An Algorithm for Allocating User Requests to Licenses in the OMA DRM System*

Nikolaos TRIANTAFYLLOY†(a), Petros STEFANEAS†(b), and Panayiotis FRANGOS†(c), Nonmembers

SUMMARY The Open Mobile Alliance (OMA) Order of Rights Object Evaluation algorithm causes the loss of rights on contents under certain circumstances. By identifying the cases that cause this loss we suggest an algebraic characterization, as well as an ordering of OMA licenses. These allow us to redesign the algorithm so as to minimize the losses, in a way suitable for the low computational powers of mobile devices. In addition we provide a formal proof that the proposed algorithm fulfills its intent. The proof is conducted using the OTS/CafeOBJ method for verifying invariant properties.

key words: Mobile DRM, OMA, Order of Rights Object Evaluation, CafeOBJ, Safety, Invariant properties

1. Introduction

Digital Rights Management (DRM) Systems are used by most digital content vendors. Thus, the need to ascertain their reliable behavior is very strong. Open Mobile Alliance (OMA) is an organization responsible for the definition of standards for the Mobile DRM systems [1]. The proposed standards include OMA-DRM [2] and OMA REL [3], where the latter specifies the language in which licenses are written. OMA-REL is XML based and is defined as a mobile profile of ODRL [2]. OMA’s DRM is currently implemented in most mobile devices and smart phones and is adopted by most vendors for mobile content. We demonstrate that the OMA License Allocation Algorithm currently in use suffers from a loss of execution permissions (or rights) and suggest a new algorithm to overcome this. Although this algorithm is designed to address problems of the OMA DRM system, the proposed methodology can be applied to other DRM domains and languages as well [4,5,6] to allow an automated license selection with minimal loss of rights.

Barth and Mitchel [7] first identified this loss of rights and argued that a correct algorithm should behave monotonically, i.e., if a set of rights is allowed by a set of licenses then this set is also allowed by a set of more flexible licenses (licenses that contain at least the same rights**).

We suggest an algorithm to overcome the unintentional loss of rights without designing complex and computationally heavy DRM agents that reallocate actions to rights. There exist some cases where our algorithm is not fully monotonic and this is intentional. In the case where some form of loss is inevitable we prompt the user to decide which rights he prefers, so that we may ascertain that those to be lost are the least desired. While this can cause the algorithm to behave non-monotonically, we believe that in this special case an algorithm that respects the desires of the user is preferable to a fully monotonic one.

The paper is organized as follows: section 2 briefly overviews related work and gives a short comparison with ours. Section 3 presents OMA’s algorithm and the loss of execution rights. In section 4 we give an introduction to ordered sorted algebra and present our new algorithm. Section 5 introduces the reader to the concepts of Observation Transition Systems (OTS) and the algebraic specification language CafeOBJ. Also in section 5 a specification of a DRM agent using the suggested algorithm is presented and used to prove formally that such a system does not suffer from unintentional loss of rights. Finally section 6 concludes the paper.

2. Related Work

DRM licenses can be regarded as special cases of authorization policies, where the properties have been widely studied in the literature. The discussed problem is similar to the policy reachability problem, i.e., given a policy and a domain, what the sequence of actions that leads to a state satisfying a goal property is. In [16] the authors analyze reachability and availability properties in Administrative Role-Based Access Control policies and prove that reachability analysis is PSPACE-complete for their domain. Dougherty et al., [17], study reachability, availability and containment queries for dynamic access control policies in Datalog and prove that reachability is in NLOGSPACE. Becker [18] presents a language for specifying dynamic authorization policies based on transition logic. Also a method for the verification of reachability based on automated theorem proving is presented. Other related research on policies includes [19] where the authors present static analysis methods for the particular questions of whether policies contain gaps or

