchor \( a \) (otherwise there would be \( z \in S \) such that \( z\omega = a'\omega \)). Consequently, each application of rule \( R \) yields a term that unifies with the anchor \( a \), which can therefore only be used as anchor for another application of \( R \). This implies that there is path in \( \Pi \) from \( z \) to \( t_i' \) consisting of (zero or more) applications of rule \( R \), the last of which yields a term that unifies with \( t_i' \). This leads to a contradiction with respect to the assumed property that anchor \( a \) does not unify with any of the premises \( t_1, \ldots, t_n \). It must then be the case that \( S'\omega \vdash t_i'\omega \).

Finally, by monotonicity of consequence relations (property 2 of Tarski’s) we have \( S'\omega \cup \{t'\omega\} \vdash t_1\omega, \ldots, S'\omega \cup \{t'\omega\} \vdash t_n\omega \). Then by applying \( R \) we obtain that \( S'\omega \cup \{t'\omega\} \vdash t\omega = u\omega \), thus completing the proof.
Lemma 7. Let $T$ be an AL theory, $\vdash$ the ground deduction relation induced by $T$, $S$ a set of terms, $u$ a term, and $\omega$ a substitution such that $S\omega \vdash u\omega$. If there exists a normal proof of $S\omega \vdash u\omega$ whose last rule is $t \leftarrow t_1, \ldots, t_n$, $a \in T$, and there exists $z \in S$ such that $z\omega = a\omega$, then $S'\omega \vdash t_1\omega, \ldots, S'\omega \vdash t_n\omega$, with $S' = S \setminus \{z\}$.

Proof. Let $\Pi$ be a normal proof of $S\omega \vdash u\omega$ ending with the application of rule $R : t \leftarrow t_1, \ldots, t_n, a \in T$. Let also $z \in S$ such that $z\omega = a\omega$. Assume by contradiction that there is $t_i$ among premises $t_1, \ldots, t_n$ such that $(S\setminus\{z\})\omega \not\models t_i\omega$. Then for any proof $\Pi'$ of $S\omega \vdash t_i\omega$ there is a leaf of $\Pi$ labeled with an application of the axiom rule that equals $z$ and some premise $p$ of a rule $R$ in $T$, under substitution $\omega$. If $p$ is a premise of a rule in $T \setminus \{R\}$ or one of the premises $t_1, \ldots, t_n$ of rule $R$, then $p\omega = z\omega = a\omega$, which contradicts the hypothesis that $a$ does not unify with any other premise in $T$ (cf. def. 2). If $p = a'$ for an instance $t' \leftarrow t'_1, \ldots, t'_n, a'$ of rule $R$, since $\text{var}(t') \subseteq \text{var}(a')$ then $t'\omega = t\omega$, thus contradicting the normality assumption of proof $\Pi$.

\qed

5.5 Solving negative policy constraints

We explain here the procedure $\text{solveNeg}$ to solve the set $N = \text{neg}(cs(\iota, G))$ of negative constraints in $cs(\iota, G)$. If there exists no solution of the constraint set $N$, then $\text{solveNeg}$ returns $\bot$. If a solution of $N$ exists, then $\text{solveNeg}$ returns $\top$ and outputs a provision formula $\Psi$, which intuitively represents a “class” of solutions of $N$; we will formally introduce provision formulas later in the section. Before describing the procedure $\text{solveNeg}$ formally, we explain our approach to solving negative constraints with some simple examples. Recall that here we are limiting our attention to CAP specifications where (1) no negative update occurs, and (2) for all guarded send events $g_3 g_3 \triangleright \text{snd}(m)$ only one of $g_3$ and $g_3$ is non-empty, and it is a singleton set.

Consider a process $\pi$ of a CAP specification with empty policy theory and with a thread that contains the guarded send event

$$\{\} \{g(X,Y)\} \triangleright \text{snd}(m),$$

where $X \in V^3$ is a positively originated variable (see Section 5.1), and $Y \in V \setminus V^3$ is a free variable; that is, $X$ originates in a prior receive event or in a prior positive guard, and $Y$ originates in the negative guard shown above (and, by origination property, it only appears in the negative guard; cf. Section 2.1). Observe, in particular, that while $X$ is bound by another event in the CAP specification, $Y$ is universally quantified in the negative guard (see Section 2.2); intuitively, the query reads as: “given $X$, does there exist no value of $Y$ such that $g(X,Y)$ holds?”.

Let $z$ be the configuration in which the negative guard above is to be evaluated, and suppose that $T = \{g(a,b)\}$ is the extensional knowledge of $\pi$ at configuration $z$. If the value of $X$ at configuration $z$ is $a$, then the guard can not evaluate
to true because there exists a value of \( Y \) (i.e. \( b \)) such that \( T \vdash_\pi g(a, Y) \). On the other hand, if the value of \( X \) is set to any value different from \( a \) (e.g. \( c \)), then the negative guard evaluates to true, because there would not exist any value of \( Y \) such that \( T \vdash_\pi g(c, Y) \).

Here lies one of the main intuitions in our approach to solving negative constraints: we can constrain the assignment of positively originated variables, so that queries in negative guards can not be inferred for any value of their free variables. In the case of the example above, we can demand that \( X \neq a \).

Of course, it is not always possible to constrain positively originated variables so that a negative guard is satisfied; however, these are the cases in which the negative guard is unsatisfiable. For instance, consider an alternative policy theory for process \( \pi \), containing the infon rule \( g(A, A) \leftarrow \); that is, facts of the form \( g(A, A) \) are always considered true in the knowledge of \( \pi \). In this case the negative guard of the example can not be satisfied, as there always exists a value of \( Y \), for any value of \( X \), for which \( T \vdash_\pi g(X, Y) \); consistently, the value of \( X \) can not be constrained in a way such that \( T \not\vdash_\pi g(X, Y) \) for all values of \( Y \).

We now turn our attention to the analysis of the negative constraints. In particular, we give an intuition of how we determine what constraints on the positively originated variables should be enforced, and how we enforce them.

Recall that, for a guarded send event \( \{ \} g \rightarrow \text{snd}(m) \) executed by a process \( \pi \), the evaluation of the negative guard \( \{ \} g \) is reflected in the constraint system by a negative constraint \( T \not\vdash_\pi u \), with \( T \) being the extensional knowledge of \( \pi \) at the configuration in which the guard is to be evaluated; cf. Section 5.2. Let \( c = S \models a \) be a positive constraint. We define the complement \( \neg c \) of \( c \) as the negative constraint \( S \not\models a \). Similarly, the complement \( \neg c \) of a negative constraint \( c = S \not\models a \) is the positive constraint \( S \models a \).

Fix a negative constraint \( n = T \not\vdash u \). By the semantics of negative constraints (see Section 5.1), \( n \) is satisfiable if

\[
\exists \tau : \mathcal{V}^3 \rightarrow \mathcal{T}_{\Sigma(\mathcal{V})}, \quad \forall \rho : \mathcal{V} \setminus \mathcal{V}^3 \rightarrow \mathcal{T}_{\Sigma(\mathcal{V})} \cdot T\tau\rho \not\models u\tau\rho,
\]

that is, if there exists an instantiation of the positively originated variables under which \( u \) can not be inferred from \( T \) for any instantiation of the free variables. \(^2\)

Consider a solution \( \sigma : \mathcal{V} \rightarrow \mathcal{T}_{\Sigma(\mathcal{V})} \) of the complement constraint \( \neg n \). The substitution \( \sigma \) can be “partitioned” into two substitutions \( \tau : \mathcal{V}^3 \rightarrow \mathcal{T}_{\Sigma(\mathcal{V})} \) and \( \rho : \mathcal{V} \setminus \mathcal{V}^3 \rightarrow \mathcal{T}_{\Sigma(\mathcal{V})} \) such that \( \tau\rho = \sigma \); trivially, \( \tau = \{ X \mapsto X\sigma \mid X \in \text{dom}(\sigma) \cap \mathcal{V}^3 \} \) and \( \rho = \{ X \mapsto X\sigma \mid X \in \text{dom}(\sigma) \setminus \mathcal{V}^3 \} \). Observe that the substitution \( \tau \) alone “violates” the negative constraint \( n \), because there exists an instantiation of the free variables \( \rho \) for which \( T\tau\rho \not\models u\tau\rho \). Assume that \( \sigma \) is the only solution of \( \neg n \); then, \( \tau \) is the only “violation” of \( n \). To ensure the satisfiability of the negative constraint \( n \)

\(^2\)Remark that, in order to ease presentation, we are considering CAP specifications in which negative guards contain at most one term. For the precise semantics of negative constraints generated for unrestricted CAP specifications see Section 5.1.
we then require that positively originated variables are never instantiated according to \( \tau \). We express these requirements by means of provisions.

Intuitively, provisions are meant to capture the satisfiability of disunification problems: for example, a provision \( t_1 \not\approx t_2 \) denotes the problem of whether there exists an instantiation \( \tau \) of the positively originated variables for which there exists no instantiation \( \rho \) of the free variables such that \( t_1 \tau \rho = t_2 \tau \rho \). Thus, for each solution of \( \neg n \) a set of provisions is generated, whose satisfiability entails satisfiability of \( n \). In particular, we employ procedure \textit{reduceProv} (see Algorithm 4) to reduce sets of provisions into “simpler” sets.

Finally, if the analysis of the negative constraint \( n \) is successful (i.e. \( n \) is satisfiable), the provisions computed are returned to the caller, i.e. the procedure \textit{solve} (see Chapter 5). Procedure \textit{solve} stores these provisions, and at each new instantiation of the positively originated variables it checks their satisfiability, calling procedure \textit{checkProv} (see Algorithm 5). If the provisions are satisfiable, \textit{solve} continues its computation; if the provisions are no longer satisfiable, \textit{solve} terminates unsuccessfully.

Below, we give a formal description of procedure \textit{solveNeg}, described in pseudo-code in Algorithm 3.

\textbf{Algorithm 3 Procedure solveNeg}

\textbf{REQUIRES:} \( N = \{n_1, \cdots, n_k\} \)

\( \Psi := \{\} \)

\textbf{for all} \( n \in N \) \textbf{do}

\textbf{while} \( \text{there exists } \sigma = \{X_1 \mapsto t_1, \cdots, X_q \mapsto t_q\} \) \text{ returned by } ps(\neg n) \text{ do}

\( \psi := [X_1 \not\approx t_1 \lor \cdots \lor X_q \not\approx t_q] \)

\textit{reduceProv}(\( \psi \))

\textbf{if} \( \psi = \[] \) \textbf{then}

\textbf{return} \( (F, \emptyset) \)

\textbf{else}

\( \Psi := \Psi \cup \{\psi\} \)

\textbf{return} \( (T, \Psi) \)

First, we define free unification of terms. Terms \( t_1 \) and \( t_2 \) are freely unifiable, denoted \( t_1 \sim t_2 \), if there exists an instantiation \( \rho \) of the free variables such that \( t_1 \rho = t_2 \rho \); otherwise, \( t_1 \not\sim t_2 \). A provision is a pair of terms of the form \( t_1 \not\approx t_2 \), that denotes the problem of “disunifying” the terms \( t_1 \) and \( t_2 \): \( t_1 \not\approx t_2 \) is solved if there exists no substitution \( \sigma : \mathcal{V} \rightarrow \mathcal{T}(\mathcal{V}) \) such that \( t_1 \sigma = t_2 \sigma \). A provision is satisfiable if there exists a substitution \( \tau \) of the positively originated variables under which it is solved.

A provision clause is a set of provisions, that we represent with the notation \( \delta = [x_1 \not\approx y_1 \lor \cdots \lor x_n \not\approx y_n] \); due to its semantics, it can be seen as a disjunction of provisions. A provision clause is solved when at least one of its disjuncts is solved, and it is satisfiable as long as at least one of its disjuncts is satisfiable.

A provision formula is a set of provision clauses, represented with the notation
\[ \Delta = \{ \delta_1 \wedge \cdots \wedge \delta_k \} \]; due to its semantics, it can be seen as a conjunction of provision clauses. A substitution \( \tau : \mathcal{V}^\exists \rightarrow \mathcal{T}_{\Sigma(V)} \) is a solution of the provision formula \( \Delta = \{ \delta_1 \wedge \cdots \wedge \delta_k \} \) if \( \tau \) solves all the conjuncts \( \delta_1, \cdots, \delta_k \) simultaneously; \( \Delta \) is satisfiable if there exists \( \tau \) that solves it. If a substitution \( \sigma \) satisfies \( \Delta \), we say that \( \sigma \) instantiates \( \Delta \).

The procedure \( \text{solveNeg} \) takes a set \( N \) of negative policy constraints as input, and returns a pair \( (V, \Psi) \) where \( V \) is a truth value, indicating whether or not the constraints in \( N \) can be satisfied, and a provision formula \( \Psi \) whose solutions are solutions of \( N \).

The procedure starts by initializing the provision formula \( \Psi \) to the empty provision formula. Then, for each constraint in the set \( N \), \( \text{solveNeg} \) invokes the procedure \( \text{ps} \) (see Section 5.4) to retrieve the solutions of the complement constraint \( \neg n \). Procedure \( \text{ps} \) returns a set of substitutions whose refinements are all the solutions of the constraint \( \neg n \) (cf. Sections 5.4.2 and 5.4.3).

Recall how procedure \( \text{ps} \) works: given a positive constraint \( S \models a \) in input, \( \text{ps} \) tries to construct a proof "skeleton" \( \Pi \) of \( S_{\sigma} \models a_{\sigma} \), and returns the substitution \( \sigma \) is a "minimal" set of variable assignments that are necessary for \( \Pi \) to hold. For each substitution \( \sigma = \{ X_1 \mapsto t_1, \cdots, X_q \mapsto t_q \} \) returned by \( \text{ps} \) for \( \neg n \), \( \text{solveNeg} \) generates the provision clause \( \psi = [X_1 \not\approx t_1 \lor \cdots \lor X_q \not\approx t_q] \). The provision clause \( \psi \) captures the intuition that if for any variable assignment \( X \mapsto t \) in \( \sigma \) the variable \( X \) is instantiated with a value \( t' \) different from \( t \) (more precisely, that does not unify with \( t \)), then the corresponding proof skeleton \( \Pi \) can not hold; therefore, an instantiation of variables that disagrees with all the (finitely many) solutions returned by \( \text{ps} \) on at least one variable assignment for the complement constraint \( \neg n \) would make \( \neg n \) unsatisfiable, and thus satisfy \( n \).

Next, \( \text{solveNeg} \) invokes the procedure \( \text{reduceProv} \) to reduce \( \psi \), shown in pseudocode in Algorithm 4 below.

**Algorithm 4** Procedure \( \text{reduceProv} \)

**REQUIRES:** \( \delta = [t_1 \not\approx r_1 \lor \cdots \lor t_k \not\approx r_k] \)

while there is \( X \not\approx t \) in \( c \) such that \( X \sim t \) do

\[ \rho := \text{mgu}^\sim(X, t) \]

\[ \delta := (\delta \setminus \{ X \not\approx t \}) \rho \]

Intuitively, \( \text{reduceProv} \) identifies and removes the provisions \( X \not\approx t \) in \( \delta \) that are trivially unsatisfiable: that is, for which there exists \( \rho : \mathcal{V} \setminus \mathcal{V}^3 \rightarrow \mathcal{T}_{\Sigma(V)} \) such that \( X_\rho = t_\rho \). When a provision \( X \not\approx t \) is removed from \( \delta \), the substitution \( \text{mgu}^\sim(X, t) \) is propagated to the remainder of the clause. Here, \( \text{mgu}^\sim(t_1, t_2) \), defined only for freely unifiable pairs of terms \( t_1 \) and \( t_2 \), is the most general unifier \( \rho \) of \( t_1 \) and \( t_2 \) such that \( \text{dom}(\rho) \subseteq \mathcal{V} \setminus \mathcal{V}^3 \); that is, \( \rho \) instantiates only free variables.

The procedure \( \text{reduceProv} \) is obviously terminating; Theorem 8 below shows that \( \text{reduceProv} \) is also sound and complete.
Theorem 8. Let $\delta$ be a provision clause, and $\delta'$ be the reduced provision clause obtained after application of reduceProv. Then a substitution $\sigma : \mathcal{V}^3 \to T_{\Sigma(\mathcal{V})}$ is a solution of $\delta'$ if and only if $\sigma$ is a solution of $\delta$.

Proof. We split the proof in two directions:

$\Rightarrow$ Let $\sigma$ be a solution of $\delta$. Then there exists $t \not\equiv r$ in $\delta$ such that $t\sigma \not\sim r\sigma$. Since reduceProv removes $t' \not\equiv r'$ such that $t' \sim r'$, the provision $t\rho \not\equiv r\rho$ is in $\delta'$, with $\rho = \text{mgu} \sim (t', r')$. Since $\text{dom}(\rho) \subseteq \mathcal{V} \setminus \mathcal{V}^3$, then $\text{dom}(\rho) \cap \text{dom}(\sigma) = \emptyset$. It is immediate then that $t\rho \sigma \not\sim r\rho \sigma$.

$\Leftarrow$ Let $\sigma$ be a solution of $\delta'$. Then there exists $t\rho \not\equiv r\rho$ in $\delta'$ such that $t\rho \sigma \not\sim r\rho \sigma$. Obviously the provision $t \not\equiv r$ is in $\delta$. Since $\rho = \text{mgu} \sim (t', r')$ for some $t' \not\equiv r'$ in $\delta$, then $\text{dom}(\rho) \subseteq \mathcal{V} \setminus \mathcal{V}^3$, and $\text{dom}(\rho) \cap \text{dom}(\sigma) = \emptyset$. It is immediate then that $t\rho \sigma \not\sim r\rho \sigma$.

The following corollary is now immediate.

Corollary 1. Let $\delta$ be a provision clause. Procedure reduceProv reduces $\delta$ to the empty clause $[]$ if and only if $\delta$ is unsatisfiable.

After procedure reduceProv has reduced the provision clause $\psi$, solveNeg checks whether $\psi$ has been reduced to the empty provision clause (denoted $[]$). If it has, then solveNeg returns $\mathsf{F}$ (along with an empty provision formula), to indicate that the negative constraints are not satisfiable; cf. Corollary 1. Otherwise, solveNeg adds $\psi$ to the provision formula $\Psi$ and proceeds with the analysis of the solutions of the complement constraint $\neg n$. When all solutions of $\neg n$ have been analyzed, solveNeg picks another constraint in $N$ and analyzes its solutions, until all constraints of $N$ have been analyzed. If for no solution of any constraint in $N$ the corresponding provision clause generated by solveNeg reduces to $[]$, then solveNeg returns the truth value $\mathsf{T}$ and the provision formula $\Psi$.

