2010 Third International Conference on Advances in Human-Oriented and Personalized Mechanisms, Technologies and Services

Context-Aware Analysis of Data Sharing Agreements

Maurizio Colombo, Fabio Martinelli, Ilaria Matteucci, Marinella Petrocchi
IIT-CNR, Pisa, Italy
{maurizio.colombo, fabio.martinelli, ilaria.matteucci, marinella.petrocchi}@iit.cnr.it

***Abstract*—A Data Sharing Agreement is an agreement among contracting parties regulating how they share data under certain contextual conditions. Upon the definition phase, where the parties negotiate the respective authorizations on data covered by the agreement, the resulting policy may be analysed in order to identify possible conflicts or incompatibilities among authorizations clauses. In this paper, we propose a formal framework for Data Sharing Agreement analysis. Our proposal is built on a process algebra formalism dealing with contextual data, encoded into the Maude engine to make it executable. 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 analysis tool.**

***Keywords*-Data Sharing Agreements, Context-aware Data Protection, Formal Analysis, Web Service Implementation.**

I. Introduction

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

Usually, organizations adopt Data Sharing Agreements (DSA), which are formal legal agreements among two or more parties regulating how they share data. 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. A general structure for a DSA consists of the following sections (see also [1], [2] 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's clauses are usually subject to a lifecycle consisting at least of the following main phases: definition and enforcement. 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.

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

The FP7 European Project Consequence [3] 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 certain files) given certain contextual conditions.

We consider a high level description of the agreement, that we specify by means of the POlicy Process Algebra POLPA [4]. As execution environment for checking if the DSA/POLPA specification satisfies a set of properties we choose the rewriting system engine Maude [5]. 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 to achieve modularity with respect to the particular tool chosen for the analysis.

The paper is structured as follows. The next section lists some related work. Then, we recall two high-level languages for the description of DSA, CNL4DSA and POLPA. Section IV presents our executable specification of POLPA into Maude. In Section V, we consider a sensitive data sharing test bed. Then, we show the implementation of the DSA analysis architecture in Section VI. Then, we give final remarks.

II. State of the Art

Our proposal for a DSA analyser becomes part of an information protection framework aiming at defining a scalable

978-0-7695-4141-9/10 $26.00 © 2010 IEEE
DOI 10.1109/CENTRIC.2010.17

architecture to enable secure management of policies. The architecture incorporates models and tools for policy specification and authoring, *e.g.,* [6], [7], [2], and secure mechanisms for policy enforcement, *e.g.,* [8].

There have been some efforts towards the analysis of DSA. In particular, [9] 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. Binder [10] is an open logic-based security language that encodes security authorizations among components of communicating distributed systems. It has a notion for context and provides flexible low-level programming tools to express delegation, even if Binder does not directly implement higher-level security concepts like delegation itself. Also, the Rodin platform provides animation and model-checking toolset, for developing specifications based on the Event-B language [11]. In [12], it was shown that the Event-B language can be used to model obliged events. This could be useful in the case of analysing obligations in DSA. Some of the authors are our partners in the Consequence project, and we are currently investigating for setting up an integrated framework that could benefit from both the approaches.

Even if not specifically DSA-related, [13] presents a policy analysis framework, which considers not only authorizations, but also obligations, giving useful diagnostic information. Furthermore, the XACML language [14] is a pretty new standard for encoding data exchange. It makes possible a simple, flexible way to express and enforce access control policies in a variety of environments, using a single language. Also, the Semantic Annotations for WSDL and XML Schema (SAWSDL) language [15] is defined in the Web Service research area for adding semantic annotations to various parts of a document written in WSDL. Both XACML and SAWSDL are used for specifying policies but there are no references to automatic mechanisms for analysing policies expressed in these languages.

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 [16] 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.

III. High-level Languages for DSA Specification

This section recalls two languages for describing DSA. The first language is called CNL4DSA [2], and it is mainly intended for authoring a DSA in a user-friendly, but 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 give the general flavor of the whole working architecture, from DSA authoring to DSA analysis. [2] shows a mapping from CNL4DSA to POLPA.

A. CNL4DSA

In [2], a Controlled Natural Language for Data Sharing Agreements, namely CNL4DSA, has been proposed. The language aims at ensuring simplicity for end-users and safe translation to formal specifications allowing for automated verification of the authorizations clauses of a DSA.

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". Fragments are composable to form more complex expressions, *i.e., composite authorization fragments* F. The syntax of a composite fragment is inductively defined as follows:

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

where C is the *context*, *i.e.,* a predicate, 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").

