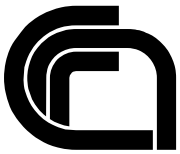

Consiglio Nazionale delle Ricerche

Analysis of Data Sharing Agreements
Automated verification and WS implementation

M. Colombo, F. Martinelli, I. Matteucci, M. Petrocchi

IIT TR-14/2010

Technical report

Marzo 2010

Istituto di Informatica e Telematica

Analysis of Data Sharing Agreements*

Automated verification and WS implementation

Maurizio Colombo, Fabio Martinelli, Ilaria Matteucci, and Marinella Petrocchi

Istituto di Informatica e Telematica, CNR, Pisa, Italy
Contact author: marinella.petrocchi@iit.cnr.it

Abstract. A Data Sharing Agreement (DSA) is an agreement among contracting parties regulating how they share data. DSA are usually subject to a lifecycle consisting (at least) of the following phases: definition, enforcement, and disposal. In particular, during the definition phase, the parties negotiate the respective authorizations on data covered by the agreement. This phase may be iterative: authoring of the DSA is followed by analysis of its content in order to identify possible conflicts or incompatibilities among authorizations clauses, before enforce them. In this paper, we concentrate on DSA formal verification by proposing a formal framework for the automated analysis of DSA. The proposed mechanism is built on a process algebra formalism dealing with contextual data, encoded into the executable specification language Maude, based on Rewriting Logic. The effectiveness of the analysis is shown through a sensitive data sharing test bed. Furthermore, we present an implementation of the analyser exposed as a Web Service built on top of Maude. The Web Service technology allows the modularity of the whole architecture with respect to the particular tool considered for the analysis.

1 Introduction

Data sharing is a critical aspect of every business, government, and social organization. On the one hand, data exchange is vital for a successful organization process. On the other hand, confidentiality and privacy requirements demand that only authorized people should be granted access to such data.

A standard trend is to address the problem of data sharing by adopting authorization criteria based upon the employee position within the organization. Also, secure data exchange procedures are mainly paper-based, and their management is often driven by non-automated mechanisms. Furthermore, when two or more organizations collaborate, information flow policies across their boundaries normally state either "to not share the data" at all, or "to not control the information flow" at all.

Usually, organizations adopt so called Data Sharing Agreements (DSA), formal legal agreements among two or more parties regulating how they share data.

* This work is partly supported by the EU project FP7-214859 Consequence: *Context-Aware Data-Centric Information Sharing*.

The core of DSA consists of a set of clauses, which main purpose is exactly that of defining what parties are allowed or required to do with respect to data covered by the agreement.

From the study of data sharing agreements currently used by various organizations, *e.g.,* [17, 20, 21, 26], we derived a general structure for a DSA, whose main components are the following (see also [8, 15] for detailed information):

Title gives a title to the DSA.
Parties defines the parties making the agreement.
Period specifies the validity period.
Data lists the data covered by the DSA.
Authorizations defines authorizations covered by the DSA.
Date and Signatures contains the date and (digital) signatures of the *Parties*.

Note that DSA are usually subject to a lifecycle consisting of the following main phases: definition, enforcement, disposal. During the definition phase, the parties negotiate the respective authorizations on data covered by the agreement. During the enforcement phase, the DSA clauses are enacted by an appropriate infrastructure ensuring that data exchange among parties comply with the DSA clauses. Finally, a DSA may be disposed of when its validity expires, or when one of the contracting parties decides to recede from it.

The FP7 European Project Consequence [28] accepted the challenge of securely managing critical data sharing and crucial information exchange within and across organizational boundaries. The project aims at delivering a "data-centric information protection framework based on data sharing agreements". Precise goals are to define a generic, context-aware, secure architecture to enable dynamic management policies based on DSA that ensure protection of data-centric information.

Our interest in the project is mainly on the DSA definition phase. We observe that the definition phase is iterative: authoring of the DSA is followed by analysis of its content in order to identify possible conflicts or incompatibilities among authorizations clauses. This process is iterated until all conflicts are solved, and parties have reached agreement on the content of the DSA.

While the authoring phase needs some controlled natural language assuring usability and user-friendliness to adoption of DSA, the analysis phase quests a more formal substrate to enable automatic verification.

In this work, we concentrate on the DSA analysis phase, by providing a formal framework for the verification of DSA with respect to some properties that it should satisfy. For example, before passing the DSA to the enforcement phase, one would like to be assured on which subjects have the rights to perform which security actions on which set of data (*e.g.,* write/read/execute certain files).

We consider a high level description of the agreement, that we specify by means of the process algebra POLPA [14], against a set of properties, expressed as a set of queries. As execution environment for checking if the DSA/POLPA specification satisfies those properties we choose the rewriting system engine Maude [4]. To this aim, an encoding of POLPA into the Maude programming

language will be presented. Analysis samples will be given by considering a case study involving the protection of scientific research data.

We also present an implementation of the overall architecture, based on Web Service technology. The choice of using this technology was driven by guarantees of a certain modularity and flexibility with respect to the particular tool chosen for the analysis.

The structure of the paper is as follows. The next section recalls two high-level languages for the description of DSA, namely CNL4DSA and POLPA. Section 3 presents our executable specification of POLPA into Maude. In section 4, we analyse some authorization DSA clauses related to a sensitive data sharing test bed. Then, we show the implementation of the DSA analysis architecture in section 5. Finally, we conclude the paper by listing some related work, and by giving final remarks.