After solveNeg has successfully computed and returned a provision formula $\Psi$ characterizing all the instantiations of positively originated variables that satisfy the negative constraints in $N$, procedure solve must enforce that $\Psi$ remains satisfiable at each further instantiation of the positively originated variables. To do so, solve invokes the procedure checkProv, discussed in Section 5.6.

The following theorems state the soundness and completeness of solveNeg; that is, that the solutions of a negative policy constraint $n$ are all and only the solutions of the provision formula returned by solveNeg for the input $n$.

Theorem 9. Let $n$ be a negative policy constraint and $\Psi$ be the provision formula returned by procedure solveNeg for the input $n$. If substitution $\sigma : \mathcal{V} \to T_{\Sigma(\emptyset)}$ is a solution of $\Psi$, then $\sigma$ is a solution of $n$.
Proof. Let $\Psi = \{\delta_1 \land \cdots \land \delta_k\}$ and $\sigma : \mathcal{V} \to T_{\Sigma(0)}$ be a substitution that satisfies $\Psi$. Without loss of generality, assume all clauses of $\Psi$ have been reduced by procedure $\text{reduceProv}$. By Corollary 1 no clause $\delta$ in $\Psi$ is empty. Then there exists a provision $X_i \neq t_i$ in each $\delta_i$, for $i \in \{1, \cdots, k\}$, such that $X_i \sigma \neq t_i \sigma$.

Assume, by contradiction, that $\sigma$ is a solution of $\neg n$; then, by completeness of procedure $ps$ (cf. Theorem 7), $ps$ returns, for the input $\neg n$, a substitution $\rho = \{Y_1 \mapsto r_1, \cdots, Y_q \mapsto r_q\}$ such that $\rho \subseteq \sigma$; hence, $Y_i \sigma = t_i \sigma$ for all $\ell \in \{1, \cdots, q\}$. However, by construction of $\Psi$, $\delta_j = [Y_1 \neq r_1 \lor \cdots \lor Y_q \neq r_q]$ for some $j \in \{1, \cdots, k\}$, and therefore $Y_j \sigma \neq r_j \sigma$ for some $j \in \{1, \cdots, q\}$, leading to a contradiction.

\[\square\]

Theorem 10. Let $n$ be a negative policy constraint and $\Psi$ be the provision formula returned by procedure $\text{solveNeg}$ for the input $n$. If substitution $\sigma : \mathcal{V} \to T_{\Sigma(0)}$ is a solution of $n$, then $\sigma$ is a solution of $\Psi$.

Proof. Let $\sigma : \mathcal{V} \to T_{\Sigma(0)}$ be a substitution that solves $n$, and $\Psi = \{\delta_1 \land \cdots \land \delta_k\}$ be the provision formula returned by procedure $\text{solveNeg}$ for the input $n$.

Assume, by contradiction, that $\Psi$ is not solved by $\sigma$. Then, for some $i \in \{1, \cdots, k\}$ and $\delta_i = [X_1 \neq t_1 \lor \cdots \lor X_q \neq t_q]$, $X_j \sigma = t_j \sigma$ for all $j \in \{1, \cdots, q\}$. By construction of $\Psi$, one of the solutions returned by $ps$ for the input $\neg n$ is the substitution $\tau = \{X_1 \mapsto t_1, \cdots, X_q \mapsto t_q\}$. By soundness of procedure $ps$ (see Theorem 6), all refinements of $\tau$ are solutions of $\neg n$. Since $\sigma \supseteq \tau$ then $\sigma$ is a solution of $\neg n$, which leads to a contradiction.

\[\square\]

Remark 4. In [25], Corin et al. prove that a negative constraint $T \not\vdash u$ over the Dolev-Yao theory is unsatisfiable only if $T \sigma \vdash u \sigma$ for a substitution $\sigma$ that replaces each variable appearing in $T \cup \{u\}$ with fresh distinct constants. This idea has been inspired by the work in [44]. Remark that negative constraints in our work contain free variables (interpreted as universally quantified), while negative constraints in [25] do not. However, when considering negative constraints containing no free variable, there are similarities between our technique for solving negative policy constraints and theirs. Let $n = T_p \not\vdash u_p$, be a negative constraint with $\text{var}(n) \subseteq \mathcal{V}^3$. We consider $n$ unsatisfiable only if the empty substitution is a (symbolic) solution for $T_p \not\vdash u_p$ (implication of Theorem 8); that is, only if it is possible to prove $u_p$ from $T_p$ when considering the variables in $T_p$ and $u_p$ as (distinct) constants. The difference between the two techniques, besides applying to different theories, lies in that the one in [25] requires a ground deduction procedure for the Dolev-Yao theory, whereas we use the AL proof search procedure (see Section 5.4) to generate a complete set of solutions for $T_p \not\vdash u_p$, and ultimately provide a provision formula that characterizes all instantiations of variables that satisfy the negative constraint.

Another approach to handling negative constraints is found in [14, 51], where
solutions of negative constraints are carried over in the symbolic analysis as a collection of disequalities; more specifically, as a conjunction of disjunctions of disequalities. This approach is very similar to provisions: disequalities are used to rule out instantiations of variables that satisfy the complements of the negative constraints. The difference between disequalities and provision formulas lies in that the latter account for universally quantified variables that can appear in negative guards in \text{CAP}s, while no free variables appear in negative queries in [14, 51]. Observe also that negative queries in [14,51] only refer to extensional predicates; that is, facts explicitly represented in the state. Our technique, in contrast, handles both extensional and intensional facts; i.e., inferred through deduction.

5.6 Enforcing the provisions

At the end of each reduction step, procedure \textit{solve} must ensure that the provision formula $\Psi$ characterizing all the instantiations of positively originated variables that satisfy the negative constraints in $cs(\iota,G)$ remains satisfiable. To do so, \textit{solve} invokes the procedure \textit{checkProv}, shown in Algorithm 5 below.

\begin{algorithm}
\textbf{Algorithm 5} Procedure \textit{checkProv} \\
\textbf{REQUIRES:} $\Delta = \{\delta_1 \land \cdots \land \delta_n\}$ \\
\textbf{for all} $\delta$ in $\Delta$ \textbf{do} \\
reduceProv($\delta$) \\
\textbf{if} $\delta = \emptyset$ \textbf{then} \\
\textbf{return} $\text{F}$ \\
\textbf{return} $\text{T}$
\end{algorithm}

Procedure \textit{checkProv} takes a provision formula $\Delta$ as input, and returns a truth value denoting whether or not $\Delta$ is satisfiable. Each provision clause $\delta$ in $\Delta$ is reduced using procedure \textit{reduceProv}; if $\delta$ reduces to the empty clause, then $\delta$ is unsatisfiable (cf. Corollary 1) and \textit{checkProv} returns F indicating that there exists no instantiation of positively originated variables that satisfies $\Delta$. If no clause in $\Delta$ reduces to the empty clause, then \textit{checkProv} returns T.

The termination of procedure \textit{checkProv} is immediate. The following theorem states that if a constraint system $CS$ is reduced to trivial form (see Section 5.1) and the provision formula $\Psi$ stemming from the reduction of $CS$ is satisfiable, then there exists a solution that simultaneously solves $CS$ and satisfies $\Delta$. The theorem hinges on the assumption, common in security protocol models, that the knowledge of a Dolev-Yao attacker includes an infinite set $C_A$ of fresh constant terms; i.e., the terms in $C_A$ do not appear in the set of function constructors $\Sigma$. This assumption reflects the intuition that an attacker can always send randomly generated bitstrings that do not match any particular cryptographic pattern. We refer to this assumption as \textit{AC}. 

75
Theorem 11. Let CS = \{T_1 \models^A X_1, \cdots, T_n \models^A X_n\} be a constraint system in trivial form, and \(\Delta\) be a provision formula such that \(\text{var}(\Delta) \cap \forall^\exists \subseteq \{X_1, \cdots, X_n\}\); that is, the positively originated variables in \(\Delta\) appear in CS. Assuming AC, if \(\Delta\) is satisfiable then there exists a solution \(\sigma\) of \(\Delta\) such that \(\sigma\) is a solution of CS.

Proof. The provision formula \(\Delta\) is of the form \(\Delta = \{\delta_1 \land \cdots \land \delta_k\}\). We assume \(\Delta\) has been reduced by procedure reduceProv; by Theorem 8, all solutions of \(\Delta\) are preserved in the reduction. If \(\Delta\) is satisfiable, then by Corollary 1 there exists at least a provision \(p_i = (t_1 \neq t_2) \in \delta_i\), for all \(i \in \{1, \cdots, k\}\). A solution \(\sigma\) of \(\Delta\) is a solution of the formula \(\Psi = Y_1 \neq r_1 \land \cdots \land Y_k \neq r_k\), for some \(Y_i \neq r_i \in \delta_i\) and all \(i \in \{1, \cdots, k\}\). If there exists a solution of \(\Psi\), then also a substitution \(\sigma\) that instantiates all the variables in \(\Psi\) with fresh constants is a solution of \(\Psi\); cf. [44]. Recall that the positively originated variables in \(\Delta\) (and thus in \(\Psi\)) originated in CS. Due to assumption AC, the attacker knowledge contains a countably infinite set of fresh constants \(C_A\); therefore, an attacker can construct a substitution \(\sigma\) as described above.

\(\square\)

5.7 Symbolic treatment of retraction

In this section we present a technique to reason symbolically about retracted facts in DC specifications, and describe how the technique integrates with the procedure reach (Algorithm 1) and its subroutines. We start with an example to show some of the subtleties intrinsic to the retraction of facts.

Example 3. Consider a process \(\pi = (\{\text{th}\}, \emptyset, \emptyset)\), where thread \(\text{th}\) is given as:

1. \(\text{rcv}(X) \quad \triangleright \quad \{f(X)\}\{\cdot\}\)
2. \(\text{rcv}(Y) \quad \triangleright \quad \{\{g(Y)\}\{f(Y)\}\})\)
3. \(\{f(Z)\}\{\cdot\} \quad \triangleright \quad \text{snd}(t_1)\)
4. \(\{g(Z)\}\{\cdot\} \quad \triangleright \quad \text{snd}(t_2)\)

The result of the evaluation of the query \(f(Z)\), at line (3), depends on the concrete values that the variables \(X\) and \(Y\) assume:

- If \(Y = X\) then the extensional knowledge \(\Omega\) of \(\pi\), at line (3), will be \(\Omega = \{f(X), g(Y)\} \setminus \{f(X)\} = \{g(Y)\}\); thus \(\Omega \vdash^\emptyset f(Z)\).

- If \(Y \neq X\) then the extensional knowledge \(\Omega\) of \(\pi\), at line (3), will be \(\Omega = \{f(X), g(Y)\} \setminus \{f(Y)\} = \{f(X), g(Y)\}\); in this case \(\Omega \sigma \vdash^\emptyset f(Z)\sigma\), for \(\sigma = \{X \mapsto Z\}\).

Suppose that the query at line (3) evaluates to true. Then the extensional knowledge of \(\pi\) is \(\Omega = \{f(X), g(Y)\}\), \(X = Z\) and \(X \neq Y\). The evaluation of the query \(g(Z)\), at line (4), also depends on the value of \(Y\):

76
• If $Z \neq Y$, the query evaluates to false since $\{f(X), g(Y)\} \not\models g(Z)$.

• If $Z = Y$, the query evaluates to true; that is, $\{f(X), g(Y)\} \sigma \models g(Z)\sigma$ for $\sigma = \{Y \mapsto Z\}$. However, such an instantiation of $Y$ cannot be allowed, as it would imply $Y = Z = X$, which would contradict $X \neq Y$.

This shows that the guards at lines (3) and (4) cannot simultaneously evaluate to true in any execution; therefore the thread $th$ can never be executed in its entirety.

(end of example)

We will shortly see that in order to account for retraction, we do not need to model it explicitly; instead, we use provisions (cf. Section 5.5) to reason about how terms appearing in negative updates affect the evaluation of the queries.

Consider any given interleaving of the events of a CAP specification. The interleaving fixes a linear order on the events; we number the events accordingly and refer to their number as their “time” of occurrence. Let $a$ be a term introduced in the extensional knowledge of a process at time $t$, and $r$ be a term appearing in a negative update of the process at time $t'$ (with $t < t'$); intuitively, $a$ depends on $r$, in that $r$ can retract $a$ under a substitution that unifies them (if any exists).

Now let $\Pi$ be a proof of a query evaluated at time $t''$, with $t < t' < t''$. If $a$ appears in $\Pi$ as one of the proof leaves, then the validity of $\Pi$ hinges upon that $a$ has not been retracted before time $t''$. Knowing that $a$ depends on $r$, we can assert that $\Pi$ holds provided that $a \neq r$. We draw from this observation our approach to handling retraction of facts symbolically: we associate to each term $a$ in the extensional knowledge (at a given configuration) a provision formula (see Section 5.5) that reflects the conditions under which $a$ can be used to infer queries.

We illustrate this intuition with a simple example.

**Example 4.** Consider a process $\pi = (\{th\}, \{f(a,c)\}, \emptyset)$, where thread $th$ is given as:

1. $\quad \text{rcv}(m(X,Y,W)) \quad \triangleright \quad \{f(a,X), g(Y,W)\}\{\}^\land$
2. $\quad \text{rcv}(t_1) \quad \triangleright \quad \{\}\{f(a,Z), g(Z,a)\}^\land$
3. $\quad \{g(b,a)\}\{\}^\land \quad \triangleright \quad \text{snd}(t_2)$

Here, $X,Y,W \in \mathcal{V}^3$ are positively originated variables and $Z \in \mathcal{V} \setminus \mathcal{V}^3$ is a free variable. At line (1), the thread receives a message of the form $m(X,Y,W)$, and adds the terms $f(a,X)$ and $g(Y,W)$ to the extensional knowledge of $\pi$; the extensional knowledge of $\pi$ becomes then $\Omega = \{f(a,c), f(a,X), g(Y,W)\}$.

At line (2), the thread receives a term $t_1$ and subsequently retracts from $\Omega$ all the pairs of facts $f(a,Z), g(Z,a)$; recall that free variables in negative updates are universally quantified. In this case, the pairs of terms that might match the pair $f(a,Z), g(Z,a)$ are two: $f(a,c)$ and $g(Y,W)$, and $f(a,X)$ and $g(Y,W)$.

Observe that the term $g(Y,W)$ appears in both pairs, which shows that a term can be retracted under a multiple number of substitutions.

In order to capture the conditions under which a term has not been retracted at a given configuration, we introduce a function $\Pi$ that associates a provision formula to each term $a$ in the extensional knowledge of process $\pi$. The provision formula,
denoted $\Upsilon^\pi(a)$, is meant to constrain the instantiation of positively originated variables so to ensure that $a$ has not been retracted before a given configuration. In this example, for the configuration reached after execution of the event at line (2),

\[
\begin{align*}
\Upsilon^\pi(f(a,c)) &= \{[Y \not= c \lor W \not= a]\}, \\
\Upsilon^\pi(f(a,X)) &= \{[X \not= Y \lor W \not= a]\}, \\
\Upsilon^\pi(g(Y,W)) &= \{[Y \not= c \lor W \not= a] \land [X \not= Y \lor W \not= a]\}.
\end{align*}
\]

We call the function $\Upsilon$ a retraction record function, in that it is used to store provision formulas related to each retraction occurred prior to a given configuration.

At line (3), the term $t_2$ is sent to the network if the query $g(b,a)$ can be inferred from the extensional knowledge of $\pi$. This is possible under the substitution $\sigma = \{Y \mapsto b, W \mapsto a\}$ that unifies the query $g(b,a)$ and the term $g(Y,W)$ in the extensional knowledge; however, to ensure that $g(Y,W)$ has not been retracted before the evaluation of the query, the provision formula $\Upsilon^\pi(g(Y,W))$ must also be enforced. After application of substitution $\sigma$, the provision formula $\Upsilon^\pi(g(Y,W))$ becomes the formula

\[
\Upsilon^\pi(g(Y,W))\sigma = \{[b \not= c \lor a \not= a] \land [X \not= b \lor a \not= a]\}
\]

Let $\Psi = \Upsilon^\pi(g(Y,W))\sigma$. The provision clauses in $\Psi$ can be reduced using procedure $\text{reduceProv}$ (see Section 5.5), thus reducing $\Psi$ to the provision formula $\Psi' = \{[b \not= c] \land [X \not= b]\}$. Because the first clause of $\Psi'$ is trivially satisfied, the satisfaction of $\Psi'$ (and, due to Theorem 8, of $\Psi$) relies entirely on the provision $X \not= b$. Therefore, the positive guard $g(b,a)$ evaluates to true under the application of $\sigma$, and as long as $X$ is instantiated with a value different from $b$.

(end of example)

In the remainder of the chapter, a solution of a constraint $c$ will refer to a pair $(\sigma, \Psi)$ where $\sigma$ is a substitution that satisfies $c$ (see Section 5.1) and $\Psi$ is a provision formula that ensures the soundness of $\sigma$ as a satisfying substitution for $c$. Occasionally, when $\Psi$ is the empty formula (e.g. for positive constraints stemming from a CAP specification where no retraction of facts occurs) we might use the term “solution” to refer to the substitution $\sigma$ that satisfies the constraint; vice versa, when $\sigma = \emptyset$ (e.g. for negative constraints) we might use the term “solution” to refer to the provision formula $\Psi$.

Due to the more complex form of solutions to constraints stemming from retraction adjustments are needed to our technique for evaluation of negative queries. We give an intuition of these adjustments through an example.

Example 5. Let $\pi$ be the process of Example 4, with the exception of thread $th$ now given as follows:

\[
\begin{align*}
(1) & \quad \text{rcv}(m(X,Y,W)) \quad \triangleright \quad \{f(a,X), g(Y,W)\}\{\}\sim \\
(2) & \quad \text{rcv}(t_1) \quad \triangleright \quad \{\} \{f(a,Z), g(Z,a)\}\sim \\
(3) & \quad \{\} \{g(b,a)\}\sim \quad \triangleright \quad \text{snd}(t_2)
\end{align*}
\]
that is, the query $g(b, a)$ at line (3) is negative; it asks whether it is not possible to infer $g(b, a)$ from the extensional knowledge of $\pi$ at configuration $z$.

Consider an alternative representation of substitutions: a substitution $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ can be represented as the formula $X_1 = t_1 \land \cdots \land X_n = t_n$. We recall the technique we use to evaluate negative queries in CAP specifications where no retraction of facts occurs (see Section 5.5). Given a negative query, for each solution $\sigma = X_1 = t_1 \land \cdots \land X_n = t_n$ of the complementary (positive) query we build a provision clause $\psi = [X_1 \not\approx t_1 \lor \cdots \lor X_n \not\approx t_n]$; $\psi$ represents the “negation” of the solution $\sigma$, in that it is satisfied when variables are instantiated differently from $\sigma$.