The intuition 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; $\mathit{if}\ C\ \mathit{then}\ F$ expresses the logical implication between a context C and a composite authorization fragment: if C holds, then F is permitted; $\mathit{after}\ f\ \mathit{then}\ F$ is the temporal sequence of fragments. Informally, after f has happened, then the composite authorization fragment F is permitted.

B. POLPA

CNL4DSA has been proposed with an eye for formal verification (see [2] for the operational semantics of the language). Its semi-formal structure makes it suitable for a simple, intuitive and easy mapping to high-level formal policy languages. Here, we consider POLPA [4], [17], built on top of CSP [18] 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 [17] and for GRID systems [4]. We recall syntax and operational semantics of the language. The reader is referred to [4] for more details. The syntax of POLPA is as follows:

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

with process $P \in \mathcal{P}$, parameterized actions $\alpha(\vec{x}) \in Act$, and $p(\vec{x}) \in Pr$ 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;
- $\alpha(\vec{x}).P$ is the process that performs action $\alpha(\vec{x})$ and then behaves as P;

- $p(\vec{x}).P$ is the process that behaves as P when the predicate $p(\vec{x})$ is true; otherwise, it gets stuck;
- $P[fun]$ is the process that behaves as P but with the parameters $\vec{x}$ substituted according to fun. $fun : \mathcal{X} \rightarrow \mathcal{X}$ is a finite substitution function;
- 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 $Alfa \subseteq Act$ can only occur when both the processes perform that action. When P_1 and P_2 engage in actions not belonging to $Alfa$, 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 Figure 1, that defines a Labelled Transition System (LTS) for POLPA processes. In particular, the LTS is a structure ($\mathcal{P}$, Act, $\xrightarrow{\alpha(\vec{x})}$), where $\mathcal{P}$ is the set of POLPA processes, Act is the set of parameterised actions, and $\xrightarrow{\alpha(\vec{x})}$ 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 $P_1 \xrightarrow{\alpha(\vec{x})} P_2$, with $P_1, P_2 \in \mathcal{P}$ and $\alpha(\vec{x}) \in Act$ means that it is admissible that the implementation of P_1 performs $\alpha(\vec{x})$ 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.

IV. Encoding of POLPA into Maude

Maude is an executable programming language that models (distributed) systems and the actions within those systems [5]. Systems are specified by defining algebraic data types axiomatizing systems states, and rewrite rules declaring the relationships between the states and the transitions between them.

Here, the goal is to exploit the Maude engine to make POLPA executable. 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 [19], that implements the CCS operational semantics [20] 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 [19], 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
  sorts  ProcId Proc . sort ActSet .
  subsort Act < ActSet . subsorts Qid < ProcId < Proc .
  op tau : -> Act . op stop : -> Proc .

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

The module POLPA-DEFINITION (not shown here), following [19], defines process definitions and the operations to work with them. In practice, a constant *definition* 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-DEFINITION .
  sort ActProcess . subsort Proc < ActProcess .
  op {_}_ : Act ActProcess -> ActProcess [frozen] .
  vars A B : Act . vars L M : Action . vars RO RO' : Role .
  vars S S' : ActSet .  vars Y Y' : Subj . vars O O' : Obj .
  *** some other variables declaration....

  *** 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 (only two examples are shown)
  ---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)) .

  *** 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. The most general parameters one may think of are the subject performing an action and the target object of that action, *e.g.,* action *read* can have as parameters subject *Bob* and object *Document1*. Subjects may be single individuals as well as organizations. Objects are 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 Scientist, *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

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

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

$$(or)\frac{P \xrightarrow{\alpha(\vec{x})} P_1}{P + Q \xrightarrow{\alpha(\vec{x})} P_1}$$

$$(comp_1)\frac{P \xrightarrow{\alpha(\vec{x})} P_1}{P|\{Alfa\}|Q \xrightarrow{\alpha(\vec{x})} P_1|\{Alfa\}|Q}\alpha(\vec{x}) \notin Alfa \qquad (comp_2)\frac{P \xrightarrow{\alpha(\vec{x})} P_1 \quad Q \xrightarrow{\alpha(\vec{x})} Q_1}{P|\{Alfa\}|Q \xrightarrow{\alpha(\vec{x})} P_1|\{Alfa\}|Q_1}\alpha(\vec{x}) \in Alfa$$

$$(const)\frac{P \xrightarrow{\alpha(\vec{x})} P_1}{Z \xrightarrow{\alpha(\vec{x})} P_1} Z \doteq P$$

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

Fig. 1. POLPA operational semantics rules

first substitution rule), and with predicates parameterised by, *e.g.,* subjects and their roles (the second substitution rule), and objects and their categories (rule not shown). 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 [19], 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 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 to prove that a process can perform a certain sequence of actions, *i.e.,* a *trace*. Thus, we consider an 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 [21], [22]. 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.