2 High-level Languages for DSA Specification

This section recalls two languages for describing DSA. The first language is called CNL4DSA [15], and it is mainly intended for authoring a DSA in a controlled manner. The second language is the POLicy Process Algebra (POLPA), mainly due with the DSA analysis phase. Even if this paper is more related to POLPA, we decide to briefly recall CNL4DSA to show the existence of a coherent *trait d'union* between the authoring phase and the analysis phase. Indeed, [15] shows a mapping from CNL4DSA to POLPA.

2.1 CNL4DSA

In [15], a Controlled Natural Language for Data Sharing Agreements, namely CNL4DSA, has been proposed. The language aims at ensuring (i) simplicity for end-users, (ii) extensibility for coping with the evolving Web environment, and (iii) safe translation to formal specifications allowing for automated verification of the authorizations clauses of a DSA.

Here, we briefly recall the syntax of the language. The interested reader can find more details, including its operational semantics, in [15].

The core of CNL4DSA is the notion of *fragment*, a tuple $f = \langle s, a, o \rangle$ where s is the subject, a is the action, o is the object. The fragment expresses that "the subject s performs the action a on the object o", *e.g.,* "Bob reads Document1". More complex expressions are generated by combining fragments. Such expressions are *composite authorization fragments*, and we denote them as F[1].

Usually, fragments are evaluated within a specific *context*, *i.e.,* a predicate c, evaluating to a boolean value, that usually characterizes contextual factors, such as time and location (*e.g.,* "more than 1 year ago" or "inside the facility").

[1] CNL4DSA has been developed to express not only authorizations, but also obligations. However, we do not recall *composite obligations fragments* here, because they are out of scope for this paper.

In order to describe complex agreements, contexts need to be composable. A *composite context* C is defined inductively as follows:

$$C \ := c \mid C \ and \ C \mid C \ or \ C \mid not \ c$$

Thus, the syntax of a composite fragment is inductively defined as follows:

$$F \ := nil \mid can \ f \mid F; F \mid if \ C \ then \ F \mid after \ f \ then \ F \mid (F)$$

The intuition for the composite authorization fragment is the following:

- nil can do nothing.
- $can \ \ f$ is the atomic authorization fragment. Its informal meaning is *the subject s can perform the action a on the object o*. $can \ \ f$ expresses that f is allowed.
- $F; F$ is a list of composite authorization fragments. The list constitutes the authorization section of the considered DSA.
- $if \ C \ then \ F$ expresses the logical implication between a composite context C and a composite authorization fragment: if C holds, then F is permitted.
- $after \ f \ then \ F$ is the temporal sequence of fragments. Informally, after f has happened, then the composite authorization fragment F is permitted.

2.2 POLPA

CNL4DSA has been proposed with an eye for formal verification. Its semi-formal structure makes it suitable for a simple, intuitive and easy mapping to high-level formal policy languages. Here, we consider the POLicy Process Algebra POLPA [14, 1], built on top of CSP [12] and enriched with a predicate construct expressing that a transition happens only if a predicate is true. This language proved to be useful for expressing usage control policies, in particular for web applications [1] and for GRID systems [14]. We recall syntax and operational semantics of the language. The reader is referred to [14] for more details. The syntax of POLPA is as follows:

$$P ::= stop \mid \alpha(\boldsymbol{x}).P \mid p(\boldsymbol{x}).P \mid P[fun] \mid P_1 or P_2 \mid P_1 | \{Alfa\} | P_2 \mid Z$$

with processes P, P_1, and $P_2 \in \mathcal{P}$, parameterized actions $\alpha(\boldsymbol{x}) \in Act$, and $p(\boldsymbol{x}) \in Pr$ parameterized predicates evaluating to a boolean value. Parameters range over $x, x_1, \ldots, x_n$ and belong to set $\mathcal{X}$. The special action τ is the silent action denoting the internal behaviour of a process. τ has no parameters.

The informal semantics of the language is the following:

- *stop* is the process that allows nothing;
- $\alpha(\boldsymbol{x}).P$ is the process that performs action $\alpha(\boldsymbol{x})$ and then behaves as P;
- $p(\boldsymbol{x}).P$ is the process that behaves as P when the predicate $p(\boldsymbol{x})$ is true; otherwise, it gets stuck;
- $P[fun]$ is the process that behaves as P but with the parameters $\boldsymbol{x}$ substituted according to fun. $fun : \mathcal{X} \rightarrow \mathcal{X}$ is a finite substitution function;

Let $\xrightarrow{\alpha(\boldsymbol{x})}$ be the smallest subset of $\mathcal{P} \times Act \times \mathcal{P}$, closed under the following rules:

$$(prefix)\frac{}{\alpha(\boldsymbol{x}).P \xrightarrow{\alpha(\boldsymbol{x})} P} \qquad (pred)\frac{}{p(\boldsymbol{x}).P \xrightarrow{\tau} P}\ p(\boldsymbol{x}) = true$$

$$(or)\frac{P \xrightarrow{\alpha(\boldsymbol{x})} P_1}{P + Q \xrightarrow{\alpha(\boldsymbol{x})} P_1} \qquad (comp_1)\frac{P \xrightarrow{\alpha(\boldsymbol{x})} P_1}{P|\{Alfa\}|Q \xrightarrow{\alpha(\boldsymbol{x})} P_1|\{Alfa\}|Q}\alpha(\boldsymbol{x}) \notin Alfa$$