Consider the negative query $g(b, a)$. The complementary query is solved by the pair $(\sigma, \Psi)$ where $\sigma = \{Y \mapsto b, W \mapsto a\}$ and $\Psi$ (after reduction, see Section 5.5) is the provision formula $\Psi = \{X \not\approx b\}$. The solution $(\sigma, \Psi)$ can be represented as the following formula:

$$Y = b \land W = a \land X \neq b.$$ 

Following the same intuition of Section 5.5, a solution to the negative query is an instantiation of the variables under which the complementary positive query can not be inferred. In this case (in which the complementary query has only one solution), it suffices that $Y \neq b$, $W \neq a$ or $X = b$. To express the requirement that $X = b$ we introduce a new form of provision: $t_1 \approx t_2$, denoting the problem of freely unifying two terms $t_1$ and $t_2$. We give a formal definition later in the section. The provision formula reflecting the negation of the solution $(\sigma, \Psi)$ of the positive query $g(b, a)$ (and thus a solution of the negative query) is the formula

$$Y \neq b \lor W \neq a \lor X \approx b.$$ 

(end of example)

We now give a formal description of how the retraction affects the procedure $\text{reach}$ (i.e. Algorithm 1) and its subroutines. Below, we recall how the procedure $\text{reach}$ works; also, we transcribe the code of Algorithm 1 for reference.

In the remainder of the chapter, we fix a CAP cap in DC and a goal $G$. Procedure $\text{reach}$, for each interleaving $\iota$ of cap, generates a constraint system $cs(\iota, G)$. The constraint system $cs(\iota, G)$ is then given as input to the procedure $\text{solve}$, that non-deterministically reduces $cs(\iota, G)$; since the reduction is non-deterministic, the procedure is invoked repeatedly until a successful reduction (i.e. that solves $cs(\iota, G)$) is performed, or all reduction paths have been explored unsuccessfully.

If a solution for $cs(\iota, G)$ is found, then $\text{solve}$ returns the truth value $T$ indicating that there exists a trace in cap that satisfies goal $G$; as witness, the (non-ground) trace $\iota \sigma$ is returned, together with the provision formula $\Psi$ that constrains the assignment of non-instantiated variables. However, if no solution for $cs(\iota, G)$ is found, $\text{solve}$ tries another interleaving of cap. If for all interleavings $\iota$ of cap
Algorithm 1 Procedure $reach$

REQUIRES: $\text{REACH}(\text{cap}, G)$, $\text{cap} = ((S, \Sigma, V), \pi_1, \cdots, \pi_\ell, A)$

$I := \text{interleavings}\{\pi_1, \cdots, \pi_\ell\}$
for all $i \in I$ do
  repeat
    $(\text{res}, \sigma, \Psi) := \text{solve}(cs(i, G))$
  until $\text{res} = T$ or all reduction paths have been explored
  if $\text{res} = T$ then
    return $(T, \sigma, \Psi)$
  return $(F, \emptyset, \emptyset)$

no solution of the constraint system $cs(i, G)$ is found, then $\text{solve}$ returns the truth value $F$ indicating that there exists no trace in cap that satisfies the goal $G$.

The first modification to algorithm $\text{solve}$, necessary to account for retraction of facts in $\text{CAP}$ specifications, concerns the constraint system generation procedure $cs$. In the construction of the constraint system, $cs$ must account for dependencies between terms in the extensional knowledge of a process $\pi$ and terms appearing in negative updates of threads of $\pi$. This is done by associating a retraction record function $\Upsilon$ (for example, see Example 4) that is meant to store the conditions under which a term in the extensional knowledge of $\pi$ has not been retracted, before a given configuration. Below, we explain these modifications formally.

A match $\mu$ between a term set $\Omega$ and a negative update $u_-$ is a subset of $\Omega$ for which there exists a substitution $\sigma : V \rightarrow \mathcal{T}_{\Sigma(V)}$ such that $\mu\sigma = u_-\sigma$. For instance, in Example 4 there are two matches between the term set $\Omega = \{f(a, c), f(a, X), g(Y, W)\}$ and the negative guard $u_- = \{f(a, Z), g(Z, a)\}$; namely, $\mu_1 = \{f(a, c), g(Y, W)\}$ and $\mu_2 = \{f(a, X), g(Y, W)\}$.

A most general unifier of two sets $A$ and $B$ is a substitution $\sigma$ such that for all substitutions $\tau$ for which $A\tau = B\tau$ there exists a substitution $\rho$ such that $\tau = \sigma\rho$. The safety provision clause of a match $\mu$ between a term set $\Omega$ and a negative update $u_-$ is the provision clause $\delta(\mu, u_-) = \{X_1 \neq t_1 \vee \cdots \vee X_n \neq t_n\}$ obtained by “negating” the most general unifier $\text{mgu}(\mu, u_-) = \{X_1 \mapsto t_1, \cdots, X_n \mapsto t_n\}$ of $\mu$ and $u_-$. Intuitively, $\delta(\mu, u_-)$ reflects the conditions under which the terms of the match $\mu$ are not retracted due to the negative update $u_-$. For a term set $\Omega$ and a negative update $u_-$, the safety provision formula of a term $a \in \Omega$ is the provision formula $\Delta(a, \Omega, u_-) = \{\delta(\mu_1, u_-) \wedge \cdots \wedge \delta(\mu_k, u_-)\}$, where $\mu_1, \cdots, \mu_k$ are all the sets such that, for $i \in \{1, \cdots, k\}$, $a \in \mu_i$ and $\mu_i$ is a match between $\Omega$ and $u_-$. The safety provision formula $\Delta(a, \Omega, u_-)$ captures the constraints on the instantiation of variables required in order not to remove the term $a$ from the extensional knowledge $\Omega$ due to the negative update $u_-$. During construction of the constraint system $cs(i, G)$ (see Section 5.2), we
store the dependencies between terms in the extensional knowledge of processes and negative updates performed. To this end, we associate to each policy constraint \( c \) of \( cs(\iota, G) \) a retraction record function \( \Upsilon \) that given a process \( \pi \) and a term \( a \) returns a provision formula \( \Psi \); \( \Psi \) is the union of all the safety provision formulas of \( a \) corresponding to the negatives updates that could have retracted \( a \) before the configuration at which constraint \( c \) is to be evaluated. We write \( \Upsilon^*(a) \) in place of \( \Upsilon(\pi, a) \).

Below, we give a modified version of the constraint system generation procedure \( cs(\iota, G) \), that accounts also for the construction of the retraction record function \( \Upsilon \) associated to each constraint of the constraint system. The association is denoted by superscripting the term set of the constraint with the function; e.g. we write \( \Omega^\Upsilon \) to denote that the retraction record function \( \Upsilon \) is associated to the constraint \( \Omega \). Below, given an interleaving \( \iota = z\iota' \), with \( z \) being a configuration, \( e \) being an event and \( \iota' \) being an interleaving, we write \( \Omega_{\iota'} \) to denote the extensional knowledge of \( \pi \) at configuration \( z \), with \( \pi \) being a process or the attacker. For simplicity, we assume that for all events of the form \( g_3 \stackrel{\text{snd}(m)}{\rightarrow} \) either \( g_3 = \{a\} \) and \( g_3 = \{\} \) or \( g_3 = \{a\} \); that is, exactly one of the sets \( g_3 \) and \( g_3 \) is non-empty, and it is a singleton set. The extension to unrestricted guards is straightforward, see Section 5.2, Remark 3.

\[
\begin{align*}
 cs(\iota, \Upsilon) &= \begin{cases} 
 0 & \text{if } \iota = \epsilon \text{ (empty interleaving)} \\
 \{\Omega^\Upsilon_x \models^\pi g\} \cup cs(\iota', \Upsilon) & \text{if } \iota = z\iota', e = \{\} \{g\} \models^{\iota \text{ snd}(t)} \text{ and } e \text{ is an event of process } \pi \\
 \{\Omega^\Upsilon_x \models^{y^\pi} g\} \cup cs(\iota', \Upsilon) & \text{if } \iota = z\iota', e = \{\} \{g\} \models^{\iota \text{ snd}(t)} \text{ and } e \text{ is an event of process } \pi \\
 \{\Omega_x \models^\iota t\} \cup cs(\iota', \Upsilon') & \text{if } \iota = z\iota', e = \text{rcv}(t) \models_{u_+u_-}, e \text{ is an event of process } \pi \text{ and } \Upsilon' = \nu(\nu(\Upsilon, \{((\pi, a), \Upsilon^\pi(a) \land \Delta(a, \Omega_{\pi}, u_-))\}), \{((\pi, a), \{\} | a \in u_+)\}) \\
\end{cases}
\end{align*}
\]

The procedure \( cs(\iota, \Upsilon) \), given an interleaving \( \iota \) and a retraction record function \( \Upsilon \), generates a constraint system that models the execution of \( \iota \): the constraints in the constraint system reflect the reception of messages from the network or the evaluation of guards. Policy constraints in \( cs(\iota, \Upsilon) \), as opposed to Section 5.2, are associated with the function \( \Upsilon \).

The procedure \( cs \) is initially fed an “empty” retraction record function \( \Upsilon^0 = \{((\pi, a), \{\})\} \). The function \( \Upsilon^0 \) associates the empty provision formula to any term and any process, reflecting that no retraction has occurred yet, hence the “use” of terms in the extensional knowledge of any process is unconditional. When the first event of the interleaving \( \iota \) is a receive event, the procedure \( cs \) is invoked on the remainder of the interleaving with a new retraction record function

\[
\Upsilon' = \nu(\nu(\Upsilon, \{((\pi, a), \Upsilon^\pi(a) \land \Delta(a, \Omega_{\pi}, u_-))\}), \{((\pi, a), \{\} | a \in u_+)\})
\]

Here, the function \( \nu \) is a retraction record overwrite function that, given two retraction record functions \( \Lambda \) and \( \Lambda' \), overwrites the entries in \( \Lambda \) for \( (\pi, a) \in \text{dom}(\Lambda') \).
with the entries of \( \Lambda' \), while all other entries remain unchanged; that is,
\[
\nu(\Lambda, \Lambda') = \{((\pi, a), \Lambda^\pi(a)) \mid (\pi, a) \notin \text{dom}(\Lambda')\} \cup \Lambda'.
\]

Intuitively, \( \Upsilon' \) modifies the existing provisions associated with terms in the extensional knowledge of a process to reflect the effect of the update \( u_+ u_- \) (as per semantics; cf. Section 2.2): the provisions of a term \( a \) are extended with the safety provision formula \( \Delta(a, \Omega^\pi, u_-) \) (i.e. the necessary conditions for term \( a \) not to be retracted from the extensional knowledge of \( \pi \) due to the negative update \( u_- \)) and the provisions associated to a term that is being (re)introduced to the extensional knowledge due to \( u_+ \) are set to the empty provision formula (i.e. these terms can now be used unconditionally).

**Example 6.** Consider again the process \( \pi \) of Example 4, and let \( \iota = z_0 e_1 z_1 e_2 z_2 e_3 z_3 \) be an interleaving where the events \( e_1, e_2 \) and \( e_3 \) are the events at line (1), (2) and (3) respectively of thread \( th \), that we transcribe below for reference.

\[
\begin{align*}
(1) & \quad \text{rcv}(m(X,Y,W)) ▶ \{f(a,X), g(Y,W)\}\{\}\neg \\
(2) & \quad \text{rcv}(t_1) ▶ \{\}\{f(a,Z), g(Z,a)\}\neg \\
(3) & \quad \{g(b,a)\}\{\}\neg ▶ \text{snd}(t_2)
\end{align*}
\]

We invoke the constraint system generation procedure \( cs \) on the interleaving \( \iota \) and the initial retraction record function \( \Upsilon^0 = \{(\pi, a), \{\}\}\), that is, the function that associates the empty provision formula to any term \( a \) for any process \( \pi \). The constraint system generated is
\[
\begin{align*}
\Omega^\pi_0 & \models^\pi m(X,Y,W) \\
\Omega^\pi_0 & \models^\pi t_1 \\
\{f(a,c), f(a,X), g(Y,W)\} & \models^\pi g(b,a)
\end{align*}
\]
where \( \Omega^\pi_0 \) is the initial extensional knowledge of the attacker, and \( \Upsilon \) is the retraction record function obtained by overwriting the function \( \Upsilon^0 \) after the corresponding constraint for event \( e_2 \) has been generated. The function \( \Upsilon \) associates the following provision formulas to the terms in the extensional knowledge of \( \pi \):
\[
\begin{align*}
\Upsilon^\pi(f(a,c)) &= \{[Y \neq c \lor W \neq a]\} \\
\Upsilon^\pi(f(a,X)) &= \{[X \neq Y \lor W \neq a]\} \\
\Upsilon^\pi(g(Y,W)) &= \{[Y \neq c \lor W \neq a] \land [X \neq Y \lor W \neq a]\}.
\end{align*}
\]

(end of example)

Similarly, we modify the constraint generation procedure \( cs(G) \) to generate the constraints that model the goal \( G \), as shown below.
\[
cs(G, \Upsilon) = \begin{cases} 
\{\Omega^\pi \models^\pi f\} & \text{if } G = \pi : f \text{ for } \pi \neq \mathcal{A} \\
\{\Omega_\mathcal{A} \models^\mathcal{A} m\} & \text{if } G = \mathcal{A} : m
\end{cases}
\]

With respect to \( cs(G) \) as defined in Section 5.2, \( cs(G, \Upsilon) \) associates the retraction record function \( \Upsilon \) the policy constraint generated if \( G = \pi : f \), that is,
if $G$ asks whether fact $f$ can be inferred from the extensional knowledge of a process $\pi$. Finally, the constraint system $cs(\iota, G)$ is given as $cs(\iota, \mathcal{T}^0) \cup cs(G, \mathcal{T})$, where $\mathcal{T}^0 = \{((\pi, a), \{\})\}$ and $\mathcal{T}$ is the “last” retraction record function computed in the construction of $cs(\iota, \mathcal{T}^0)$.

After the generation of the constraint system $cs(\iota, G)$, procedure $solve$ is invoked to solve $cs(\iota, G)$. Below, in Algorithm 2a, we present an adaptation of Algorithm 2.

**Algorithm 2a Procedure $solve$**

**REQUIRES**: constraint sequence $CS$

$\tau = \emptyset$, $\Psi = \{\}$

while $CS$ is not trivial && $CS$ is not maximally reduced do

non-deterministically execute one of the following

- if $att(CS)$ is not trivial then $(CS', \sigma, \Delta) = reduceDY(CS)$;
- if $pos(CS) \neq \emptyset$ then $(CS', \sigma, \Delta) = solvePos(CS)$;
- if $neg(CS) \neq \emptyset$ then $(CS', \sigma, \Delta) = solveNeg(CS)$;

$CS := CS' \sigma$, $\tau := \tau \cup \sigma$, $\Psi := (\Psi \land \Delta)\sigma$

if $CS$ is trivial then

$(V, \rho, \Psi') := checkProv(\Psi)$

if $V = T$ then

$CS := CS \rho$, $\tau := \tau \cup \rho$, $\Psi := \Psi'$

else

return $(F, \emptyset, \emptyset)$

if $CS$ is trivial then

return $(T, \tau, \Psi)$

else

return $(F, \emptyset, \emptyset)$

Procedure $solve$, given the constraint system $CS = cs(\iota, G)$ in input, non-deterministically tries one reduction path for $CS$. If the reduction is successful (i.e., $CS$ is reduced to a trivial constraint system, and the provision formula $\Psi$ constructed during the reduction is satisfied) $solve$ returns the truth value $T$ (indicating that a solution to $CS$ has been found), a substitution $\tau$ that satisfies $CS$ and the provision formula $\Psi$ that constrains the assignment of non-instantiated variables.

First, $CS$ initializes a substitution $\tau$ and a provision formula $\Psi$ to be returned in case of a successful run. Then, until $CS$ is not reduced to a trivial constraint system (cf. Section 5.1), or no further reduction is possible, $solve$ non-deterministically invokes one reduction procedure among $reduceDY$ (which reduces attacker constraints in $att(CS)$, until $att(CS)$ is trivial), $solvePos$ (which reduces positive policy constraints, until $pos(CS) = \emptyset$) and $solveNeg$ (which reduces negative policy constraints, until $neg(CS) = \emptyset$).

Each call to these reduction procedures reduces $CS$ to a “simpler” constraint system $CS'$, and returns an instantiation of the variables $\sigma$ and a provision formula $\Delta$. The provision formula $\Psi$, that stores the provision formulas returned by
previous calls to the reduction procedures, is conjoined with $\Delta$. The substitution $\sigma$ is propagated to the new constraint system $CS'$, the substitution $\tau$ and the provision formula $\Psi$.

After each reduction call, if the constraint $CS$ is in trivial form, $solve$ invokes procedure $checkProv$ to check the satisfiability of the provision formula $\Psi$. In contrast with $checkProv$ as defined in Section 5.6, $checkProv$ now returns a truth value $V$ denoting whether $\Psi$ is satisfiable, an instantiation of the variables $\rho$ and a provision formula $\Psi'$; $\rho$ is computed non-deterministically to satisfy provisions of the form $X \approx t$ in the provision formula $\Psi$, and $\Psi'$ is obtained by removing provision clauses in $\Psi$ that are satisfied by $\rho$. If $\Psi$ is satisfiable, then the substitution $\rho$ is propagated to the constraint system, the provision formula $\Psi$ is replaced by $\Psi'$ and the computation continues; otherwise, the truth value $F$ is returned (along with a dummy substitution and provision formula).

Finally, when no further reduction is possible, if $CS$ has been reduced to a trivial constraint system then the triple $(T, \tau, \Psi)$ is returned; otherwise, $solve$ returns the triple $(F, \emptyset, \emptyset)$. Recall that $solve$ explores only one possible reduction path for the constraint system $CS$, hence the truth value $F$ indicates that a solution for $CS$ has not been found for this reduction path. The procedure $reach$ invokes $solve$ repeatedly on $CS$ in order to exhaustively explore all possible reduction paths.

A few modifications to the subroutines of $solve$ are necessary to accommodate the retraction of facts symbolically:

- The procedure $solvePos$ for positive policy constraints (see Section 5.4) must couple each substitution $\sigma$ that solves the constraints $pos(CS)$ with a provision formula $\Psi$. The formula $\Psi$ reflects the conditions under which the proofs found by $solvePos$ for the constraints in $pos(CS)$ are sound, that is, no fact used by the proofs has been retracted; $\Psi$ is then used to constrain future instantiation of the variables.