These formulas will represent the logical specification of the DSA properties. Some examples will be given in Section V.

Here, we do not show the modules implementing the Hennessy-Milner logic over POLPA, since they are essentially the same as the ones in [19]. 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
  inc QID . inc QID .
--- here some sorts declaration
  ...
--- some roles declaration :
 ops PrincipalInvestigator Coinvestigator : -> Role .
--- some data category declaration:
 ops image investigation-metadata numerical : -> Typ .
--- some subjects and data declaration:
 op Caroline : -> Subj . ops doc1 doc2 : -> Obj .
--- predicates terms:
 ops has-role has-data-category has-embargo-end-date
 current-time-is-before 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(PR) = false [owise] .
endfm
```

V. 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 be applied 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. These policies have been provided by

colleagues from the Science and Technology Facility Council (STFC), our partner in the Consequence project.

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.

We are interested in answering to questions like:

- Action list: which are all the authorised actions in the investigated DSA?
- Single authorization: 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, under a set of contextual conditions?
- Look for temporal sequences of actions: is it true that, after opening object y, subject x can read object y?
- Search for *opposite* actions: is it true that subject x can read/write/print... object y and that, under the same context, 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, 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 an initial configuration with three subjects, *i.e.,* Caroline, Stephen, and Eve, and two objects, *i.e.,* doc1 and doc2. 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).

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.

Excerpts of the Maude representation of the POLPA process specifying the STFC DSA are shown in Figure 2. 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 subprocesses constituting the DSA (0 is the empty set of actions).

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 I
EXAMPLE QUERIES FOR THE SENSITIVE DATA SHARING TEST BED

We will ask something about the subjects' capabilities to perform certain actions on the objects, given certain contextual conditions. Table I shows an example list of queries and the expected boolean result for each query. The queries are given to the analyser in the Maude metalevel programming representation, *e.g.,* query 2 is represented as follows

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

Figure 4 in the next section shows the screenshot of executing Maude over the DSA specification and this list of queries.

Note that, in the DSA specification, 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, all the possible substitutions of the variables with the values declared in the configuration file are applied.

VI. IMPLEMENTATION

We choose to adopt the Web Service technology to publish the functionalities of the analysis tool. 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.

Within the Consequence project, a general xsd schema has been defined to include the DSA sections listed in the introduction of this paper. In particular, Figure 3 shows an XML instance of DSA according to the schema, highlighting the *Authorizations* section (in particular, Statement 9 of the specification given in Figure 2). 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.,* the Maude engine; 3) a graphical user interface (GUI) that acts as a front-end for the WS-Analyzer. This GUI shows the analysis results in a readable way.

A generic client application for the WS-Analyzer invokes the *analyze* method to perform the analysis of the DSA.

```
eq definition = ( '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)) @
   ('read(X4,X0)) . stop) [Stephen / X4] [doc2 / X0] ) & ....
   ( 'Statement9 =def ((has-data-category(X0,numerical)) @ (not-has-location-country(X4,Badland)) @
   ('read(X4,X0)) . stop) [Eve / 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 ..... ) .
```

Fig. 2. Excerpts of STFC DSA specification