$$(const)\frac{P \xrightarrow{\alpha(\boldsymbol{x})} P_1\ Z \doteq P}{Z \xrightarrow{\alpha(\boldsymbol{x})} P_1} \qquad (comp_2)\frac{P \xrightarrow{\alpha(\boldsymbol{x})} P_1 \quad Q \xrightarrow{\alpha(\boldsymbol{x})} Q_1}{P|\{Alfa\}|Q \xrightarrow{\alpha(\boldsymbol{x})} P_1|\{Alfa\}|Q_1}\alpha(\boldsymbol{x}) \in Alfa$$

$$(subs_1)\frac{P \xrightarrow{\alpha(\boldsymbol{x})} P_1}{P[fun] \xrightarrow{\alpha(fun(\boldsymbol{x}))} P_1[fun]} \qquad (subs_2)\frac{}{P[fun] \xrightarrow{\tau} P_1[fun]}\ p(fun(\boldsymbol{x})) = true$$

Fig. 1. POLPA operational semantics rules

- P_1 or P_2 is the choice between the two processes;
- $P_1\ |\{Alfa\}|\ P_2$ is the composition operator and it represents concurrent activity requiring synchronization between P_1 and P_2. In particular, any action belonging to the set of actions $\{Alfa\}$ can only occur when both the processes perform that action. When P_1 and P_2 engage in actions not belonging to $\{Alfa\}$, this term represents independent concurrent activity and the events from both the processes are arbitrarily interleaved in time;
- Z is the constant process. We assume that there is a specification $Z \doteq P$ and Z behaves as P.

This intuitive explanation is made precise by the operational semantics shown in Fig. 1, that defines a Labelled Transition System (LTS) for POLPA processes. In particular, the LTS is a structure ($\mathcal{P}$, Act, $\xrightarrow{\alpha(\boldsymbol{x})}$), where $\mathcal{P}$ is the set of POLPA processes, Act is the set of parameterised actions, and $\xrightarrow{\alpha(\boldsymbol{x})}$ is a ternary relation, *i.e.,* a subset of $\mathcal{P}$ x Act x $\mathcal{P}$, representing a transition relation between processes through an action. Notation $P_1 \xrightarrow{\alpha(\boldsymbol{x})} P_2$, with $P_1, P_2 \in \mathcal{P}$ and $\alpha(\boldsymbol{x}) \in Act$ means that it is admissible that the implementation of P_1 performs $\alpha(\boldsymbol{x})$ and then behaves like P_2. As usual, rules are expressed in terms of a set of premises, possibly empty (above the line) and a conclusion (below the line). The operators for choice and composition are assumed commutative and associative, and the symmetric cases are not shown in the figure.

3 An Executable Specification of POLPA into Maude

Maude is "a programming language that models (distributed) systems and the actions within those systems" [4]. Systems are specified by defining algebraic

data types axiomatizing systems states, and rewrite rules axiomatizing systems local transitions. The power of Maude is that it is executable and counts with a toolkit that allows formal reasoning of the specifications produced.

Here, the goal is to exploit the Maude engine to implement an executable specification of the POLPA language, in its turn used to specify DSA as distributed systems. Then, Maude built-in commands and/or ad hoc strategies can be used to search for allowed traces of the specified DSA. These traces represent the actions that are authorised by the DSA.

Our encoding is inspired by the work of Verdejo and Martí-Oliet [27], that implements the CCS operational semantics [16] in Maude 2.0, by seeing transitions as rewrites and inference rules as conditional rewrite rules.

First, we define the POLPA syntax in Maude. As in [27], all the non constant operators are defined as *frozen*, meaning that the use of rewrite rules in the evaluation of the arguments is forbidden. The reason is detailed below.

```
fmod POLPA-SYNTAX is

  inc PRED-EVAL .
  sorts  ProcessId Process .
  sort ActSet .
  subsort Act < ActSet .
  subsorts Qid < ProcessId < Process .

  op tau : -> Act .
  op stop : -> Process .

---Polpa operators
  op _._ : Act Process -> Process [frozen prec 25] .                 *** prefix
  op _or_ : Process Process -> Process [frozen assoc comm prec 35] . *** choice
  op _|_|_ : Process ActSet Process -> Process [frozen prec 30] .    *** composition
  op _@_ : Pred Process -> Process [frozen prec 25] .                *** predicate
  op _[_/_] : Process Subj Subj -> Process [frozen prec 20] .        *** substitution1
  op _[_/_] : Process Obj Obj -> Process   [frozen prec 20] .        *** substitution2

endfm
```

Process definitions are represented by means of *contexts*. The contexts, as well as the operations to work with them, are defined in the module POLPA-CONTEXT (not shown here), following [27]. In practice, a constant *context* is defined, through which the definition of the process identifiers of the POLPA specifications is kept.