- The procedure $solveNeg$ to solve negative policy constraints (see Section 5.5) must be extended to support a richer set of provision formulas that include also provisions of the form $t_1 \approx t_2$. In particular, the provision clause reduction procedure $reduceProv$ is revised.

- In order to ascertain the satisfiability of provision formulas that contain provisions of the form $t_1 \approx t_2$, procedure $checkFormula$ must also be revised.

In the following, we explain formally the modification listed above.

**Procedure $solvePos$.** The procedure $solvePos$ is based on the proof search procedure $ps$. We recall how $ps$ works, as defined in Section 5.4. The input to $ps$ is a finite set of positive constraints over an AL theory $T = T_\rightarrow \sqcup T_\leftarrow$ (cf. Definition 1); typically, a singleton constraint system $C_0 = \{\Omega \models u\}$. Intuitively, $ps$ guesses tentative proof trees for $\Omega \tau \vdash u\tau$, with $\tau$ being a solution of $\Omega \models u$. The constraint system created and further reduced by $ps$ represents the set of (not yet
discharged) leaves for the tentative proof tree. If \( ps \) can discharge all the leaves, then \( ps \) terminates successfully. However, if \( ps \) fails to discharge all the leaves of the tentative proof, then \( ps \) backtracks. There are finitely many guesses that \( ps \) can make (see Theorem 5). If \( ps \) fails to discharge the leaves (i.e. assumptions) for all the tentative proofs, then \( ps \) returns a failure.

We present here a modification of \( ps \) that accounts for retraction of facts in \( \text{CAP} \) specifications. In the following, we assume that \( T \) is the policy theory of a process \( \pi \) in \( \text{CAP} \). The procedure \( ps \) applies non-deterministically one of the constraint reduction rules, shown in Figure 5.3, as long as the constraint system is not empty. The reduction rules reflect the infon clauses in \( T \) and the axiom rule. We write \( C \xrightarrow{\gamma(\tau, \Psi)} C' \) to denote that the constraint system \( C \) is reduced to the constraint system \( C' \) by applying the partial substitution \( \tau : \text{var}(C) \rightarrow T_{ \Sigma(V)} \) and assuming that the provision formula \( \Psi \) is satisfied.

\[
\begin{align*}
(1) & \quad C \cup \{(S \cup \{t\})^T \vdash u\} \xrightarrow{\gamma(\tau, \Psi)} C\tau \quad \text{if } \tau = \text{mgu}(u, t) \text{ and } \Psi = \Upsilon^\tau(t) \tau \\
(2) & \quad C \cup \{(S \cup \{z\})^T \vdash u\} \xrightarrow{\gamma(\tau, \Psi)} (C \cup \{S^T \vdash t_1, \ldots, S^T \vdash t_n, (S \cup \{t\})^T \vdash u\})\tau \quad \text{if } \tau = \text{mgu}(a, z) \text{ for some } t \leftarrow t_1, \ldots, t_n, a \in T_\rightarrow \\
& \quad \Psi = \Upsilon^\tau(z)\tau \text{ and } \Upsilon' = \nu(T, \{(\pi, t), \{\}\})) \\
(3) & \quad C \cup \{S^T \vdash u\} \xrightarrow{\gamma(\tau, \Psi)} (C \cup \{S^T \vdash t_1, \ldots, S^T \vdash t_n\})\tau \quad \text{if } \tau = \text{mgu}(u, t) \text{ for some } t \leftarrow t_1, \ldots, t_n \in T_\rightarrow
\end{align*}
\]

Figure 5.3: Constraint reduction system for \textbf{AL} proof search

Below, we describe the rules of procedure \( ps \):

- Rule (1) discharges one of the leaves of the tentative proof using unification. For a constraint \( (S \cup \{t\})^T \vdash u \in C \), if \( u \) can be unified with a term \( t \) in the left hand side of the constraint, then the constraint is removed, and their most general unifier \( \tau \) is applied to the remaining (if any) constraints in \( C \). Furthermore, the provision formula \( \Upsilon^\tau(t)\tau \) is assumed; \( \Upsilon^\tau(t)\tau \) characterizes the instantiations of variables under which \( t \) has not been removed from the symbolic extensional knowledge of \( \pi \).

- Rule (2) applies a \( T_\rightarrow \) rule \( R : t \leftarrow t_1, \ldots, t_n, a \) to a constraint \( (S \cup \{z\})^T \vdash u \in C \) such that \( z \) unifies with the anchor \( a \) of rule \( R \). The substitution \( \tau = \text{mgu}(a, z) \) is applied to the reduced constraint and the provision formula \( \Upsilon^\tau(z)\tau \) is assumed. In the reduced constraint system, the constraint \( (S \cup \{z\})^T \vdash u \) is replaced by \( n + 1 \) new constraints: the first \( n \) constraints, \( (S^T \vdash t_1, \ldots, S^T \vdash t_n)\tau \), check that the premises of the rule (except for \( a \)) can be proved from \( S \); the last constraint, \( ((S \cup \{t\})^T \vdash u)\tau \), checks whether \( u \) can be proved knowing \( S \) and \( t \), with the provisions \( \Upsilon' = \)
\[ \nu(\Upsilon, \{(\pi, t), \{\}\}) \]. Here, we overwrite the entry for \((\pi, t)\) in \(\Upsilon\) with the empty provision formula, so that when future reductions of the constraint should use \(t\) as a leaf of the tentative proof tree, the empty provision formula would be assumed. That is, the use of \(t\) would be unconditional in that \(t\) is derived from another term; hence \(t\) does not come from the extensional knowledge and can therefore not be retracted.

- Finally, rule (3) applies a rule \(t \leftarrow t_1, \ldots, t_n \in T_{\tau}\) to a constraint \(S_t \models u \in C\). If there exists a most general unifier \(\tau\) of \(u\) and \(t\), then the constraint is replaced by \(n\) constraints \((S_t \models t_1, \ldots, S_t \models t_n)\). The partial substitution \(\tau\) is also applied to the remaining (if any) constraints in \(C\), while the provision formula does not change.

Let \(\sim^*\) be the reflexive and transitive closure of \(\sim\). We write \(C \sim^*_{(\tau, \Psi)} C'\) to denote that there exist constraint systems \(C_1, \ldots, C_k\), substitutions \(\tau_1, \ldots, \tau_k\) and provision formulas \(\Psi_1, \ldots, \Psi_k\) (for \(k \geq 0\)) such that \(C \sim_{(\tau_1, \Psi_1)} C_1 \cdots \sim_{(\tau_k, \Psi_k)} C_k\), \(C_k = C'\), \(\tau = \tau_1 \cdots \tau_k\) and \(\Psi = \Psi_1 \land \cdots \land \Psi_k\). If \(C_0 \sim^*_{(\tau, \Psi)} \emptyset\) then all leaves of a tentative proof tree have been discharged; hence \(ps\) returns the substitution \(\tau\) and the assumed provision formula \(\Psi\). If \(C_0 \sim^*_{(\tau, \Psi)} C'\), with \(C'\) being a non-empty constraint system not further reducible, then the tentative proof at hand fails, and \(ps\) backtracks (i.e. \(ps\) tries different sequences of applications of the rules of Figure 5.3). If no sequence of rule applications leads to an empty constraint system, then \(ps\) reports that the constraint system \(C_0\) has no solutions.

**Example 7.** Recall the process \(\pi = (\{th\}, 0, 0)\) of Example 3, with thread \(th\) defined as follows:

- (1) \(\text{rcv}(X) \triangleright \{f(X)\}\)
- (2) \(\text{rcv}(Y) \triangleright \{g(Y)\}\)
- (3) \(\{f(Z)\}\)
- (4) \(\{g(Z)\}\)

Consider the interleaving \(\iota\) given by the sequence of events of thread \(th\). The constraint system generation procedure constructs \(cs((\iota, \Upsilon^0))\) as follows:

- (c1) \(\Omega_A \models^A X\)
- (c2) \(\Omega_A \models^A Y\)
- (c3) \(\{f(X), g(Y)\}^\Upsilon \models^0 f(Z)\)
- (c4) \(\{f(X), g(Y)\}^\Upsilon \models^0 g(Z)\)

where \(\Omega_A\) is the attacker knowledge, and \(\Upsilon = \{(\pi, f(X)), \{[X \neq Y]\}\}\); that is, \(\Upsilon\) associates to the term \(f(X)\) (in the symbolic extensional knowledge of \(\pi\)) the safety provision formula \{[\(X \neq Y]\]\} that characterizes the instantiations of variables under which \(f(X)\) is not retracted.

For the constraint \((c3)\), \(ps\) finds the solution \((\sigma, \Psi)\), with \(\sigma = \{X \mapsto Z\}\) and \(\Psi = \{[Z \neq Y]\}\). The solution is found by applying rule (1) which unifies \(f(Z)\)
and \( f(X) \); hence the substitution \( \sigma = \{ X \mapsto Z \} \). Furthermore, also the safety provision formula for \( f(X) \) (under substitution \( \sigma \)) must be assumed; therefore also \( \Psi = \{ [Z \not\approx Y] \} \) is returned. The provision \( \Psi \) is stored by \texttt{solve} and must remain satisfiable under future instantiations of variables.

For the constraint \((\epsilon k), ps\) finds the solution \( (\tau, \{\}) \), with \( \tau = \{ Y \mapsto Z \} \). The solution is found by applying rule (1) that unifies \( g(Z) \) with \( g(Y) \); hence the solution \( \tau = \{ Y \mapsto Z \} \). After propagation of \( \tau \), however, the provision formula \( \Psi \) becomes \( \Psi_{\tau} = \{ [Z \not\approx Z] \} \), which is unsatisfiable.

(end of example)

**Procedure solveNeg.** Procedure \texttt{solveNeg} takes a set of negative policy constraints \( N \) as input, and returns a provision formula \( \Psi \) that characterizes the instantiations of variables that are solutions for all the constraints in \( N \); cf. Section 5.5. Let \( n = \Omega \not\models u \) be a negative policy constraint. The intuition behind \texttt{solveNeg} is that the solutions of the complement \( \neg n = \Omega \models u \) of the constraint \( n \) are instantiations of the variables that do not satisfy \( n \); hence, instantiations that differ from all the solutions of \( \neg n \) are solutions of \( n \). While retaining the same intuition, we extend procedure \texttt{solveNeg} to solve negative constraints stemming from \texttt{CAP} specifications where retraction of facts occurs.

We extend the definition of provisions from Section 5.5. A provision is a pair of terms of either of the two following forms:

- \( t_1 \not\approx t_2 \), denoting the problem of “disunifying” the terms \( t_1 \) and \( t_2 \): the provision \( t_1 \not\approx t_2 \) is solved if there exists no substitution \( \sigma : \mathcal{V} \rightarrow T_{\Sigma(V)} \) such that \( t_1\sigma = t_2\sigma \).

- \( t_1 \approx t_2 \), denoting the problem of “freely unifying” the terms \( t_1 \) and \( t_2 \): the provision \( t_1 \approx t_2 \) is solved if \( t_1 \sim t_2 \), i.e. if there exists a substitution \( \rho : \mathcal{V} \setminus \mathcal{V}^2 \rightarrow T_{\Sigma(V)} \) such that \( t_1\rho = t_2\rho \).

A provision is satisfiable iff there exists a substitution \( \tau \) of the positively originated variables under which it is solved. Provision clauses, provision formulas and their solutions are defined as before; see Section 5.5.

We define negation of provisions. The negation of a provision \( t_1 \not\approx t_2 \) is the provision \( t_1 \approx t_2 \); similarly, the negation of a provision \( t_1 \approx t_2 \) is the provision \( t_1 \not\approx t_2 \). For a provision clause of the form \( \psi = [t_1 \not\approx r_1 \lor \cdots \lor t_k \not\approx r_k] \) the negation of \( \psi \) is the provision \( \neg \psi = f(t_1, \cdots, t_k) \not\approx f(r_1, \cdots, r_k) \), with \( f \) being a fresh \( k \)-ary function constructor; due to the semantics of provisions and provision formulas, \( \neg \psi \) is equivalent to the provision formula \( \{ t_1 \approx r_1 \} \land \cdots \land \{ t_k \approx r_k \} \). The negation of a provision formula \( \Psi = \{ \psi_1 \land \cdots \land \psi_q \} \) is the provision clause \( \neg \Psi = [\neg \psi_1 \lor \cdots \lor \neg \psi_q] \).

We give a modified version of procedure \texttt{solveNeg}, described in pseudocode in Algorithm 3a.

The procedure \texttt{solveNeg} takes a set \( N \) of negative policy constraints in input, and returns a a provision formula \( \Psi \) whose solutions are solutions of \( N \). Observe
Algorithm 3a Procedure solveNeg

REQUIRES: \( N = \{n_1, \cdots, n_k\} \)
\( \Psi := \emptyset \)

for all \( n \in N \)
do
while there exists \((\sigma, \Delta)\) returned by \( ps(\neg n) \) do
Let \( \sigma = \{X_1 \rightarrow t_1, \cdots, X_q \rightarrow t_q\} \)
\( \psi := [X_1 \neq t_1 \lor \cdots \lor X_q \neq t_q] \lor \neg \Delta \)
\( \text{reduceProv}(\psi) \)
\( \Psi := \Psi \land \{\psi\} \)
return \( \Psi \)

that \( \Psi \) might be unsatisfiable; procedure solve will ascertain satisfiability of \( \Psi \) by invoking procedure checkProv.

For each constraint \( n \) in the set \( N \), solveNeg invokes the procedure \( ps \) (see paragraph above that explains adaptation of \( ps \) to accommodate retraction) to retrieve the solutions of the complement constraint \( \neg n \). For each solution \((\sigma, \Delta)\) of \( \neg n \), with \( \sigma = \{X_1 \rightarrow t_1, \cdots, X_q \rightarrow t_q\} \), procedure solveNeg generates the provision clause \( \psi = [X_1 \neq t_1 \lor \cdots \lor X_q \neq t_q] \lor \neg \Delta \). The provision clause \( \psi \) represents the “negation” of the solution \((\sigma, \Delta)\), in that \( \psi \) is satisfied for all instantiations \( \sigma' \) of the positively originated variables that disagree with the substitution \( \sigma \) on some element of \( \text{dom}(\sigma) \cap \text{dom}(\sigma') \) or that satisfy one of the provision clauses in \( \neg \Delta \). Observe that since \( \Delta \) is a provision formula that ensures that no term \( a \) used in the proof of \( \neg n \) has been retracted, then satisfying one of the provisions in \( \neg \Delta \) corresponds to retracting \( a \).

Next, solveNeg invokes the procedure reduceProv (cf. Section 5.5) to reduce \( \psi \). We extend reduceProv to reduce provision clauses that contain also provisions of the form \( X \approx t \); the procedure is shown in pseudocode in Algorithm 4a below.

Algorithm 4a Procedure reduceProv

REQUIRES: \( \delta = [t_1 \neq r_1 \lor \cdots \lor t_k \neq r_k \lor s_1 \neq w_1 \lor \cdots \lor s_q \neq w_q] \)
while there is \( X \neq t \) in \( \delta \) such that \( X \sim t \) do
\( \rho := \text{mgu}^\sim(X, t) \)
\( \delta := (\delta \setminus \{X \neq t\})\rho \)
while there is \( X \approx t \) in \( \delta \) such that \( \exists \sigma : V \rightarrow T_{\Sigma(V)} \) s.t. \( X\sigma = t\sigma \) do
\( \delta := \delta \setminus \{X \approx t\} \)

Intuitively, reduceProv identifies and removes the provisions in \( \delta \) that are trivially unsatisfiable; provisions of the form \( X \neq t \) are removed when there exists \( \rho : V \setminus V^3 \rightarrow T_{\Sigma(V)} \) such that \( X\rho = t\rho \), and provisions of the form \( X \approx t \) are removed when there exists no \( \sigma : V \rightarrow T_{\Sigma(V)} \) such that \( X\sigma = t\sigma \). When a provision \( X \neq t \) is removed from \( \delta \), the substitution \( \text{mgu}^\sim(X, t) \) is propagated to the remainder of the clause. Recall that \( \text{mgu}^\sim(t_1, t_2) \) is the most general unifier of two
freely unifiable terms \( t_1 \) and \( t_2 \) that instantiates only free variables; see Section 5.5.

The procedure \( \text{reduceProv} \) is obviously terminating; furthermore, Theorem 12 below shows that \( \text{reduceProv} \) is also sound and complete.

**Theorem 12.** Let \( \delta \) be a provision clause, and \( \delta' \) be the reduced provision clause obtained after application of \( \text{reduceProv} \). Then a substitution \( \sigma : \forall \rightarrow T_{\Sigma(V)} \) is a solution of \( \delta' \) if and only if \( \sigma \) is a solution of \( \delta \).

**Proof.** Assume, without loss of generality, that \( \delta \) is reduced to \( \delta' \) in one step of reduction of the procedure \( \text{reduceProv} \). If a provision of the form \( p = t \not\approx r \) is removed, then the statement of the theorem trivially holds: any solution \( \sigma \) of \( \delta \) satisfies a provision \( p' \) in \( \delta \) such that \( p' \neq p \) (obvious given that \( p \) is unsatisfiable) and \( p' \) appears in \( \delta' \); vice versa, any solution \( \sigma \) of \( \delta' \) satisfies a provision \( p' \) in \( \delta' \) and \( p' \) appears also in \( \delta \).

Assume now that a provision of the form \( p = t \not\approx r \) is removed. We split this case in two directions:

- \( \Rightarrow \) Let \( \sigma \) be a solution of \( \delta \). Then there is a provision \( p \) in \( \delta \) that is satisfied by \( \sigma \). If \( p = t \not\approx r \) then \( t\sigma \not\approx r\sigma \). Since \( \text{reduceProv} \) removes \( t' \not\approx r' \) such that \( t' \sim r' \), the provision \( t\rho \not\approx r\rho \) is in \( \delta' \), with \( \rho = \text{mgu}(t', r') \). Since \( \text{dom}(\rho) \subseteq V \setminus V^3 \), then \( \text{dom}(\rho) \cap \text{dom}(\sigma) = \emptyset \). It is immediate then that \( t\rho\sigma \not\sim r\rho\sigma \). Similarly for the case \( p = t \approx r \).

- \( \Leftarrow \) Let \( \sigma \) be a solution of \( \delta' \). Then there is a provision \( p \) in \( \delta' \) that is satisfied by \( \sigma \). If \( p = t\rho \not\approx r\rho \) in \( \delta' \) then \( t\rho\sigma \not\sim r\rho\sigma \). Obviously the provision \( t \neq r \) is in \( \delta \). Since \( \rho = \text{mgu}^{-1}(t', r') \) for some \( t' \neq r' \) in \( \delta \), then \( \text{dom}(\rho) \subseteq V \setminus V^3 \), and \( \text{dom}(\rho) \cap \text{dom}(\sigma) = \emptyset \). It is immediate then that \( t\sigma \not\sim r\sigma \). Similarly for the case \( p = t \approx r \).