```
- <dsa id="DSA-1.xml" status="NEW" xmlns="http://www.consequence-project.eu/dsa">
    <title>DSA</title>
  + <parties>
  + <validity>
  + <data>
  - <authorizations>
    - <authorization id="AUT_1264582586546">
        <expression language="CNL4DSA-E">IF ?X0 has_data_category ?X64:Numerical AND ?X4 has_role ?X66:Any AND ?X4 NOT
          has_location_country ?X67:Badland THEN ?X4 CAN read ?X0</expression>
        <expression language="UserText">IF that data has as data category a numerical data AND that person has as role any
          role AND that person NOT has location Badland THEN that person CAN read that data</expression>
        <expression language="CNL4DSA">IF has_data_category(?X0,?X64) AND has_role(?X4,?X66) AND NOT
          has_location_country(?X4,?X67) AND NOT has_location_country(?X4,?X68) THEN CAN [?X4, read, ?X0]</expression>
        <expression language="POLPA">(has-data-category(X0,numerical)) @ (has-role(X4,Any)) @ (not-has-location-country
          (X4,Badland)) @ ('read(X4,X0)) . stop)</expression>
      </authorization>
    </authorizations>
  + <TransactionHistory>
  </dsa>
```

Fig. 3. Instance of DSA with the Authorizations section highlighted

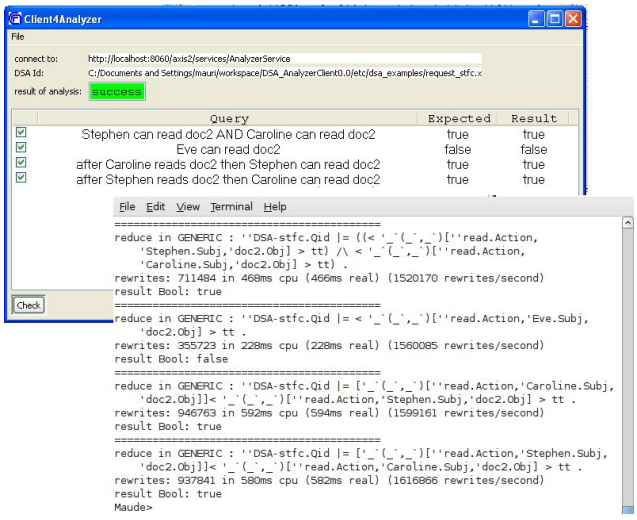

Fig. 4. GUI and Maude computation screenshot

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.

The GUI behaves as a client for the Analyzer and allows the user to load a specific DSA to be analysed. 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 4 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.

VII. 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.

Acknowledgments

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

References

[1] Deliverable D2.1, "Methodologies and tools for data sharing agreements infrastructure," http://www.consequence-project.eu/Deliverables_Y1/D2.1.pdf, 2008.
[2] I. Matteucci, M. Petrocchi, and M. L. Sbodio, "CNL4DSA: a controlled natural language for data sharing agreements," in *SAC*. ACM, 2010.
[3] "EU Project Consequence," http://www.consequence-project.eu/.
[4] F. Martinelli and P. Mori, "A model for usage control in grid systems," in *GRID-STP*, 2007.
[5] M. Clavel *et al.*, Eds., *All About Maude - How to Specify, Program and Verify Systems in Rewriting Logic*. Springer, LNCS 4350, 2007.
[6] C. Brodie *et al.*, "The coalition policy management portal for policy authoring, verification, and deployment," in *POLICY*, 2008.
[7] K. Kaljurand, "Attempto Controlled English as a Semantic Web Language," Ph.D. dissertation, Tartu Univ., 2007.
[8] E. Scalavino, V. Gowadia, and E. C. Lupu, "Paes: Policy-based authority evaluation scheme," in *DBSec*, 2009, pp. 268–282.
[9] V. Swarup *et al.*, "A data sharing agreement framework," in *ICISS*, 2006.
[10] M. Abadi, "Logic in access control," in *LICS*. IEEE, 2003, p. 228.
[11] "Event-B and the Rodin Platform," www.event-b.org.
[12] J. Bicarregui *et al.*, "Towards modelling obligations in Event-B," in *ABZ*, 2008, pp. 181–194.
[13] R. Craven *et al.*, "Expressive policy analysis with enhanced system dynamicity," in *ASIACCS*, 2009.
[14] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml, Last access May 17, 2010.
[15] "Usage guide," http://www.w3.org/TR/2007/NOTE-sawsdl-guide-20070828/, Last access May 17, 2010.
[16] R. De Nicola, G. L. Ferrari, and R. Pugliese, "Programming access control: The klaim experience," in *CONCUR*, 2000, pp. 48–65.
[17] B. Aziz *et al.*, "Controlling usage in business process workflows through fine-grained security policies," in *TrustBus, LNCS 5185*. Springer, 2008.
[18] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.
[19] A. Verdejo and N. M. i-Oliet, "Implementing CCS in Maude 2," *ENTCS 71*, 2002.
[20] R. Milner, *Communication and Concurrency*. Prentice Hall, 1989.
[21] M. Hennessy and R. Milner, "On observing nondeterminism and concurrency," in *ICALP*, 1980, pp. 299–309.
[22] C. Stirling, "Modal and temporal logics for processes," in *Logics for concurrency: structure versus automata*. Springer, 1996, pp. 149–237.