The module POLPA implements the operational semantics of the language. The transitions are seen as rewrite rules. The value $\{A\}P$ of sort $ActProcess$ indicates that process P has performed action A.

```
mod POLPA is
  pr POLPA-CONTEXT .
  sort ActProcess .
  subsort Process < ActProcess .
  op {_}_ : Act ActProcess -> ActProcess [frozen] .
  vars A B : Act .
  vars L M : Action .
  vars S S' : ActSet .
  vars PR PR' : Pred .
  vars P P' Q Q' R : Process .
```

```
  var X : ProcessId .
  var AP : ActProcess .
  vars Y Y' : Subj .
  vars O O' : Obj .
  vars AS AS' : Assertion .
  vars RO RO' : Role .
  vars T T' : Typ .
  var D : Date .
  var LOC : Country .
  var SET : Set .

  *** Prefix
  rl [pref] : A . P => {A}P .

  *** Choice
  crl [sum] : P or Q => {A}P' if P => {A}P' .

  *** Composition (a la CSP)
  crl  P | S | Q  => {A}(P' | S |  Q )  if P => {A}P' /\ not (A incl S) .
  crl  P | S | Q  => {A}(P | S |  Q' )  if Q => {A}Q' /\ not (A incl S) .
  crl  P | S | Q  => {A}(P' | S |  Q') if P => {A}P' /\ Q => {A}Q' /\ A incl S  .

  *** Predicate  (go ahead if predicate == true)
  crl  PR @ P => P if  eval(PR) .

  *** Substitution
  ---subject/object parameter in actions:
  rl [rel] : ((L(Y,O)) . P) [Y' / Y][O' / O] => {L(Y',O')} P [Y' / Y][O' / O] .
  ---subject parameter in role predicates:
  crl [rel] : ((AS(Y,RO)) @ P )  [Y' / Y][O' / O] => P[Y' / Y][O' / O] if eval(AS(Y',RO)) .
  ---object in data predicates (type):
  crl [rel] : ((AS(O,T)) @ P )   [Y' / Y][O' / O] => P[Y' / Y][O' / O] if eval(AS(O',T)) .
  ---object in embargo-end-period predicates (type):
  crl [rel] : ((AS(O,D)) @ P )   [Y' / Y][O' / O] => P[Y' / Y][O' / O] if eval(AS(O',D)) .
  ---subject parameter in location predicates:
  crl [rel] : ((AS(Y,LOC)) @ P ) [Y' / Y][O' / O] => P[Y' / Y][O' / O] if eval(AS(Y',LOC)) .
  ...
  ...
  *** Definition
  crl [def] : X => {A}P if (X definedIn context) /\ def(X,context) => {A}P .

  *** reflexive, transitive closure
  sort TProcess .
  subsort TProcess < ActProcess .
  op !_ : Process -> TProcess [frozen] .
  crl [refl] : ! P => {A}Q  if P => {A}Q .
  crl [tran] : ! P => {A}AP if P => {A}Q /\ ! Q => AP .

endm
```

We briefly comment the content of the POLPA module. POLPA actions and predicates are parameterised. Some of the variables declared at the beginning of the POLPA module represent those parameters.

In particular, the most general parameters one may think of are the subject performing an action and the target object of that action. As an example, action *read* can have as parameters subject *Bob* and object *Document1*. Subjects may be single individuals, either humans or electronic devices, as well as organizations. In the context of DSA, objects are mainly the data whose sharing is regulating by the DSA itself.

Other parameters specifically depend on the DSA one aims at specifying. Generally speaking, they represent contextual data such as time, locations, roles covered by subjects (*e.g.,* Principal Investigator, Co-Investigator, Beam-line Sci-

entist, *etc...*), and objects' categories (*e.g.,* numerical data, image data, investigation metadata, *etc...*).

The substitution rules shown above are specific for dealing with actions parameterised by a subject and an object (the very first rule), and with predicates parameterised by, *e.g.,* subjects and their roles, and objects and their categories.

Operator *incl* returns true if an action is included in a certain set. Appropriate equations for all the operators are defined in the complete specification.

As in [27], we consider only terms that are well-formed, *i.e.,* that can be associated to a sort. Thus, rules cannot be applied unless both the right hand side and the left hand side of the rule are well-formed. The POLPA syntax operators have been defined as *frozen* since, when dealing with recursive processes, there could be an infinite loop in the attempt to apply a rule, since, for example, the built terms are not well formed.

On the other hand, one of our goals is exactly to prove that a process can perform a certain sequence of actions, in other words a *trace*. Thus, as in [27], we will consider a dummy operator ! whenever we will ask if a process can perform a certain trace. Instead, when we will just ask for the one-step successor of a process, we will not use the ! operator.

Once defined the transitions semantics for the POLPA language, one can implement, on top of it, the Hennessy-Milner modal logic for describing capabilities of processes [11, 23]. This allows us to prove in Maude if a POLPA process satisfies a logical formula. Formulas are defined by the following grammar, over some set of actions K:

$$\Psi ::= tt \mid ff \mid \Psi_1 \wedge \Psi_2 \mid \Psi_1 \vee \Psi_2 \mid [K]\Psi \mid \ < K > \Psi$$

A formula: 1) is always *true* or always *false*; 2) can be the conjunction and the disjunction of two formulas; 3) must hold for all the K-derivatives of a process; 4) must hold for some K-derivatives of a process. K-derivatives are the states reachable by a process by performing actions in K.

Formulas represent the logical specification of the DSA properties that we would like to investigate. Some examples will be given in Section 4.

Here, we do not show the modules implementing the Hennessy-Milner logic over POLPA, since they are essentially the same as the ones in [27]. We refer to that paper for the detailed implementation of those modules.