\( \square \)

After procedure \( \text{reduceProv} \) has reduced the provision clause \( \psi \), \( \text{solveNeg} \) adds \( \psi \) to the provision formula \( \Psi \) and proceeds with the analysis of the solutions of the complement constraint \( \neg n \). When all solutions of \( \neg n \) have been analyzed, \( \text{solveNeg} \) picks another constraint in \( N \) and analyzes its solutions, until all constraints of \( N \) have been analyzed. Finally, procedure \( \text{solveNeg} \) returns the provision formula \( \Psi \), characterizing all the instantiations of positively originated variables that satisfy the negative constraints in \( N \).

The following theorems state the soundness and completeness of \( \text{solveNeg} \); that is, that the solutions of a negative policy constraint \( n \) are all and only the solutions of the provision formula returned by \( \text{solveNeg} \) for the input \( n \).

**Theorem 13.** Let \( n \) be a negative policy constraint and \( \Psi \) be the provision formula returned by procedure \( \text{solveNeg} \) for the input \( n \). If substitution \( \sigma : \forall \rightarrow T_{\Sigma(\emptyset)} \) is a solution of \( \Psi \), then \( \tau \) is a solution of \( n \).
Proof. Let $\Psi = \{\delta_1 \land \cdots \land \delta_k\}$ and $\sigma : \mathcal{V} \rightarrow T_{\Sigma(0)}$ be a substitution that satisfies $\Psi$. Without loss of generality, assume all clauses of $\Psi$ have been reduced by procedure $\text{reduceProv}$. As immediate consequence of Theorem 12, no clause $\psi$ in $\Psi$ is empty. Then there exists a provision $p_i$ in each $\psi_i$, for $i \in \{1, \cdots, k\}$, such that $\sigma$ satisfies $p_i$.

Assume, by contradiction, that $\sigma$ is a solution of $\neg n$; then, by completeness of procedure $\text{ps}$ (cf. Theorem 7 and paragraph above), $\text{ps}$ returns, for the input $\neg n$, a substitution $\rho = \{X_1 \mapsto t_1, \cdots, X_q \mapsto r_q\}$ and a provision formula $\Delta = \{\delta_1 \land \cdots \land \delta_m\}$ such that $\rho \subseteq \sigma$ and $\sigma$ instantiates $\Delta$; hence, $Y_i \sigma = t_\ell \sigma$ for all $\ell \in \{1, \cdots, m\}$ and $\sigma$ satisfies $\delta_\ell$ for all $\ell \in \{1, \cdots, m\}$. However, by construction of $\Psi$, $\psi_j = [Y_1 \not\equiv r_1 \lor \cdots \lor Y_q \not\equiv r_q \lor \neg \delta_1 \lor \cdots \lor \neg \delta_m]$ for some $j \in \{1, \cdots, k\}$, and therefore either $Y_j \sigma \not\equiv r_j \sigma$ for some $j \in \{1, \cdots, q\}$ or $\sigma$ does not satisfy $\delta_j$ for some $j \in \{1, \cdots, m\}$, leading to a contradiction.

\[ \square \]

**Theorem 14.** Let $n$ be a negative policy constraint and $\Psi$ be the provision formula returned by procedure $\text{solveNeg}$ for the input $n$. If substitution $\sigma : \mathcal{V} \rightarrow T_{\Sigma(0)}$ is a solution of $n$, then $\tau$ is a solution of $\Psi$.

Proof. Let $\sigma : \mathcal{V} \rightarrow T_{\Sigma(0)}$ be a substitution that solves $n$, and $\Psi = \{\psi_1 \land \cdots \land \psi_k\}$ be the provision formula returned by procedure $\text{solveNeg}$ for the input $n$. Assume, by contradiction, that $\Psi$ is not solved by $\sigma$. Then, for some $i \in \{1, \cdots, k\}$ and $\psi_j = [Y_1 \not\equiv r_1 \lor \cdots \lor Y_q \not\equiv r_q \lor \neg \delta_1 \lor \cdots \lor \neg \delta_m]$, $X_j \sigma = t_j \sigma$ for all $j \in \{1, \cdots, q\}$ and $\sigma$ does not satisfy any $\neg \delta_j$ for $j \in \{1, \cdots, m\}$. By construction of $\Psi$, one of the solutions returned by $\text{ps}$ for the input $\neg n$ is $(\tau, \Delta)$, where $\tau$ is the substitution $\tau = \{X_1 \mapsto t_1, \cdots, X_q \mapsto t_q\}$ and $\Delta$ is the provision formula $\Delta = \{\delta_1 \land \cdots \land \delta_m\}$. By soundness of procedure $\text{ps}$ (see Theorem 6 and paragraph above), all refinements of $\tau$ that satisfy $\Delta$ are solutions of $\neg n$. Since $\sigma \subseteq \tau$ and $\sigma$ instantiates $\Delta$ then $\sigma$ is a solution of $\neg n$, which leads to a contradiction.

\[ \square \]

Theorems 13 and 14 subsume Theorems 9 and 10 of Section 5.5.

After that procedure $\text{solveNeg}$ has returned the provision formula $\Psi$ that characterizes all solutions of the negative constraints in the set of negative constraints $N$, procedure $\text{solve}$ must enforce that $\Psi$ remains satisfiable at each further instantiation of the positively originated variables. To do so, $\text{solve}$ invokes the procedure $\text{checkProv}$, discussed below.

**Procedure checkProv.** After each call to one of the constraint reduction procedures $\text{reduceDY}$, $\text{solvePos}$ or $\text{solveNeg}$, procedure $\text{solve}$ must check that the provision formula $\Psi$ is satisfiable, in order to continue its computation. The check is performed by the procedure $\text{checkProv}$, introduced in Section 5.6. Here we present an adaptation of $\text{checkProv}$ in order to account for provision formulas that may include provisions of the form $X \approx t$. The adaptation is shown in pseudocode in Algorithm 5a below.

90
Algorithm 5a  Procedure checkProv

REQUIRES: $\Delta = \{\delta_1 \land \cdots \land \delta_n\}$

for all $\delta$ in $\Delta$ do
  $\text{reduceProv}(\delta)$
  if $\delta = []$ then
    return $(F, \emptyset, \emptyset)$
  repeat
    pick non-deterministically a provision $p_i$ from $\delta_i$ for all $i \in \{1, \cdots, n\}$
    $\Psi := \{[p_1] \land \cdots \land [p_n]\}$
    while $\exists t \approx r$ in $\Psi \land \sigma = \text{mgu}(t, r)$ do
      $\Psi := (\Psi \setminus \{[t \approx r]\}) \sigma$
      if $(\not\exists t \approx r$ in $\Psi) \land (\not\exists t \not\approx r$ in $\Psi. t \sim r)$ then
        return $(T, \sigma, \Psi)$
    until all combinations of $p_1, \cdots, p_n$ have been tried
  return $(F, \emptyset, \emptyset)$

Intuitively, procedure checkProv tries to construct a “witness” of the satisfiability of a provision formula $\Delta$ in input. First, checkProv reduces all provision clauses $\delta$ in $\Delta$ using procedure reduceProv; if $\delta$ reduces to the empty clause, then $\delta$ is unsatisfiable (immediate implication of Theorem 12) and checkProv returns $F$ (along with a dummy substitution and provision formula) indicating that there exists no instantiation of positively originated variables that satisfies $\Delta$.

If all clauses in $\Delta$ are non-empty, then checkProv non-deterministically picks a provision $p_i$ from each provision clause $\delta_i$ (with $i \in \{1, \cdots, n\}$) and constructs a provision formula $\Psi$ by conjoining the singleton provision clauses $[p_i]$. Obviously, satisfiability of $\Psi$ implies satisfiability of $\Delta$.

Next, checkProv tries to construct an instantiation $\tau$ that satisfies $\Psi$. For all provisions of the form $t \approx r$, procedure checkProv computes (if it exists) the most general unifier $\sigma = \text{mgu}(t, r)$ of the terms $t$ and $r$; then checkProv removes the provision $t \approx r$ and propagates $\sigma$ to the remainder of $\Psi$.

Then checkProv checks whether:

1. There is a provision of the form $t \approx r$ in $\Psi$; if there is one, then it has not been removed from $\Psi$ because unsatisfiable.

2. There is a provision of the form $t \not\approx r$ in $\Psi$ and $t \sim r$; if there is one, then it is unsatisfiable, and so is $\Psi$.

If none of the two cases above occurs, then $\Psi$ is of the form $\Psi = t_1 \not\approx r_1 \land \cdots \land t_k \not\approx r_k$, and by Theorem 11 it is satisfiable; checkProv therefore returns the truth value $T$ indicating that $\Delta$ is satisfiable, along with the substitution $\sigma$ and the provision formula $\Psi$.

However, if $\Psi$ is not satisfiable, checkProv constructs another provision formula from another combination of provisions of the provision clauses in $\Delta$, until
all combination have been tried. If no combination of provisions is satisfiable, then `checkProv` returns the truth value `F` (along with a dummy substitution and provision formula) indicating that there exists no instantiation of the positively originated variables that satisfies `Δ`.

Procedure `checkProv` is obviously terminating. The following theorems state the correctness of procedure `checkProv`; that is, that the solutions of the provision formula `Ψ'` returned by `checkProv` for the provision `Δ` in input are all and only the solutions of `Δ`.

**Theorem 15.** Let `Δ` be a provision formula, and `(σ,Ψ')` be a solution of procedure `checkProv` for the input `Δ`. Then a solution `σ' ⊆ σ` of `Ψ'` is a solution of `Δ`.

**Proof.** Let `Δ = {δ₁ ∧ · · · ∧ δₖ}`, and `Ψ = { [p₁] ∧ · · · ∧ [pₖ] }` be the provision formula constructed by `checkProv`, where `pᵢ` is a provision picked nondeterministically from δᵢ. It is obvious that a solution of `Ψ` is also a solution of `Δ`.

If `Ψ = { [δ₁] ∧ · · · ∧ [δₖ] }`, then `Ψ = mgu((X₁,· · ·,Xₙ),(t₁,· · ·,tₙ))` and `Ψ' = { [Y₁] ∧ · · · ∧ [Yₘ] }`. Let `σ' ⊆ σ` be a solution of `Ψ'`; that is, `Yᵢσ' ≠ rᵢσ` for all `i ∈ {1, · · · , m}`. By idempotence of substitutions, `σσ' = σ'`, hence `σ'` is a solution of `Yᵢ ≠ rᵢ` for all `i ∈ {1, · · · , m}`. Furthermore, since `Xᵢσ = tᵢσ` for all `i ∈ {1, · · · , m}`, then also `Xᵢσ' = tᵢσ'` for all `i ∈ {1, · · · , n}`. Hence `σ'` is a solution of `Ψ`, and consequently of `Δ`.

**Theorem 16.** Let `Δ` be a provision formula. If `σ'` is a solution of `Δ`, then procedure `checkProv` returns a solution `(σ,Ψ')` such that `σ ⊆ σ'` and `σ'` is a solution of `Ψ'`.

**Proof.** Let `Δ = {δ₁ ∧ · · · ∧ δₖ}`. If `σ'` is a solution of `Δ`, then there exists (at least) a provision `pᵢ` in each provision clause `δᵢ`, with `i ∈ {1, · · · , k}`, such that `σ'` solves `pᵢ`. Procedure `checkProv` eventually considers `Ψ = { [p₁] ∧ · · · ∧ [pₖ] }`. Let `Ψ = { [X₁ ≈ t₁] ∧ · · · ∧ [Xₙ ≈ tₙ] ∧ [Y₁ ≠ r₁] ∧ · · · ∧ [Yₘ ≠ rₘ] }`; then `Xᵢσ' = tᵢσ'` for all `i ∈ {1, · · · , n}` and `Yᵢσ' ≠ rᵢσ` for all `i ∈ {1, · · · , m}`. The solution returned by `checkProv` is `(σ,Ψ')`, with `σ = mgu((X₁,· · ·,Xₙ),(t₁,· · ·,tₙ))` and `Ψ' = { [Y₁] ∧ · · · ∧ [Yₘ] }`. By properties of most general unifiers, `σ ⊆ σ'`. Furthermore, by idempotence of substitutions, `σσ' = σ'`, hence `Yᵢσ' ≠ rᵢσ` for all `i ∈ {1, · · · , m}`. Consequently, `σ'` is a solution of `Ψ'`.

**5.8 Correctness of the decision procedure for REACH in CAPS in DC**

In this section we prove Theorem 1, by showing that `reach` (Algorithm 1) is correct; that is, it is sound, complete and terminates in finite time for any `CAP` in `DC`.
Theorem 17. For any input \texttt{REACH}(\texttt{cap}, G), with \texttt{cap} = ((S, \Sigma, V), \pi_1, \cdots, \pi_\ell, \Lambda) being a \texttt{CAP} in \texttt{DC}, and \( G = \pi : a \) being a goal such that either \( \pi = \Lambda \) and \( a \in T_{\Sigma(\emptyset)}^{\text{msg}} \) or \( \pi \in \{ \pi_1, \cdots, \pi_\ell \} \) and \( a \in T_{\Sigma(\emptyset)}^{\text{inf}} \), Algorithm 1 terminates in finite time.

Proof. It follows trivially from the number of finite interleavings stemming from \texttt{CAP}s in \texttt{DC}, and termination of procedure \texttt{solve}. Termination of procedure \texttt{solve} is also clear from termination of procedures \texttt{reduceDY}, \texttt{solvePos}, \texttt{solveNeg} and \texttt{checkProv} (cf. Theorems 2, 5 and Section 5.7).

In the following we assume that all guarded send events in \texttt{cap} are of the form \( g_3g_2 \triangleright \text{snd}(m) \), where either \( g_3 = \{ a \} \) and \( g_2 = \emptyset \), or \( g_3 = \{ a \} \) and \( g_2 = \emptyset \); the extension to unrestricted \texttt{CAP}s in \texttt{DC} is straightforward, see Remark 3. Then, given a trace \( z_0e_1z_1 \cdots e_nz_n \) of a \texttt{CAP} (cf. Section 2.2), for each event \( e \in \{ e_1, \cdots, e_n \} \) there is exactly a constraint \( e \) in the constraint system generated by procedure \texttt{cs} that reflects the evaluation of the conditions that enable event \( e \) (cf. Section 5.7); with abuse of notation, we write \( s \models e \) to denote \( e \). For example, if \( s = (\Omega_\pi, \Omega_{\pi_2}, \Omega_\Lambda) \) and \( e \) is the event \( \{ a \} \) \ \triangleright \text{snd}(m) \) performed by process \( \pi_2 \), then \( s \models e \) denotes the constraint \( \Omega_{\pi_2} \models e \). Observe that \( s \) is the symbolic configuration computed by procedure \texttt{cs} (see Section 5.7) that symbolically models the concrete configuration \( z \). Similarly, we write \( s \models G \) to denote the constraint that reflects the evaluation of goal \( G \) at symbolic configuration \( s \).

Given a ground configuration \( z \) and an event \( e \), we write \( z \models e \) to denote that \( s \) satisfies the conditions that enables \( e \); similarly, we write \( z \models G \) the goal \( G \) is satisfied at configuration \( z \).

Theorem 18. Let \( \texttt{cap} = ((S, \Sigma, V), \pi_1, \cdots, \pi_\ell, \Lambda) \) be a \texttt{CAP} in \texttt{DC}, and \( G = \pi : a \) be a goal such that either \( \pi = \Lambda \) and \( a \in T_{\Sigma(\emptyset)}^{\text{msg}} \) or \( \pi \in \{ \pi_1, \cdots, \pi_\ell \} \) and \( a \in T_{\Sigma(\emptyset)}^{\text{inf}} \). If Algorithm 1 terminates successfully for the reachability problem instance \texttt{REACH}(\texttt{cap}, G), then \texttt{REACH}(\texttt{cap}, G) = \true.
\[ s_{i-1} \sigma \vdash e_i \sigma. \] If \( s_{i-1} \vdash e_i \) is a negative policy constraint then by soundness of procedure \textit{solveNeg} (cf. Theorem 13) \( s_{i-1} \sigma \vdash e_i \sigma. \) Also, the application of procedure \textit{checkProv} to the provision formula \( \Psi \) is sound; cf. Theorem 15.

It remains to prove that the solution \( \sigma \) is sound when facts are retracted from the extensional knowledge of the processes in \( \text{cap} \); that is, \( s_{i-1} \sigma \vdash e_i \sigma, \) when \( \sigma \) instantiates the provision formula \( \Psi, \) implies \( z_{i-1} \vdash e_i \sigma. \) Recall that \( \Psi \) is the provision formula constructed by procedure \textit{solve} that includes (1) the solutions of the negative constraints in \( CS \) and (2) all safety provision formulas of terms used for the proofs of policy constraints in \( CS \), stored by associating the retraction record function \( \Upsilon \) to each symbolic configuration. Moreover, observe that \( z_i \subseteq s_i \sigma \) for all \( i \in \{0, \cdots, n\} \) (obvious by construction of the constraint system \( CS \), where terms in a symbolic configuration are always carried over to subsequent symbolic configurations).

Assume, by contradiction, that a proof of a constraint \( s_{i-1} \sigma \vdash e_i \sigma \) uses a term \( a \in s_{i-1} \sigma \) such that \( a \notin z_{i-1}. \) Then term \( a \) has been removed by a negative update \( u_- \) occurring prior to configuration \( z_{i-1} \); that is, there was a match \( \mu \in z_j, \) for some \( j \in \{0, \cdots, i - 2\}, \) such that \( \mu \sigma = u_- \sigma \) and \( a \in \mu \sigma. \) In particular, \( \sigma \) refines the most general unifier \( \theta = \text{mgu}(\mu, u_-) = \{X_1 \leftarrow t_1, \cdots, X_k \leftarrow t_k\}. \) By construction of the retraction record function \( \Upsilon \) at configuration \( z_{i-1}, \) \( \Upsilon(a) \) contains the provision clause \( [X_1 \neq t_1 \lor \cdots \lor X_k \neq t_k]. \) Recall that upon use of \( a \) the provision formula \( \Upsilon(a) \) is added to the provision formula \( \Psi. \) Since \( \sigma \) instantiates \( \Psi, \) then \( X_q \sigma \neq t_q \sigma \) for some \( q \in \{1, \cdots, k\}, \) leading to a contradiction because \( \sigma \models \theta \) and \( X_q \leftarrow t_q \in \theta. \)

Similar reasoning applies trivially to the constraint \( s_n \vdash G \) that models the evaluation of the goal \( G. \)

\[ \square \]

**Theorem 19.** Let \( \text{cap} = ((S, \Sigma, \mathcal{V}), \pi_1, \cdots, \pi_\ell, \mathcal{A}) \) be a \text{cap} in \( \text{DC}, \) and \( G = \pi : a \) be a goal such that either \( \pi = \mathcal{A} \) and \( a \in T^\text{msg}_\Sigma(0) \) or \( \pi \in \{\pi_1, \cdots, \pi_\ell\} \) and \( a \in T^\text{infom}_\Sigma(0). \) If \( \text{REACH}(\text{cap}, G) = \top \) then Algorithm 1 terminates successfully for the reachability problem instance \( \text{REACH}(\text{cap}, G). \)

**Proof.** Assume substitution \( \sigma : \mathcal{V}^3 \rightarrow T^\Sigma(0) \) is a solution of the reachability problem instance \( \text{REACH}(\text{cap}, G). \) Let \( z_0 e_1 z_1 \cdots e_n z_n \) be a trace of \( \text{cap} \) such that \( z_n \vdash G \) and \( \sigma : \mathcal{V}^3 \rightarrow T^\Sigma(0) \) be a ground substitution such that \( z_0 \sigma \models z_1 \cdots e_n \sigma = z_n. \)

For any event \( e_i \) with \( i \in \{1, \cdots, n\}, \) if \( e_i \) is a receive event then \( s_{i-1} \vdash e_i \) is an attacker constraint; by completeness of procedure \textit{reduceDY} (see Theorem 4), \textit{reduceDY} solves the constraint for a substitution \( \sigma_1 \subseteq \sigma. \) If \( e_i \) is a guarded send event with non-empty positive guard, then \( s_{i-1} \vdash e_i \) is a positive policy constraint; by completeness of procedure \textit{ps} (see Theorem 7) \textit{ps} solves the constraint for a substitution \( \sigma_2 \subseteq \sigma. \) If \( e_i \) is a guarded send event with non-empty negative guard, then \( s_{i-1} \vdash e_i \) is a negative policy constraint; procedure \textit{solveNeg}, for the
constraint $s_{i-1} \vdash e_i$, returns a provision formula $\Delta$, that is added to the provision formula $\Psi$ stored by $solve$. By Theorem 14 there exists an instantiation of $\sigma_3$ of $\Delta$; in particular, $\sigma_3 \subseteq \sigma$. Finally, the application of procedure $checkProv$ is complete: there exists a substitution $\sigma_4 \subseteq \sigma$ returned by $checkProv$; cf. Theorem 16.

It is clear that $\sigma_1 \sigma_2 \sigma_3 \sigma_4 \subseteq \sigma$. It remains now to show that the provision formula $\Psi$ stored by $solve$ is satisfied by $\sigma$. Assume, by contradiction, that $\sigma$ does not satisfy $\Psi$. The conjuncts of the $\Psi$ are provision clauses that stem either from the analysis of negative constraints or from the use of potentially retracted terms in the proofs of policy constraints. Since the former are instantiated by $\sigma$, $\sigma$ must not satisfy $\Psi$ on one of the latter; then there is a provision clause $\delta = [X_1 \not\approx t_1 \lor \cdots \lor X_k \not\approx t_k]$ in $\Psi$ that is not satisfied by $\sigma$, i.e. $X_j \sigma = t_j \sigma$ for all $j \in \{1, \cdots, k\}$. The provision clause $\delta$ has been introduced by a proof of a policy constraint $s_i \sigma \vdash e_{i+1} \sigma$ (with $i \in \{1, \cdots, n-1\}$) that uses some fact $a$ for which $\delta$ is a conjunct of the safety provision formula $\Upsilon(a)$ of $a$. Then, by construction of the retraction record function $\Upsilon$, there exists a negative update $u_-$ performed at configuration $z_j$ (with $i \in \{0, \cdots, i-1\}$) and a match $\mu$ in the extensional knowledge of the process such that $\theta = mgu(\mu, u_-) = \{X_1 \mapsto t_1, \cdots, X_k \mapsto t_k\}$ and $a \in \mu \sigma$. Since $\theta \subseteq \sigma$, then $a \not\in z_i$, and consequently $z_i \not\vdash e_{i+1} \sigma$, which contradicts the hypotheses.

Similar reasoning applies trivially to the deduction $z_n \vdash G$, representing the satisfaction of the goal $G$. $\Box$
Chapter 6

Reducing reachability in CAPs to secrecy

In this chapter, we identify a fragment of CAPs that admits decision algorithms for the reachability problem. The fragment, referred to as DC₁, is strictly smaller than DC (cf. Chapter 4). However, the decision procedure given for the DC₁ fragment follows the alternative approach of reducing reachability problems in the context of CAPs to secrecy problems in the context of security protocols. To show the reduction, we encode the derivation of policy statements in the policy of a process into message inference trees induced by the Dolev-Yao deduction rules. The encoding benefits us in two ways: (1) it simplifies the decidability proof, and (2) it allows us to build upon existing tools which have been originally developed for verifying security protocols.

6.1 The DC₁ class of CAPs

In DC₁, the policy of the attacker is fixed to the standard Dolev-Yao (DY) deduction rules, as formalized in Section 4.1. The policies of honest processes are centered around trust application TA, and (transitive) trust delegation TD rules, adopted and adapted from DKAL, and can also express typical RBAC systems with role hierarchy.

Definition 3. Fragment DC₁ consists of all CAPs cap = ((S, Σ, V), π₁, · · · , πℓ, A), which satisfy the following syntactical conditions:

- (S, Σ, V) is a signature such that:
  - A finite subset of constants in Σ of type msg, denoted by Agents, represents the set of the names of the processes in π₁, · · · , πℓ.

\[\text{Intuitively, one (or more) public key is attributed to each element of Agents. The public keys are known to everyone, and to the attacker in particular. Using the public key of } a \in \text{Agents one can encrypt messages for } a \text{ and can verify the authenticity of the messages signed by } a.\]
Apart from nullary functions (i.e. constants), the only functions of type \( \text{msg} \) contained in \( \Sigma \) are \( \{\cdot\}, \{\cdot\} \), \( \text{pk}(\cdot) \), \( \text{sk}(\cdot) \), \( \text{sig}(\cdot, \cdot) \) and \( (\cdot, \cdot) \). These represent respectively asymmetric and symmetric encryption, public and secret key constructors, digital signature, hash and pairing functions, interpreted as usual.

\( \Sigma \) contains, in particular, the functions \( \text{tdOn}(\cdot, \cdot) \) and \( \text{said}(\cdot, \cdot) \) of type \( \text{infon} \), with \( \text{tdOn}, \text{said} : \text{Agents} \times \text{infon} \rightarrow \text{infon} \). These stand for trusted on and said, respectively.

Any process \( \pi = (\eta^n_\pi, \Omega^n_\pi, I^n_\pi) \) with \( \pi \in \{\pi_1, \cdots, \pi_I\} \) meets the conditions:

- For all threads in \( \eta^n_\pi \):
  * All the terms sent and received are elements of \( T_{\Sigma(V)}^{\text{msg}} \), i.e. terms of type \( \text{msg} \).
  * All send events in the thread are of the form
    \( g_\exists \{\cdot\} \triangleright \text{snd}(m) \)
    with \( g_\exists \) being a finite set of terms of type \( \text{infon} \) and \( m \) being a term of type \( \text{msg} \). Furthermore, every variable \( v \in \text{var}(g_\exists) \) originates in a previous receive event (see the notion of origination of a variable in Section 2.1).
  * All receive events in the thread are of the form
    \( \text{rcv}(m) \triangleright u_+ \{\cdot\} \)
    with \( m \) being a term of type \( \text{msg} \) and \( u_+ \) being a finite set of terms of type \( \text{infon} \). Furthermore, for every variable \( v \in \text{var}(u_+) \), either \( v \in \text{var}(m) \) or \( v \) originates in a previous receive event; cf. origination of variables in Section 2.1.

- \( \Omega^n_\pi \subseteq T_{\Sigma(\emptyset)}^{\text{infon}} \) is a finite set of ground terms type \( \text{infon} \).
- \( I^n_\pi \) includes the \( \text{TA} \) and \( \text{TD} \) rules
  \[
  X \leftarrow \text{tdOn}(A, X), \text{said}(A, X) \quad (\text{TA})
  \text{tdOn}(A, \text{tdOn}(B, X)) \leftarrow \text{tdOn}(A, X) \quad (\text{TD})
  \]

The set of all the other rules in \( I^n_\pi \) constitutes a type-1 theory, as defined below, and \( \text{said} \) and \( \text{tdOn} \) do not appear in this set.

\( A^n = (\Omega^n_A, I^n_A) \), where \( \Omega^n_A \subseteq T_{\Sigma(\emptyset)}^{\text{msg}} \) is a finite set of ground terms of type \( \text{msg} \), and \( I^n_A \) consists of the message rules that reflect the capabilities of the standard DY attacker as formalized in Section 4.1.

In the remainder of the chapter, we will write \( g_\exists \triangleright \text{snd}(m) \) for events of the form \( g_\exists g_\exists \triangleright \text{snd}(m) \), and \( \text{rcv}(m) \triangleright u_+ \) for events of the form \( \text{rcv}(m) \triangleright u_+ u_- \).
indeed, neither negative guards nor negative updates are allowed in the DC\textsubscript{1} fragment.

We define type-1 theories in order to extend the policies of (honest) processes beyond TA and TD, e.g. to express typical RBAC systems with role hierarchy.

**Definition 4.** A finite set of infon clauses $T$, defined over signature $(\Sigma, \Sigma, \mathcal{V})$ is called a type-1 theory, iff

(a) For all $a \leftarrow a_1, \cdots, a_n$ in $T$, $\bigcup_{i \in \{1, \cdots, n\}} \text{var}(a_i) \subseteq \text{var}(a)$.

(b) The infon dependency graph of $T$ is acyclic. The infon dependency graph of $T$ is a directed graph defined by the pair $(\Sigma_{\text{infon}}, \text{Edges})$ with $\Sigma_{\text{infon}}$ being the set of function constructors of type infon in $\Sigma$ and $(f, g) \in \text{Edges}$ iff there exists an infon clause in $T$ using $g$ in its head, and $f$ in its body.

Observe that the definition of infon dependency graphs is purely syntactical; indeed, it does not take the arguments of infons into account. We remark that neither TA nor TD fall into type-1 theories, due to conditions (a) and (b) in Definition 4, respectively.

**Example 8.** In this example we look at a variant of the policy presented in Section 3.1 that falls in the type-1 fragment. This variant, unlike the original policy, does not account for transitivity of attributes of files and links to files.

Consider a file server which implements an RBAC system with two roles, user and admin. Users may read any public file, admins may read any classified file, and admins may also write to any file. Admins inherit all the rights attributed to users. Below, we give a type-1 theory which describes this RBAC system.

\[
\begin{align*}
\text{user}(A) & \leftarrow \text{admin}(A) \\
\text{can_read}(A, F) & \leftarrow \text{user}(A), \text{public}(F) \\
\text{can_write}(A, F) & \leftarrow \text{admin}(A), \text{classified}(F)
\end{align*}
\]

Here \(\Sigma\) contains the set of identities of involved processes, the set of names of files, the set of usual cryptographic functions, and the set of infon constructors \(\Sigma_{\text{infon}} = \{\text{user}, \text{admin}, \text{public}, \text{classified}, \text{can_read}, \text{can_write}\}\), with obvious arities. The infon dependency graph for this theory, shown in Figure 6.1, is indeed acyclic.

(end of example)

In Section 6.2, we give a decision algorithm for the reachability problem for CAPs in the DC\textsubscript{1} fragment. The decision algorithm is based upon encoding policy level computations of processes into message derivation trees of the standard DY model.

### 6.2 A decision procedure for REACH for CAPs in DC\textsubscript{1}

In this section, we give an algorithm for deciding REACH in CAPs belonging to DC\textsubscript{1}. We start with presenting an encoding from policy statements (i.e., infon
terms) into message terms. Then, we present an algorithm for deciding reachability of the “encoded” DC₁ CAPs that builds upon the constraint solving algorithm of Millen and Shmatikov.

The purpose of the encoding is to replace the logic programs of the participating processes in a DC₁ CAP with the derivation rules of the DY model. Then, intuitively, the attacker and all the processes would be equipped with the reasoning power of the DY model, which is well understood and comes with decision algorithms for reachability.

### 6.2.1 Encoding policy level computations

The encoding consists of two functions: \( \zeta : \mathcal{T}^{\text{infon}}_{\Sigma(\mathcal{V})} \rightarrow \mathcal{T}^{\text{msg}}_{\Sigma(\emptyset)} \) which maps infons to message terms, and \( \mathcal{E} \) which maps infons to “disjunctive” guards (see Remark 1). We start with an initial encoding for trust application only. Then, we extend the encoding to trust delegation and type-1 theories.

**TA only.** We recursively define the encoding for infon \( i \):

\[
\zeta(i) = \begin{cases} 
\{ \zeta(X), \text{sig}(\bar{A}, \zeta(X)) \} & \text{if } i = \text{said}(A, X) \\
\text{tdOn}(\bar{A}, \zeta(X)) & \text{if } i = \text{tdOn}(A, X) \\
i & \text{otherwise}
\end{cases}
\]

where \( \bar{\cdot} : \text{Agents} \rightarrow \bar{\text{Agents}} \) is a bijection which associates a fresh unique name to each element of \( \text{Agents} \). Elements of \( \bar{\text{Agents}} \) belong to \( \mathcal{T}^{\text{msg}}_{\Sigma(\emptyset)} \), and are defined solely for the encoding function \( \zeta \), i.e. they do not appear in the specifications of CAPs. In particular, the attacker does not know their identities. Here, we identify the members of \( \bar{\text{Agents}} \) with the private keys associated to them.

Consider as an example the ground infon \( \text{said}(a, \text{key}(k, x)) \), with \( a \in \text{Agents} \) and \( k, x \in \mathcal{T}^{\text{msg}}_{\Sigma(\emptyset)} \). The infon constructor \( \text{said}(\cdot, \cdot) \) is interpreted as usual (cf. Definition 3) and the infon constructor \( \text{key}(\cdot, \cdot) \) associates a cryptographic key (first argument) to its owner (second argument). The encoding of \( \text{said}(a, \text{key}(k, x)) \) is the ciphertext \( \{ \text{key}(k, x), \text{sig}(\bar{a}, \text{key}(k, x)) \} \text{tdOn}(\bar{a}, \text{key}(k, x)) \) from which the infon \( \text{key}(k, x) \) (i.e. what process \( a \) said) can be obtained using the symmetric decryption (Sdec) rule of DY only if the key \( \text{tdOn}(\bar{a}, \text{key}(k, x)) \) (i.e. \( a \) is trusted on...
key(k, x)) is obtained first. This indicates that if TA is applicable on a set of facts at the policy level, then the DY rule

\[ X \leftarrow \{X\}_K, K \quad {\text{Sdec}} \]

is applicable to the terms resulting from the encoding.

Intuitively, the role of the subterm \( \text{sig}(\bar{a}, \text{key}(k, x)) \) is to ensure that terms of the form \( \{\text{key}(k, x), \text{sig}(\bar{a}, \text{key}(k, x))\}_\text{TD} \) can not be forged by a DY attacker; instead, \( \{\text{key}(k, x), \text{sig}(\bar{a}, \text{key}(k, x))\}_\text{TD} \) is known to the attacker only if the fact \( \text{said}(a, \text{key}(k, x)) \) can be derived in the policy level. Recall that the attacker does not know the private key of \( \bar{a} \).

**TA, TD and type-1 theories.** In order to include TD and type-1 theories in the encoding, we define an expansion function \( \mathcal{E} \) that for any infon term returns a disjunctive guard. That is, \( \mathcal{E} \) associates to any infon term a disjunction of sets of infon terms, each interpreted as a conjunction. We have shown in Remark 1 (see Section 2.2) how such “disjunctive” guards can be integrated in our formalism, and how they can be reflected in the semantics of a specification. In the remainder of this chapter, we will refer to guards as to disjunctions \( g_1 \lor \cdots \lor g_n \) where \( g_i \), for all \( i \in \{1, \cdots , n\} \), is a set of infon terms.

We motivate the expansion function via a simple example. Suppose that the fact \( \text{tdOn}(a, i) \) is present in the knowledge of a process \( \pi \), and that the guard \( \text{tdOn}(a, \text{tdOn}(b, i)) \) is to be evaluated. The TD rule implies that the guard can be derived, while there is no corresponding inference tree (in the Dolev-Yao model) for \( \zeta(\text{tdOn}(a, \text{tdOn}(b, i))) \), given \( \zeta(\text{tdOn}(a, i)) \). However, the set of infons which yield \( \text{tdOn}(a, \text{tdOn}(b, i)) \) via applying only the TD rule is finite. This set of infons can be seen as a guard, namely \( \{\text{tdOn}(a, \text{tdOn}(b, i))\} \lor \{\text{tdOn}(a, i)\} \). The fact that \( \text{tdOn}(a, i) \) yields \( \text{tdOn}(a, \text{tdOn}(b, i)) \) in the policy of process \( \pi \) is reflected in the DY model by: either \( \zeta(\text{tdOn}(a, \text{tdOn}(b, i))) \) or \( \zeta(\text{tdOn}(a, i)) \), or both, are obtained from \( \zeta(\text{tdOn}(a, i)) \).

The expansion function \( \mathcal{E}_P(Q, i) \) is defined for finite sets of infon clauses \( P \) and \( Q \), and infon \( i \):

\[
\mathcal{E}_P(\emptyset, i) = \{i\} \\
\mathcal{E}_P(\{r \leftarrow r_1, \ldots , r_t\} \cup Q', i) = \\
\begin{cases} 
\mathcal{E}_P(Q', i) \lor (\mathcal{E}_P(P, r_1) \cup \cdots \cup \mathcal{E}_P(P, r_t)) & \text{if } i = r \rho \\
\mathcal{E}_P(Q', i) & \text{if } \neg \exists r. \ i = r \rho 
\end{cases}
\]

where \( \cup \) distributes over \( \lor \), i.e. \( S \cup (S_1 \lor S_2) = (S_1 \lor S_2) \cup S = (S_1 \cup S) \lor (S_2 \cup S) \). Here \( Q \) is a support theory used only to ensure that the expansion of any infon results in a finite set; see Lemma 8 below.