Maude modules PREDICATE and PRED-EVAL define, resp., 1) the collection of sorts for actions, predicates, parameters, *etc...* , and the equationally specifiable operators acting on those sorts (and constants), and 2) ad-hoc defined truth tables for predicates' evaluation. We show some excerpts of these modules.

```
fmod PREDICATE is
*** for the definition of POLPA parameterised actions and predicates
  inc QID .
  inc INT .
  sorts Subj Obj Action Act Assertion Role Typ Date Location Country Set .
  subsort Qid < Action Date Set .
  sorts Pred RolePred DataPred TimePred LocPred .
  subsorts  RolePred DataPred LocPred TimePred < Pred .
--- roles declaration for the sensitive data sharing case study:
  ops PrincipalInvestigator Coinvestigator BeamlineScientist Any : -> Role .
--- data category declaration for the sensitive data sharing case study:
  ops image investigation-metadata numerical : -> Typ .
--- subjects declaration
 op Caroline : -> Subj .
 ...
--- data declaration
 op doc1 : -> Obj .
 op doc2 : -> Obj .
--- country declaration : e.g., UK, Italy, and a blacklisted country
 op Badland : -> Country .
 op italy : -> Country .
 op uk : -> Country .
---parameterised actions declaration:
  op _(_,_) : Action Subj Obj -> Act .
---parameterised predicates declaration:
 op _(_,_) : Assertion Subj Role -> RolePred .
 op _(_,_) : Assertion Subj Country -> LocPred .
 op _(_,_) : Assertion Obj Typ -> DataPred .
 ...
--- predicates terms
 op has-role : -> Assertion .
 op has-data-category : -> Assertion .
 op has-embargo-end-date : -> Assertion .
 op current-time-is-before : -> Assertion .
 op has-location-country : -> Assertion .
 ...
endfm

fmod PRED-EVAL is
  inc PREDICATE .
  op eval : Pred -> Bool .
  ---some variables declaration
  ...
  ...
  ---example of predicates' table of truth:
  eq eval(has-role(Caroline,Coinvestigator)) = true .
  eq eval(has-role(Stephen,PrincipalInvestigator)) = true .
  eq eval(has-role(John,Coinvestigator)) = true .
  eq eval(has-role(Guest,Any)) = true .
  ...
--- principal investigators from blacklisted countries:
  eq eval(has-role(Eve,PrincipalInvestigator)) = true .
  ...
  eq eval(PR) = false [owise] .
endfm
```

4 A Case Study: Sensitive Data Sharing

Here, we consider the protection of sensitive scientific research data across organisational boundaries.

Scientific and technologies councils in Europe and throughout the world operates large facilities, *e.g.,* x-ray, ultraviolet light, neutron and muon sources used as large microscopes to determine the structure of materials for biosciences and physical sciences. Teams from universities and companies come to these facilities to run experiments that produce data. Also, many of the users of these facilities use more than one, in order to exploit the differences between them, so that they can gain as many data as possible. Furthermore, scientists would like to access the data from remote sites, *e.g.,* to analyse them. The management of large data files, the presence of many users, and the wide range of usage applications, make necessary to make agreements between the facility host organization and funding bodies, universities, and companies, to provide them with data in accordance with all the individual data policies. The set of data policies which can apply to the data coming from one experiment can be both large and contradictory, and heavily influenced by contextual factors like, *e.g.,* user role, geographical location, data category, and time.

We consider the following data policies, which represents the *Authorizations* Section of a DSA, expressed in the English natural language[2].

1. Before the end of the embargo period, access to the experimental data is restricted to the principal investigator and co-investigators.
2. After the embargo period, the experimental data may be accessed by all users.
3. Beam-line scientists can access image data related to or produced on their experimental station.
4. Access to numerical data should be denied to users which country is subject to embargo regulations.

One may be interested in answering to questions like the following:

- Action list: which are all the authorised actions in the investigated DSA?
- Existence of one particular action: is action *read(x,y)* authorised?
- Answer to specific authorization-related queries: is it true that subject x is authorised to perform action z on object y, within a certain interval of time, and/or from a certain location?
- Look for temporal sequences of actions: is it true that, after opening object y, subject x can read object y?
- Search for the co-existence of *opposite* actions: is it true that subject x can read/write/print... object y and that, at the same time, subject x cannot read/write/print...object y?

Maude makes possible to answer to those queries by means of its built-in commands. By using command *search*, it is possible either to find all the possible one-step successors of a process, or all its possible successors, or to ask if there is one (or more) way to rewrite a process into a certain sequence of actions. Also,

[2] These data policies have been provided by colleagues from the Science and Technology Facility Council (STFC), our partner in the Consequence project.

exploiting the implementation of modal logic over the POLPA semantics, one can prove if a modal formula, representing a certain query, is satisfied by the Maude representation of the DSA POLPA specification.

In order to show some analysis examples, we consider the following initial configuration. Note that this configuration is completely arbitrary: aiming at describing how the framework works, we fixed some constant values for the subjects (and their roles and locations), for the objects (and their categories), and we gave predefined tables of truth relating, *e.g.,* a subject value to a subject role value, an object value to an object category value (see Maude modules PREDICATE and PRED-EVAL in the previous section).

```
--- subjects declaration     ---data declaration
 op Caroline : -> Subj .      op doc1 : -> Obj .
 op Stephen : -> Subj .       op doc2 : -> Obj .
 op Eve : -> Subj .
```

In particular, we assume that Stephen is a PrincipalInvestigator and Caroline a Co-investigator, both coming from UK. Also, Eve is a PrincipalInvestigator from Badland. Doc1 is an image data, and doc2 is a numerical data (both image and numerical data are experimental data). Finally, we assume that we are in a period of time that is before the end of the embargo period, that is set to 31/12/2010.