**Lemma 8.** Let \( P = P^1 \cup \{TD\} \), with \( P^1 \) being the type-1 theory in a process of a DC\(_1\) CAP. Then, \( \mathcal{E}_P(P, i) \) is a finite set for any infon \( i \).
Proof. Immediate, since the dependency graph of $P^1$ is acyclic, $P^1$ does not contain $tdOn$, and the infon clause which encodes $TD$ strictly decreases the number of $tdOn$ functions. 

We write $E(i)$ for $E(P, i)$, when $P$ is clear from the context. We also write $g \in E(i)$ if $E(i) = g_1 \lor \cdots \lor g \lor \cdots \lor g_n$, with $n \geq 1$.

Example 9. Consider the infon $i = \text{can}_\text{read}(a, \text{file})$ along with the type-I theory of Example 8. Then,

$$E(i) = \{\text{user}(a), \text{public}(\text{file})\} \lor \{\text{admin}(a), \text{public}(\text{file})\} \lor \{\text{admin}(a), \text{classified}(\text{file})\} \lor \{\text{can}_\text{read}(a, \text{file})\}$$

We write $E(i) = g_1 \lor g_2 \lor g_3 \lor g_4$, with $g_1 = \{\text{user}(a), \text{public}(\text{file})\}$, etc. The guard $E(i)$ is interpreted as: $\text{can}_\text{read}(a, \text{file})$ holds, i.e. $a$ can read file in example 8, iff at least one of the following conditions holds:

1. $g_1$ user(a) and public(file) are known, or
2. $g_2$ admin(a) and public(file) are known, or
3. $g_3$ admin(a) and classified(file) are known, or
4. $g_4$ can_read(a, file) is known via an inference outside the RBAC system of Example 8.

Similarly, we get $E(\text{can}_\text{write}(a, \text{file})) = \{\text{admin}(a)\} \lor \{\text{can}_\text{write}(a, \text{file})\}$. 

(end of example)

We refine the function $\zeta$ (introduced above) by incorporating the expansion function $E$ into $\zeta$. This ensures that $\zeta(i)$, for infon $i$, is obtainable from $\zeta(\text{said}(a, i))$ if there exist at least one $g \in E(tdOn(a, i))$ such that $\zeta(g)$ can be obtained first. Hence, we define:

$$\zeta(i) = \begin{cases} \{\zeta(X), \text{sig}(\bar{A}, \zeta(X))\} & \text{if } i = \text{said}(\bar{A}, X) \\ \zeta(E_P(tdOn(\bar{A}, X))) & \text{if } i = tdOn(\bar{A}, X) \\ \text{otherwise} & \end{cases}$$

Here, $\{x\}_{k_1, \ldots, k_t}$ stands for the tuple $\{x\}_{k_1} \cdots \{x\}_{k_t}$, the function $\zeta$ distributes over $\lor$, and $P = P^1 \cup \{TD\}$ with $P^1$ being the type-1 theory at hand. For a finite set of infons $g$, $\zeta(g)$ is defined as the concatenation (i.e. nested pairing) of $\zeta(i)$, for all $i \in g$.

We remark that elements of $E_P(P, tdOn(\bar{A}, X))$ in the definition of $\zeta$ are singletons. This is because in any CAP belonging to DC$_1$, $E_P(P, tdOn(a, i)) = E_{\{TD\}}(\{TD\}, tdOn(a, i))$, as $tdOn$ does not appear in $P^1$.  

102
Correctness of the encoding. We remark that the purpose of the proposed encoding is to replace the policy inference theories of processes with the derivation rules of the Dolev-Yao model. The following theorem ensures that if a policy fact is derivable in the policy of a process, then its corresponding encoded message term can be derived using the DY inference rules, and vice versa. We consider the standard DY capabilities for the term algebra $T_{\Sigma(V)}$, which comprises both infon and message constructors. That is, the infon constructors are seen as uninterpreted functions, while message constructors (e.g. $\{\cdot\}$) have their standard meaning in the DY model; in particular, the attacker deduction will employ only the standard DY inference rules as formalized in Section 4.1.

**Theorem 20.** Let $P$ be the policy of a process in a $CAP$ belonging to $DC_1$, with $P = \{TA\} \cup \mathcal{Q}$, $\mathcal{Q} = \{TD\} \cup P^1$ and $P^1$ being a type-I theory. For any (ground) infon $f$ and finite set of (ground) infons $G$

$$G \vdash^P f \iff \exists g \in \varepsilon_{\mathcal{Q}}(Q, f). \quad \zeta(G) \vdash^A \zeta(g),$$

where $\vdash^P$ is the ground deduction relation induced by theory $P$ and $\vdash^A$ is the ground deduction relation induced by the Dolev-Yao attacker theory $I_A$, as formalized in Section 4.1.

**Proof.** Fix the set of infon clauses $P$. We write $\varepsilon(f)$ in place of $\varepsilon_{\mathcal{Q}}(Q, f)$, and $G \vdash \{g_1, \cdots, g_n\}$ for $G \vdash g_1, \cdots, G \vdash g_n$. The proof is split into two directions.

$\Rightarrow$ We use structural induction on proof trees for $f$, given $G$. If $f \in G$, then the implication is trivial. Otherwise, consider the last rule applied in the proof tree:

- (TA) Then $G \vdash^P said(a, f), tdOn(a, f)$, for some $a \in Agents$. By induction hypotheses,

$$\exists s \in \varepsilon \left(\varepsilon_{\{said(a, f)\}}, \varepsilon_{\{tdOn(a, f)\}}. \zeta(G) \vdash^A \{\zeta(s), \zeta(t)\} \right).$$

Observe that, since no rule of $Q$ mentions the infon constructor $said$, $s = said(a, f)$. Consequently

$$\zeta(s) = \zeta(said(a, f)) = \{\zeta(f), \text{sig}(\pi, \zeta(f))\}_{\zeta(\varepsilon_{\{tdOn(a, f)\}})}$$

Since $t \in \varepsilon \left(\varepsilon_{\{tdOn(a, f)\}}, \varepsilon_{\{tdOn(b, i)\}}. \zeta(G) \vdash^A \zeta(t)\right)$, by applying the $Sdec$ rule and unpairing we get $\zeta(G) \vdash^A \zeta(f)$.

- (TD) Then $f = tdOn(a, tdOn(b, i))$ for some $a, b \in Agents$ and $i \in \text{infon}$, and $G \vdash^P tdOn(a, i)$. By induction hypotheses, $\exists t \in \varepsilon \left(\varepsilon_{\{tdOn(a, i)\}}. \zeta(G) \vdash^A \zeta(t)\right)$. Now, the claim follows since for any infon $t$, $t \in \varepsilon \left(\varepsilon_{\{tdOn(a, i)\}}. G \vdash^P tdOn(a, i)\right)$ implies $t \in \varepsilon \left(\varepsilon_{\{tdOn(a, tdOn(b, i))\}}. G \vdash^P tdOn(a, tdOn(b, i))\right).$
• (Type-1) Let \( R = r \leftarrow r_1, \ldots, r_\ell \in P^1 \) be the last rule applied. Then \( f = r \rho \) and \( G \vdash^P r_1 \rho, \ldots, r_\ell \rho \), for some grounding substitution \( \rho \) (cf. condition (b) in Definition 4). By induction hypotheses, \( \exists r'_1 \in \mathcal{E}(r_1 \rho), \ldots, r'_\ell \in \mathcal{E}(r_\ell \rho). \zeta(G) \vdash^A \zeta(r'_1), \ldots, \zeta(r'_\ell) \). By definition of \( \mathcal{E} \), \( \{r'_1, \ldots, r'_\ell\} \in \mathcal{E}(f) \), hence follows the claim.

\[ \Leftarrow \]
First, we claim that \( \zeta(G) \vdash^A \zeta(g) \) implies \( G \vdash^P g \), for a (ground) infon \( g \) and a finite set of (ground) infons \( G \). Notice that the \( \zeta(g) \) is either of the form \( \zeta(x), \sigma(x) \) or of the form \( i(x) \), with \( i \) being an infon constructor. The claim follows by case analysis on the DY attacker’s message (de)composition abilities. In particular, note that (1) to fabricate the term \( \zeta(x), \sigma(x) \), the attacker needs to construct \( \sigma(x) \), which is impossible as the attacker does not own the private key of any \( a \in \mathcal{A} \), and (2) infon constructors are uninterpreted functions in the DY model, i.e. they can neither be applied by the attacker, nor be deconstructed. The other cases are straightforward; we thus omit them here. Finally, notice that if \( G \vdash^P g \) and \( g \in \mathcal{E}(f) \), then \( G \vdash^P f \); hence follows the claim.

This completes our proof.

\[ \square \]

6.2.2 A decision algorithm for reachability

We begin with a brief description of Millen and Shmatikov’s constraint solving algorithm for deciding reachability in cryptographic protocols [50]. Recall that participants are specified as sequences of communication (i.e. send and receive) events in [50], and the reachability problem, given a message \( s \), asks whether there exists a reachable configuration \( z \) of the protocol where \( T(z) \vdash^A s \), with \( T(z) \) being the attacker’s knowledge in \( z \).

The algorithm of [50] searches the finite set of interleavings of (symbolic) actions performed by the participants and a fictitious test thread, which receives \( s \) and then sends stop to the search algorithm. For each interleaving, a sequence \( C \) of attacker constraints is constructed. An attacker constraint is a pair \( T \vdash^A m \), where \( m \) is a term that the attacker should derive from the set of terms \( T \), using his inference capabilities (cf. Section 5.1). The constraint sequence is built for each interleaving as: when a participant sends a message term, the term is added to the attacker term set, and when a receive action occurs, a constraint \( T \vdash^A m \) is added to \( C \), with \( m \) being the term that is to be received and \( T \) is the current attacker term set.

A solution for a constraint sequence \( C = T_1 \vdash^A m_1 \cdots T_i \vdash^A m_i \cdots T_n \vdash^A m_n \), with \( m_i = s \), is a (grounding) substitution \( \sigma : \text{var}(T_1 \vdash^A m_1 \cdots T_i \vdash^A m_i) \rightarrow T^{\text{msg}}(\emptyset) \) such that \( T_j \sigma \vdash^A m_j \sigma \), for \( 1 \leq j \leq i \); here, \( \text{var}(c_1 \cdots c_i) \) is the set of variables appearing in the constraints \( c_1, \ldots, c_i \). In our presentation, therefore, we account for partial executions as well, cf. [24]. Millen and Shmatikov’s
algorithm applies a number of reduction rules which reduce $C$ to a sequence of immediately (un)satisfiable constraints. In the following, we refer to their reduction procedure as $MSReduce$. If $MSReduce$ does not succeed for $C$ (i.e. $C$ is unsatisfiable), next interleaving is considered, until all the interleavings are exhausted. If one of the constraint sequences is satisfiable, then the (supposedly) secret message $s$ is revealed to the attacker. That is, an attack is found. Otherwise the protocol is correct, i.e. $s$ is not revealed to the attacker, for the instantiation at hand.

The following three observations enable us to use Millen and Shmatikov's procedure (with minor extensions, see Section 6.2.2) for deciding reachability for $DC_1$ CAPs:

1. Checking guards in, and making updates to, the policies of the processes can be emulated by communication actions. Namely, sending an infon to the knowledge set of a process reflects updating the policy of that process, while receiving an infon derived from the knowledge set reflects querying the policy of the process for evaluating a guard.

2. The encoding presented in Section 6.2.1 entails that the same inference rules which are used for the attacker (namely the standard DY message derivation capabilities) can model the computations of the participants at their policy level. Therefore, the send and receive actions which would emulate checking guards and updating knowledge sets of processes (mentioned above) can be treated with the constraint solving procedure that one would use for the attacker knowledge (here, $MSReduce$).

3. Renaming the infon constructors for each process of the CAP ensures that DY deductions on the encoded terms of two processes (or a process and the attacker) never interfere with each other, as the policy rules of one do not apply to encoded terms of the other.

Algorithm 11 Constraint solving for deciding reachability of CAPs in $DC_1$

```latex
\begin{algorithm}
\textbf{REQUIRES:} \textsc{Reach}(\text{cap}, a : f), \text{cap} = (\langle S, \Sigma, V \rangle, \pi_1, \ldots, \pi_\ell, A)
\text{rename}_\text{infons}(\text{cap})
\text{expand}(\{\pi_1, \ldots, \pi_\ell\})
I := \text{interleavings}(\{\pi_1, \ldots, \pi_\ell\}, a, f)
\textbf{for all } i \in I \text{ do}
\text{flatten}(i)
C := \text{constraint}_\text{sequence}(i)
\text{trace} := \text{MSReduce}(C)
\text{if } \text{trace} \neq \emptyset \text{ then}
\text{return } (\text{reach} : \text{trace})
\text{return } (\text{unreach})
\end{algorithm}
```

Algorithm 11 details our constraint solving procedure for deciding reachability in CAPs belonging to $DC_1$. The input to the algorithm is a reachability
problem $\text{REACH}(\text{cap}, a : f)$, with $\text{cap} = ((\Sigma, \Sigma, V), \pi_1, \ldots, \pi_\ell, A)$ being a $\text{CAP}$ in $\text{DC}_1$. The algorithm either returns a $\text{trace}$ witnessing $\text{REACH}(\text{cap}, a : f) = T$, or returns $\text{unreach}$ if $\text{REACH}(\text{cap}, a : f) = F$.

To improve readability, in the remainder of the chapter we will write $g_\exists \triangleright\triangleright \text{snd}(m)$ for guarded send events of the form of $g_\exists\{\} \triangleright\triangleright \text{snd}(m)$ and $r\text{cv}(m) \triangleright\triangleright u_+$ for receive events followed by updates of the form $r\text{cv}(m) \triangleright\triangleright u_+\{\}$; indeed, negative guards and negative updates are always empty sets in specifications in $\text{DC}$.

Algorithm 11 starts with calling the procedure $\text{rename}_\text{infons}(\text{cap})$. This procedure renames all infons appearing in $\text{cap}$ with fresh infon constructor names. More precisely, $\text{rename}_\text{infons}$ employs a tagging function $\text{tag}_a$ that given a process (here identified with the name $\pi$) and an infon constructor name $i$ associates $i$ to a fresh infon constructor name $i\overline{\pi}$. For each process $\pi \in \{\pi_1, \ldots, \pi_\ell\}$ every term occurring in $\pi$ (that is, in $\pi$’s policy, $\pi$’s extensional knowledge, or in the guards and updates of the threads of $\pi$) is (recursively) rewritten using the tagging function $\text{tag}_a$. Then, if $a \in \{\pi_1, \ldots, \pi_\ell\}$, also the goal $f$ is rewritten using the tagging function $\text{tag}_a$.

Example 10. Let $\pi = \{\{\text{th}\}, \{\text{said}(a, \text{tdOn}(b, f)), \text{tdOn}(a, f)\}, \{TA, TD\}\}$ be a process of a $\text{CAP}$ cap and let the thread $\text{th}$ be as follows:

$$\{\text{tdOn}(b, f)\} \triangleright\triangleright \text{snd}(m)$$

Recall that both $\text{said}$ and $\text{tdOn}$ are functions from $\text{Agents} \times \text{infon}$ to $\text{infon}$. After applying procedure $\text{rename}_\text{infons}$ to $\text{cap}$, process $\pi$ is rewritten as $\pi = \{\{\text{th}'\}, \{\text{said}^{\pi}(a, \text{tdOn}^{\pi}(b, f^{\pi})), \text{tdOn}^{\pi}(a, f^{\pi})\}, \{TA', TD'\}\}$, with $\text{th}'$ being $\{\text{tdon}^{\pi}(b, f^{\pi})\} \triangleright\triangleright \text{snd}(m)$ and $\text{TA'}$ and $\text{TD'}$ being as follows:

$$X \leftarrow \text{tdOn}^{\pi}(A, X), \text{said}^{\pi}(A, X) \quad \text{(TA') }$$

$$\text{tdOn}^{\pi}(A, \text{tdOn}^{\pi}(B, X)) \leftarrow \text{tdOn}^{\pi}(A, X) \quad \text{(TD') }$$

(end of example)

After application of procedure $\text{rename}_\text{infons}$ to $\text{cap}$, for any two distinct processes $\pi_1$ and $\pi_2$ of $\text{cap}$ the set of infons occurring in $\pi_1$ is disjoint from the set of infons occurring in $\pi_2$. More formally, we extend the set of function constructors $\Sigma$ with the disjoint sets $\Sigma_{\text{infon}}^{\pi_1}, \ldots, \Sigma_{\text{infon}}^{\pi_\ell}$, where for each $\pi \in \{\pi_1, \ldots, \pi_\ell\}$ the set $\Sigma_{\text{infon}}^{\pi}$ is the set of infon constructors introduced by the application of $\text{rename}_\text{infons}$ to process $\pi$; we denote the set of infons of process $\pi$ after application of $\text{rename}_\text{infons}$ as $\text{infon}^{\pi}$, defined as the smallest set such that:

- $\forall_{\text{infon}} \subseteq \text{infon}^{\pi}$, and
- $f(t_1, \ldots, t_n) \in \text{infon}^{\pi}$ for any $f \in \Sigma_{\text{infon}}^{\pi}$ (of arity $n$) and $t_i$ being of the correct type. In particular, $f$ associates to each of its arguments a type in $\{\text{msg}, \text{infon}^{\pi}\}$. 

106
It is obvious that the sets of ground infons of the processes of a CAP, after application of procedure `rename_infons`, are pairwise disjoint. Observe also that the sets \( \text{infon}^{|_1}, \ldots, \text{infon}^{|_r} \) do not constitute a partition of the set infon. We will consider the infons in \( \text{infon} \setminus (\text{infon}^{|_1} \cup \cdots \cup \text{infon}^{|_r}) \) as ill-formed, and ignore them in the remainder of the section. In the following, when clear from the context, we will suppress the superscript \( \pi \) from infon constructors.

Algorithm 11 calls then the procedure `expand`\((\{\pi_1, \cdots, \pi_\ell\})\) to expand the guards in each thread, for all the processes \( \pi_1, \cdots, \pi_\ell \). A guard \( g_1 \lor \cdots \lor g_n \) in a process with policy \( P \) is expanded to \( \bigvee_{j \in \{1,\ldots,n\}} \mathcal{E}_Q(Q, g_j) \), where \( Q = P \setminus \{TA\} \) and \( \mathcal{E}_Q(Q, g) = \bigcup_{i \in g} \mathcal{E}_Q(Q, i) \), for a finite set of infons \( g \). Recall that \( \lor \) distributes over \( \lor \) (cf. Section 6.2.1). The `expand` procedure thus rewrites guards into guards.

**Example 11.** Let \( g_3 = \{a, b\} \) be a guard in a process with policy

\[
\{ TA, TD, a \leftarrow c, b \leftarrow d \}.
\]

After application of `expand`, \( g_3 \) is expanded to \( \{a, b\} \lor \{c, b\} \lor \{a, d\} \lor \{c, d\} \).

(end of example)

The procedure `interleavings`\((\{\pi_1, \cdots, \pi_\ell\}, a, f)\) computes the finite set of interleavings of events of the processes \( \pi_1, \cdots, \pi_\ell \). Furthermore, this procedure

(1) adds a testing thread whose sole purpose is to indicate that the search has reached a configuration in which \( \Omega_a \vdash a \), given \( \text{REACH}::<a, f> \); that is, if \( a = A \), then the testing thread simply receives \( f \) and then sends \( \text{stop} \) to the search algorithm. If \( a \not= A \), then the testing thread \( \{f\} \triangleright \text{snd}(<\text{stop}) \) is added to process \( a \).

(2) The `interleavings` procedure treats the disjunction operator \( \lor \) inside guards as branching points, e.g. the interleaving \( \epsilon_1 \cdot (d_3 \triangleright \text{snd}(m)) \cdot \epsilon_2 \), with \( \epsilon_1 \) and \( \epsilon_2 \) being sequences of events and \( d_3 = g_1 \lor g_2 \), gives rise to two interleavings \( \epsilon_1 \cdot (g_1 \triangleright \text{snd}(m)) \cdot \epsilon_2 \) and \( \epsilon_1 \cdot (g_2 \triangleright \text{snd}(m)) \cdot \epsilon_2 \). Consequently, no \( \lor \) appears in guards for any interleaving \( \iota \in I \) in Algorithm 11. Branching interleavings over guards is sound with respect to the semantics of specifications; see Section 2.2, Remark 1.

For each interleaving, the procedure `flatten` intuitively “flattens” the two levels (i.e., communication and policy level) of specification into one. That is, a sequence of events (i.e. guarded sends, and receives followed by updates) is translated into a sequence of send and receive actions, as follows:

\[
\begin{align*}
\{a_1, \ldots, a_\ell\} \triangleright \text{snd}(m) & \text{ maps to } \text{rcv}(\zeta(a_1), \cdots, \zeta(a_\ell)) \cdot \text{snd}(m) \\
\text{rcv}(m) \triangleright \{a_1, \ldots, a_\ell\} & \text{ maps to } \text{rcv}(m) \cdot \text{snd}(\zeta(a_1), \cdots, \zeta(a_\ell))
\end{align*}
\]

The `flatten`ing step therefore reduces policy level constructs (that is, guards and updates) to communication events, as is common in security protocol specifi-
cations. After flattening, the procedure constraint_sequence generates a constraint sequence for the interleaving (as it is done in [50]). The resulting constraint sequence is then fed to the procedure MSReduce.

**Example 12.** Consider a file server process fs, with policy $P = \{TA, TD\} \cup P^1$, where $P^1$ is the type-1 theory of Example 8. A typical event of the main thread of the fs process is \{can_read($A$, $F$)\}, $\triangleright$ snd(\{F\}_pk($A$)), where $A$ and $F$ denote, respectively, a client of the process and a file stored on fs. For this event the expand procedure returns $\{user(A), public(F)\} \lor \{admin(A), public(F)\} \lor \{admin(A), classified(F)\} \lor \{can_read($A$, $F$)\}, $\triangleright$ snd(\{F\}_pk($A$)); see Example 9.

The interleavings procedure then creates a branch for each $g \in v$ the resulting guard, while interleaving this event with other events of the CAP. For each of the branches the flatten procedure maps the events into communication actions. For example, the guarded send $\{user(A), public(F)\}, $\triangleright$ snd(\{F\}_pk($A$)) maps to $rcv(user(A), public(F)) \cdot snd(\{F\}_pk($A$))$.

The following theorem states that Algorithm 11 is terminating on decision problems for CAPs in $DC_1$; moreover the algorithm is correct (i.e. sound and complete) with respect to the semantics of CAPs.

**Theorem 21.** Given an instance of the decision problem REACH(cap, $a : f$), with cap being a CAP in $DC_1$, Algorithm 11 terminates, and returns a trace if and only if $\text{REACH(cap, } a : f \text{)} = \top$.

**Proof.** Termination of Algorithm 11 is immediate, as procedure interleavings produces a finite number of (symbolic) interleavings, and all the functions applied to interleavings, in particular the constraint reduction procedure MSReduce (cf. [50]) and expand (due to Lemma 8), terminate in finitely many steps.

We now show the correctness of Algorithm 11. Let cap be a CAP in $DC_1$. We say that an interleaving $\iota$ (as formally defined in Section 5.1) is realizable in cap iff there exists a grounding substitution $\sigma$, such that $\iota \sigma$ is a trace of cap (cf. Section 2.2). Given an interleaving $\iota$ of cap and a sequence of attacker constraints $C$

1. Algorithm 11 presents here a fundamental difference with the constraint solving algorithm of [35]. In [35] the subroutine flatten translates events of CAPs (that is, guarded send events and receive events followed by updates) into sequences of annotated send and receive actions. The annotations indicate the knowledge set with which the communication is carried out: A for network communications through the attacker, or $\pi$ for the interaction with the policy of a process $\pi \in \{\pi_1, \ldots, \pi_\ell\}$. The translation is defined as follows:

   \begin{align*}
   & (\{a_1, \ldots, a_\ell\} \triangleright \text{snd}(m)) \text{ maps to } \text{rcv}^\pi(\zeta(a_1), \ldots, \zeta(a_\ell)) \cdot \text{snd}^A(m) \\
   & (\text{rcv}(m) \triangleright \{a_1, \ldots, a_\ell\}) \text{ maps to } \text{rcv}^A(m) \cdot \text{snd}(\zeta(a_1), \ldots, \zeta(a_\ell))
   \end{align*}

   The annotated send and receive actions are used, in [35], in the reduction procedure MSReduce$,^\pi$, which is a syntactic modification of MSReduce. The modification consists in allocating one term set for each process in $\{\pi_1, \ldots, \pi_\ell\}$, and one set for the attacker. This is in contrast to MSReduce where only one term set, denoting the attacker knowledge, is considered.
we define their correspondence inductively: if \( \iota = z_0 \), then the empty sequence, i.e. \( C = \text{nil} \), corresponds to \( \iota \). Now, let \( \iota = z_0 e_1 \cdots z_n e z \), with \( n \geq 0 \), and \( C' \) be a sequence of constraints that corresponds to \( z_0 \cdots z_n \). If \( e = g \triangleright \text{snd}(m) \), then any \( C = C' \cdot \xi(\Omega_{\pi \sigma}^z) \models A \xi(g_i) \) corresponds to \( \iota \), where \( \pi \) is the process that performs \( e \), \( g_i \in \mathcal{E}(g) \) is calculated with respect to the policy of \( \pi \), and \( \Omega_{\pi \sigma}^z \) refers to the knowledge of \( \pi \) at symbolic configuration \( z_n \). The correspondence between event \( e \) and constraint \( \xi(\Omega_{\pi \sigma}^z) \models A \xi(g_i) \) hinges upon Theorem 20, which tells us that the encoding function \( \xi \) is such that \( \xi(G) \models A \xi(g_i) \) for at least one \( g_i \in \mathcal{E}(g) \) if \( G \models P \), given any set of ground infons \( G \). If \( e = \text{rcv}(m) \triangleright u \), then \( C = C' \cdot \Omega_{\pi \sigma}^z \models A \cdot m \) corresponds to \( \iota \), where \( \Omega_{\pi \sigma}^z \) refers to the attacker knowledge at symbolic configuration \( z_n \).

From the definition above, clearly there are finitely many constraint sequences (created due to the \( \lor \) operator in guards) corresponding to any interleaving \( \iota \). We write \( \hat{\iota} \) to denote the finite set of constraint sequences corresponding to \( \iota \).

The proof of Theorem 21 proceeds by showing that for each interleaving \( \iota \) of \( \text{cap} \), \( \iota \) is realizable in \( \text{cap} \) if and only if a constraint sequence corresponding to \( \iota \) is satisfiable in Algorithm 11. The proof is split into two directions.

⇒ Assume \( \text{Reach}(\text{cap}, a, f) = T \). Then, there exists a realizable interleaving \( \iota = z_0 e_1 \cdots z_n e z \), and a non-empty set of substitutions \( \hat{\iota} = \{ \sigma_1, \sigma_2, \ldots \} \) where \( \iota \sigma_j \) is a trace of \( \text{cap} \), for any \( \sigma_j \in \hat{\iota} \). As a consequence of Theorem 20, there exists at least one constraint sequence \( C \in \hat{\iota} \), which is satisfiable by any substitution \( \sigma_j \in \hat{\iota} \); since Algorithm 11 considers for all interleavings all the corresponding constraint sequences, also \( C \) is considered. It now remains to show that \( \text{MSReduce} \) returns a witness trace for \( C \). This holds due to Lemma 9, below.

⇐ Let \( C \) be a constraint sequence generated in Algorithm 11. Then, \( C \in \hat{\iota} \), for some interleaving \( \iota \). Suppose \( \text{MSReduce} \) returns a witness trace, showing that \( C \) is satisfiable under (non-ground) substitution \( \rho \) (see Lemma 9 on applicability of \( \text{MSReduce} \), below). By Theorem 20 and the fact that solving constraint \( T \models A \cdot m \) implies \( T \models A \cdot m \), it follows that \( \iota \sigma \) is a trace of \( \text{cap} \), for any ground substitution \( \sigma \) that refines \( \rho \). That is, \( \text{Reach}(\text{cap}, a, f) = T \).

This completes the proof.

\[ \square \]

**Applicability of MSReduce**

In the following we give an informal explanation of why the constraint reduction system of Millen and Shmatikov can be applied to the constraint sequences generated from \text{CAP} in \text{DC1}.

Any constraint sequence \( C \) generated from \text{CAP} in \text{DC1} can be partitioned into two subsets \( C_A \) and \( C_P \). These subsets respectively refer to \text{attacker constraints} and \text{policy constraints}. Intuitively, for each interleaving, constraints in \( C_A \)
correspond to messages that must be generated by the attacker so that the interleaving is realizable, and constraints in $C_P$ correspond to guards that need to be evaluated to true in honest processes so that the interleaving is realizable. Observe that constraints in $C_A$ are always syntactically distinguishable from constraints in $C_P$.

We focus first on the set of policy constraints $C_P$. We notice that for any constraint of the form $K \vdash^A q$ in $C_P$, with $V$ being a variable appearing in $K$, there exists a constraint $T \vdash^A m$ in $C_A$, where $V$ is a subterm of $m$. This is because all variables in our specifications are originally instantiated at a receive event in an honest process. The key idea in using Millen-Shmatikov’s reduction procedure for CAPs in DC$_1$ is that all the variables appearing in policy constraints are wrapped with infon constructors (which are seen as uninterpreted function constructors by the reduction procedure). Hence, the only applicable reduction rule on a policy constraint $K \vdash^A q$ is the unification rule as far as $q$ is an infon. Therefore, the policy constraints are ultimately either removed from $C$ as they can be solved using unification, or they are unsatisfiable as the infon constructors are not available to the attacker. Recall that due to our expansion procedure (which is a “backward proof search”) the set of premises from which infons can be inferred have already been closed under the policy rules, before the constraint reduction procedure starts. Furthermore, Theorem 20 and Lemma 9 ensure the soundness and completeness of the solutions found by the Millen-Shmatikov reduction system $\text{MSReduce}$ for constraints in $C_P$.

Finally, Lemma 9 shows also that $\text{MSReduce}$ is readily applicable to the elements of the set $C_A$, intuitively because terms obtained by encoding infons of processes in cap do not interfere with the attacker deduction on message terms; more precisely, spurious terms can always be syntactically distinguished from valid message terms and consequently discarded.

Let $G$ be a finite set of ground terms. We write $\lceil G \rceil$ to denote the closure of $G$ with respect to the DY attacker theory; i.e. the set $\{g \mid G \vdash^A g\}$. To ease the presentation of Lemma 9 below we extend the mapping function $\zeta$ of Section 6.2.1 to message terms; for any term $m \in T_{\text{msg}}\Sigma(V)$, $\zeta(m) = m$.

Let $\pi$ be a process of cap. Recall that after application of the procedure $\text{rename\_infons}$ to cap all infons appearing in $\pi$ belong to $\text{infon}^\pi$. Given a finite set of ground terms $G$, we define the subset of $G$ visible to $\pi$, denoted $\langle G \rangle^\pi$, as the set $\{g \mid g \in G, g \in \zeta(\text{infon}^\pi)\}$; that is, the set of terms in $G$ that correspond to the encoding of some infon in $\text{infon}^\pi$. Similarly, the closure of $G$ visible to the attacker $A$ is the set $\langle G \rangle^A = \{g \mid g \in G, g \in T_{\Sigma(V)}^{\text{msg}}\}$.

The following lemma states that, given a set of terms $G$ that are either message terms or encodings of infons of the processes in cap, the set of terms in the closure of $G$ (with respect to the DY attacker theory) that are visible to $\pi$, for $\pi \in \{\pi_1, \cdots, \pi_n, A\}$, depends exclusively on $\langle G \rangle^\pi$. Observe that whether a term is visible to a process $\pi$, or not, can always be syntactically checked.

\textbf{Lemma 9.} Let $\text{cap} = ((S, \Sigma, V), \pi_1, \cdots, \pi_n, A)$ be a CAP belonging to DC$_1$, and
\( G \subseteq \zeta(\mathcal{T}_{\Sigma_{(\emptyset)}}^{msg} \cup \text{infon}_{\pi_1} \cup \cdots \cup \text{infon}_{\pi_\ell}). \) For any \( \pi \in \{ \pi_1, \ldots, \pi_n, \emptyset \} \)

\[ \langle \lceil G \rceil \rangle^\pi = \langle \lceil \langle G \rangle \rangle^\pi \]

**Proof.** We start with the inclusion \( \langle \lceil \langle G \rangle \rangle^\pi \subseteq \langle \lceil G \rceil \rangle^\pi \). By definition of \( \langle G \rangle^\pi \), \( \langle G \rangle^\pi \subseteq G \). The inclusion then follows trivially by monotonicity of the closure for any ground deduction relation, and by monotonicity of the visible subset.

We focus now on the inclusion \( \langle \lceil G \rceil \rangle^\pi \subseteq \langle \lceil \langle G \rangle \rangle^\pi \). Observe that, by definition of the encoding \( \zeta \), the encoding of a ground infon \( i \in \text{infon}^\pi \) always contains a subterm that belongs to \( \text{infon}^\pi \); in particular, an infon \( i = \text{said}(a,x) \) is mapped to the ciphertext \( \zeta(i) = \{\zeta(x), \text{sig}(\alpha, \zeta(x))\} \zeta(\text{E}(\text{tdOn}(a,x))) \), whose content and key both contain subterms in \( \text{infon}^\pi \). On the other hand, message terms and terms obtained by encoding ground infons in \( \text{infon}^\pi \) (for \( \pi' \in \{ \pi_1, \ldots, \pi_n \}, \pi' \neq \pi \)) do not contain subterms that are in \( \text{infon}^\pi \).

Assume now, by contradiction, that there exists \( g \in \langle \lceil G \rceil \rangle^\pi \) such that \( g \notin \langle \lceil \langle G \rangle \rangle^\pi \); that is, \( G \vdash^A g \), and for all proofs one of the leaves is in \( G' = G \setminus \langle G \rangle^\pi \).

If \( g = \zeta(i(x_1, \ldots, x_n)) \), where \( i \in \Sigma_{\text{infon}}^\pi \) of arity \( n \) and \( i \neq \text{said} \), \( g \) can not be constructed with the composition rules of DY. Furthermore, \( g \) can not be obtained by decomposition: terms in \( G' \) do not contain any term in \( \text{infon}^\pi \), and they can not be used as keys for ciphertexts in \( \zeta(\text{infon}^\pi) \).

If \( g = \zeta(\text{said}(a,x)) \) then \( g = \{\zeta(x), \text{sig}(\alpha, \zeta(x))\} \zeta(\text{E}(\text{tdOn}(a,x))) \). Therefore \( g \) can be obtained using terms in \( G' \) neither by composition nor by decomposition, as they do not contain subterms in \( \text{infon}^\pi \). In particular we remark that \( \alpha \) can not occur in \( G' \) as it is a term freshly introduced by the mapping function \( \zeta \) and can not be extrapolated from a signature.

\( \square \)
Chapter 7

Conclusion

We have presented a language for formal specification of communicating authorization policies. The language allows us to specify communication level events, policy level decisions, and the interface between the two. Guards and updates are powerful means for modeling complex (possibly non-deterministic) authorization requirements and state change. In particular, negative updates allow to express revocation of rights in a natural way. We have defined a generic reachability problem in the framework. The reachability problem subsumes the secrecy problem for security protocols [31] and the safety problem for authorization policies [43].

We have shown that the reachability problem is decidable for a fragment of systems specified in the language, called DC. The decidable fragment is of immediate practical relevance, as demonstrated through examples. In DC, the policies of the participant belong to a fragment of Horn theory, called AL. The fragment is expressive and, in contrast to other authorization languages like [3, 27], allows for infinite minimal models. The definition of the AL fragment hinges upon the notion of anchors, which intuitively are extensional facts that yield a finite number of intensional facts. We intend to investigate a refinement of the notion of anchors that would allow to define a larger fragment of policies than AL while retaining decidability. We have given a decision algorithm for the reachability problem in specifications in DC. The algorithm builds upon, and substantially extends, the existing constraint reduction systems for analyzing security protocols; in particular, a novel proof search procedure for policies in AL is employed, and novel techniques for handling (symbolically) negative queries and retracted policy statements.

We have given a different proof technique for another decidable fragment of communicating authorization policies, called DC1, strict subset of the DC fragment. The proof technique is based on encoding the derivation of policy statements with respect to the policy of a participant into message inference trees induced by the Dolev-Yao deduction rules. The encoding shows that the reachability problem for communicating authorization policies (in DC1) is reducible to the secrecy problem in security protocols, and allows us to build upon existing tools which have been originally developed for verifying security protocols. e.g. [50].
Deciding reachability in communicating authorization policies is an NP-hard problem, because it subsumes the reachability problem for security protocols (which is an NP-complete problem [54]). More precise estimates of computational complexity of the algorithms are considered as future work. The efficiency of the decision algorithms has so far not been a major concern in our study. We intend to investigate more efficient (in asymptotic terms) decision algorithms for this fragment. In particular, partial order reduction techniques to reduce the number of interleavings and more efficient proof search techniques would be effective in this respect.
Bibliography


[33] Eran Hammer et al. The OAuth 2.0 authorization framework, 2012. IETF.