DSA authorization clauses are expressed by using variables instead of constant values. The queries will fix the values of the variables, being specific for some subjects and objects.

The Maude representation of the POLPA process specifying the STFC DSA authorizations clauses is kept in the following context (due to space restriction, we give only some excerpts of the specification):

```
eq context = ( 'Statement1 =def ((has-data-category(X0,numerical)) @
      (has-embargo-end-date(X0,'31/12/2010)) @ (current-time-is-before('31/12/2010)) @
      (has-role(X4,PrincipalInvestigator)) @ (has-principal-investigator-uid(X0,'SET))
      @ (has-principal-investigator-uid(X0,'SET)) @ (in(X4,'SET)) @ ('read(X4,X0)) . stop)
      [Stephen / X4] [doc2 / X0] ) &
             ...
             ...
      ( 'Statement4 =def ((has-data-category(X0,image)) @
      (has-embargo-end-date(X0,'31/12/2010)) @ (current-time-is-before('31/12/2010)) @
      (has-role(X4,Coinvestigator)) @ (has-coinvestigator-uid(X0,'SET)) @ (in(X4,'SET)) @
      ('read(X4,X0)) . stop) [Caroline / X4] [doc1 / X0] ) &
             ...
      ( 'Statement9 =def ((has-data-category(X0,numerical)) @
      (not-has-location-country(X4,Badland)) @ @ ('read(X4,X0)) . stop)
      [Eve / X4] [doc2 / X0] ) &
      ( 'Statement10 =def ((has-data-category(X0,numerical)) @
      (has-embargo-end-date(X0,'31/12/2010)) @ (current-time-is-after('31/12/2010))
      @ (has-role(X4,Any)) @ ('read(X4,X0)) . stop) [Caroline / X4] [doc2 / X0] ) &
     ( 'Statement11 =def ((has-data-category(X0,image)) @
     (has-embargo-end-date(X0,'31/12/2010)) @ (current-time-is-after('31/12/2010)) @
     (has-role(X4,Any)) @ ('read(X4,X0)) . stop)
     [Stephen / X4] [doc1 / X0] ) &
     ('DSA  =def ('Statement1 | 0 | ('Statement2 | 0 | ('Statement3 | 0 | ('Statement4 | 0 |
                 ('Statement5 | 0 | ('Statement6 | 0 | ('Statement7 | 0 | ( 'Statement8 | 0 |
                 ( 'Statement9 | 0 | ('Statement10 | eset | 'Statement11 ))))))))))) .
```

The specification consists of eleven sub-processes, representing the authorization clauses of the STFC DSA, plus their parallel composition DSA (the last

process definition). No synchronization over a set of actions is required among the sub-processes constituting the DSA (being 0 an empty set of actions).

We can ask something related to that context. In particular, we would like to know about the subjects' capabilities to perform certain actions on the objects.

Below, we give an example list of queries, formatting according to the Maude metalevel programming representation. A textual form, expressing the meaning of each query, and the expected boolean result for that query are shown in Table 4. Figure 3 in the next section shows the screenshot of executing Maude over the DSA specification kept by the above context and this list of queries.

Textual queries	Expected
Is it true that:	
1) Stephen AND Caroline can read doc2?	true
2) Eve can read doc2?	false
3) after Caroline reads doc2 then Stephen can read doc1?	true

Table 1. Example queries for the sensitive data sharing test bed

```
red ''DSA.Qid |=   ( (<  '_`(_`,_`)[''read.Action, 'Stephen.Subj,
'doc2.Obj] > tt) /\ (< '_`(_`,_`)[''read.Action, 'Caroline.Subj, 'doc2.Obj] > tt)) .

red ''DSA.Qid |=  <  '_`(_`,_`)[''read.Action, 'Eve.Subj, 'doc2.Obj] > tt .

red ''DSA.Qid |= [  '_`(_`,_`)[''read.Action, 'Caroline.Subj, 'doc2.Obj] ]
                 < '_`(_`,_`)[''read.Action, 'Stephen.Subj, 'doc2.Obj] > tt .
```

Note that in the context shown above, we consider only some substitutions of values replacing variables. For example, Statement 1 is evaluated by replacing $X4$ with *Stephen*, and $X0$ with *doc2*. In the actual implementation, presented in the next section, all the possible substitutions of the variables with the values declared in the configuration file are applied.

5 Implementation

In this section, we show how we have implemented the DSA Analysis framework. We choose to adopt the Web Service technology to publish the functionalities of the analysis tool. In this paper, we have considered Maude as the rewrite engine for the analysis. An architecture based on web services will allow us to easily adopt other analysis tools, thus achieving as much modularity as possible.

First, we consider the xsd schema adopted in the Consequence project for representing DSA documents. We refer to [6–8] for details about the Consequence management of the entire DSA lifecycle.

A general xsd schema has been defined to include the DSA sections listed in the introduction of this paper. In particular, Figure 2 shows an XML instance

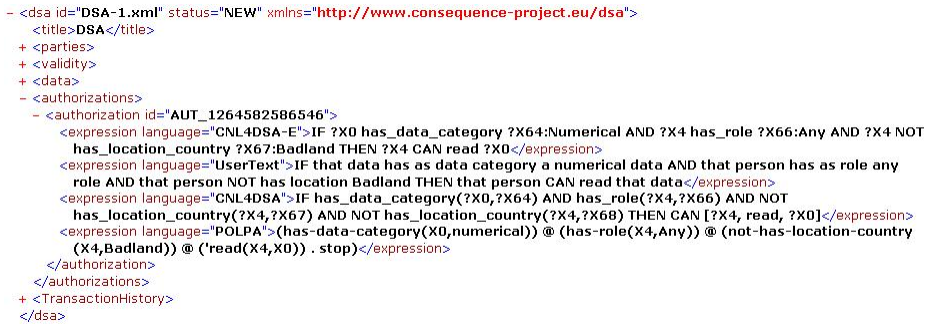

Fig. 2. Instance of DSA with the Authorizations section highlighted

of DSA according to the schema, highlighting the *Authorizations* section (in particular, Statement 9 of the context given in the previous section). The DSA authorization clauses are expressed in all the languages adopted in the definition phase of the DSA lifecycle, *i.e.,* English natural language, CNL4DSA (used in the authoring phase), and POLPA (used in the analysis phase).

The DSA Analysis framework comprises:

1. the WS-Analyzer component which functionalities are exposed through a generic web service interface;
2. the internal analysis tool, *i.e.,* in our case, the Maude engine;
3. a graphical user interface (GUI) that acts as a user front-end for the WS-Analyzer. This GUI is able to manage the analysis's outcome in order to show the results in a more readable way.

A generic client application for the WS-Analyzer invokes the *analyze* method to perform the analysis of the DSA. This method takes as input: 1) the DSA's identifier; 2) the XML representation of the DSA, containing also the POLPA specification of the agreement; and 3) a set of queries.

In our framework, the GUI behaves as a client for the Analyzer and allows the user to load a specific DSA whenever the analysis needs to be performed on it. The analysis is executed through a set of system calls directed to the Maude tool installed on the system and fully wrapped into the web service. The output from Maude is opportunely formatted and returns back to the client. The GUI has been designed in order to make the analysis result understandable for a human reader. Figure 3 shows the graphical interface with the textual queries which related logical formulas are being checked by Maude. The two leftmost columns show, respectively, the expected results and the real results of the analysis. If the real results match the expected results, then the analysis is a success, otherwise, a failure. The results of the Maude computation are also shown in the figure.

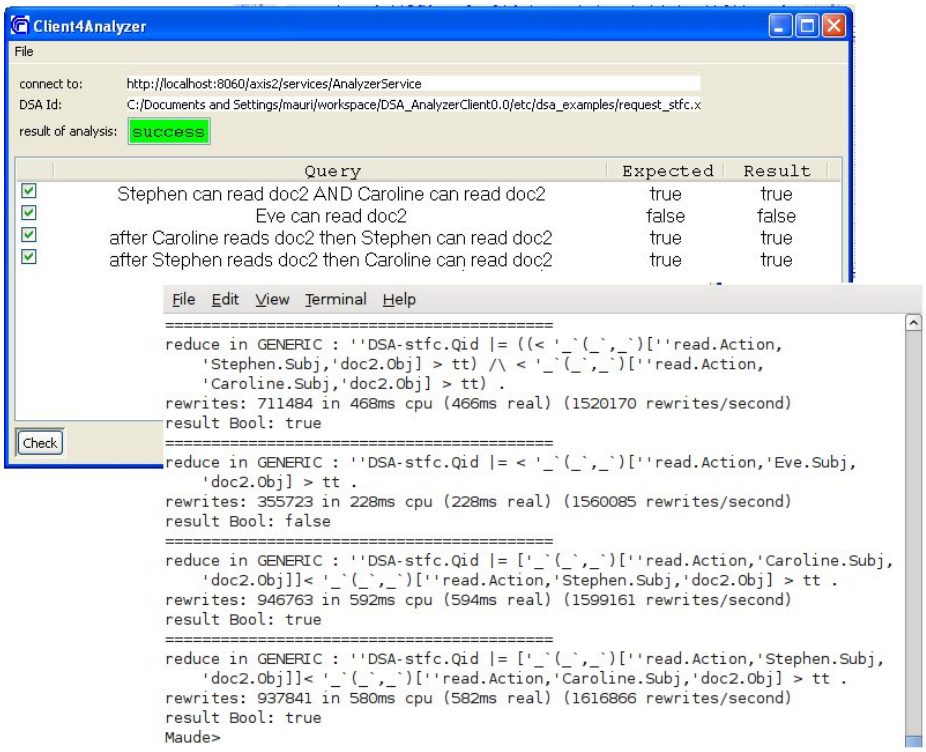

Fig. 3. GUI and Maude computation screenshot

6 Related Work

Our proposal for a DSA analyser becomes part of an information protection framework aiming at defining a scalable architecture to enable secure management of policies. In particular, the architecture incorporates models and tools for policy specification and authoring, *e.g.,* [3, 13, 15], and secure mechanisms for the enforcement of controlled data sharing, *e.g.,* [9, 22].

In the literature, there have been some efforts towards the analysis of DSA. In particular, [24, 25] focus on a specification model based on distributed temporal logic predicates (DTL). A precise formal semantics for that language has not been given. However, it is the authors' opinion that it can be enriched with such a semantics, leading to a variety of automated analysis.

Other alternative approaches are possible for modeling and analysing DSA, including the use of Event-B and the Rodin platform [10]. The Rodin platform provides animation and model-checking toolset, for developing specifications based on the Event-B language. In [2], it was shown that the Event-B language can be used to model obliged events. This could be used in the case of analysing obligations in DSA. Some of the authors are our partners in the Con-

sequence project, and we are currently investigating for setting up an integrated framework that could benefit from both the approaches.

Even if not specifically related with DSA, [5] presents a logic-based policy analysis framework which considers not only authorizations, but also obligations, giving useful diagnostic information.

Finally, there exists generic formal approaches that could *a priori* be exploited for the analysis of some aspects of DSA. As an example, the Klaim family of process calculi [19] provide a high-level model for distributed systems, and, in particular, exploits a capability-based type system for programming and controlling access and usage of resources [18].

7 Conclusions and Future Work

We focused on the development of an executable specification of a process algebra in Maude, to exploit its built-in capabilities for the analysis of DSA authorizations clauses. Our work aims at identifying inconsistencies and possible conflicts among the clauses before their actual enforcement. We presented a Web Service implementation that publish the functionalities of the analysis tool. This approach facilitates the usage of other background analysis engines.

We are currently developing a DSA infrastructure in which the DSA back-end analysis engine is interfaced with a user-friendly DSA authoring tool, which will enable end-users to easily write authorizations clauses in a controlled natural language, and to see possible conflicts detected by the analysers.

8 Acknowledgments

The fourth author would like to thank Alberto Verdejo for his support and help in encoding process algebras into Maude.

References

1. Benjamin Aziz et al. Controlling usage in business process workflows through fine-grained security policies. In *TrustBus*, LNCS 5185. Springer, 2008.
2. Juan Bicarregui, Alvaro Arenas, Benjamin Aziz, Philippe Massonet, and Christophe Ponsard. Towards modelling obligations in Event-B. In *ABZ*, pages 181–194, 2008.
3. Carolyn Brodie et al. The coalition policy management portal for policy authoring, verification, and deployment. In *POLICY*, pages 247–249, 2008.
4. Manuel Clavel et al., editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
5. Robert Craven et al. Expressive policy analysis with enhanced system dynamicity. In *ASIACCS*, 2009.
6. Consequence Deliverable D1.1. First version of Consequence architecture. http://www.consequence-project.eu/Deliverables_Y1/D1.1.pdf, December 2008.

7. Consequence Deliverable D1.2. Second version of Consequence architecture, To be published.
8. Consequence Deliverable D2.1. Methodologies and tools for data sharing agreements infrastructure. http://www.consequence-project.eu/Deliverables_Y1/D2.1.pdf, December 2008.
9. E.Scalavino, G. Russello, R. Ball, V. Gowadia, and Emil C. Lupu. An opportunistic authority evaluation scheme for data security in crisis management scenarios. In *ASIACCS*, 2010.
10. Event-B and the Rodin Platform. www.event-b.org, Last access March 1, 2010.
11. Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *ICALP*, pages 299–309, 1980.
12. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
13. Kaarel Kaljurand. *Attempto Controlled English as a Semantic Web Language*. PhD thesis, in Mathematics and Computer Science, Tartu Univ., 2007.
14. Fabio Martinelli and Paolo Mori. A model for usage control in grid systems. In *GRID-STP07*, 2007.
15. Ilaria Matteucci, Marinella Petrocchi, and Marco Luca Sbodio. CNL4DSA: a controlled natural language for data sharing agreements. In *SAC: Privacy on the Web Track*. ACM, 2010.
16. Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
17. National Collaborative on Workforce and Disability. Sample inter-agency data-sharing agreement. http://www.ncwd-youth.info/assets/guides/assessment/sample_forms/data_share.pdf, Last access, February 26, 2010.
18. Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Programming access control: The klaim experience. In *CONCUR*, pages 48–65, 2000.
19. Rocco De Nicola, Gianluigi Ferrari, and Rosario Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
20. Norfolk County Council. Data Sharing Agreements. www.norfolk.gov.uk/consumption/idcplg?IdcService=SS_GET_PAGE&nodeId=3516, Last access February 26, 2010.
21. Oklahoma Health Care Authority. Interagency agreement. www.ohca.state.ok.us/WorkArea/linkit.aspx?LinkIdentifier=id&ItemID=8131, Last access February 26, 2010.
22. Enrico Scalavino, Vaibhav Gowadia, and Emil C. Lupu. Paes: Policy-based authority evaluation scheme. In *DBSec*, pages 268–282, 2009.
23. Colin Stirling. Modal and temporal logics for processes. In *Logics for concurrency: structure versus automata*, pages 149–237. Springer, 1996.
24. Vipin Swarup et al. A data sharing agreement framework. In *ICISS*, pages 22–36, 2006.
25. Vipin Swarup et al. Specifying data sharing agreements. In *POLICY*, pages 157–162, 2006.
26. US Dept. of Health. NIH Data Sharing Policy and Implementation Guidance. http://grants.nih.gov/grants/policy/data_sharing/data_sharing_guidance.htm, Last access February 26, 2010.
27. Alberto Verdejo and Narciso Martí-Oliet. Implementing CCS in maude 2. *Electr. Notes Theor. Comput. Sci.*, 71, 2002.
28. EU Project Consequence Website. http://www.consequence-project.eu/, Last access: March 1, 2010.