---

*The author is with the school of Electrical and Computer Engineering, National Technical University of Athens
†The author is with the school of Applied Mathematical and Physical Sciences, National Technical University of Athens
‡This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program ‘Education and Lifelong Learning’ of the National Strategic Reference Framework (NSRF) - Research Fund–Program: THALES: Reinforcement of the interdisciplinary and/or inter-institutional research and innovation
a) E-mail: nitriant@central.ntua.gr
b) E-mail: petros@math.ntua.gr
c) E-mail: pfrangos@central.ntua.gr
DOI: 10.1587/trans.E0.??.1

**Note that in this context we are only interested in the monotonicity of the existence of rights and not of the quantity of these rights

Copyright © 200x The Institute of Electronics, Information and Communication Engineers
conflicts. In [20] a method based on Event Calculus (EC) for policy and system specifications is presented that allows the analysis and detection of various conflict types.

The work of Barth and Mitchell [7] is different than the previous and is the closest to ours. They investigate the same problem with us (i.e., losing rights in the context of the OMA DRM) and prove that it is NP-complete. Also, they present a formalization of licenses using linear logic and define an algorithm that allows the revoke of the previous allocations and their reallocation to achieve monotonic behavior. Their approach however is rather computationally heavy and hard to implement; the DRM agent must keep track of all past allocations for all licenses ever installed so as to be able to revoke them if a loss occurs. The main differences of our approach is that by using order sorted algebra we were able to move the computational weight from the DRM agent to the creator of the licenses who naturally has more resources available than a mobile device. Also the computationally heavy steps need only be conducted once at the creation of the licenses. This allowed us to design an algorithm that behaves monotonically (when such behavior is desired) and is more suitable for the mobile environment.

3. The OMA Allocation Algorithm

3.1 Licenses in OMA DRM/REL

**Definition 1:** A license written in OMA REL consists of a set of sublicenses. Each sublicense is defined as subl =< Cons, CPerm >, where Cons is a set of constraints and CPerm is a constraint permission set. The semantics of a sublicense is that the set of constraint permissions, CPerm, is authorized if all the constraints of Cons are met. A set of constraints is defined as Cons = \{ c | c \in Constraints \}, where Constraints = \{ count, timed-count, Date-time; Interval, True, individual, system, accumulated \}. A Constraint permission set is defined as CPerm =< Cons, Perm > where Perm is a set of permissions. The semantics of a constraint permission set, CPerm, is that the set of permissions (or rights), Perm, is allowed to be executed when the set of constraints, Cons, is met. A set of permissions is defined as Perm = \{ \langle p, cont > | p \in Permissions \} where Permissions = \{ \langle play, display, print, execute, export \rangle \} and cont denotes a content protected by the DRM system.

We should note here that the count and timed-count constraints contain a positive integer stating the number of times the right can be executed and the DRM agent must reduce this number with each execution [3]. The semantics of datetime is that the constrained right can only be exercised within the specified date. That of interval is that the right can only be used for the defined time period, which starts after the first use of the right. The accumulated constraint specifies the maximum period of metered usage time during which the rights can be exercised over media content. The individual constraint binds the content to a user identity. Finally the system constraint defines the system to which the content can be exported to. For the rest of the paper we will regard licenses that contain an accumulated constraint, as being constrained with a count constraint which only allows one more execution. This is due to the nature of accumulated that can potentially be falsified after any execution of the right it constrains †. Also for simplicity since individual and system constraints are not taken into account by the original algorithm we will consider them as true constrained, i.e., unconstrained.

Please note that the constraints allowed by OMA REL do not allow references to other licenses/sublicenses and cannot express prohibition of actions. For example, it is not possible to express the licenses: License 1: “do A”; License 2: “if did A or C then cannot do B”. The use of the rights in a license only affects that license. Thus, permissions of a license can only become unavailable by using that license and thus making some of its constraints invalid, i.e., depleting a part of the license. ††.

3.2 The OMA Allocation Algorithm

In a DRM environment users may end up with licenses from different sources. Thus these licenses may refer to the same content. For example, L1: “you may listen to songs A or B once before the end of the month”. L2: “you may listen to songs A or D ten times.” A problem arises as to what license should be considered optimal. OMA specifies a set of rules defined in [3], that the DRM agent must apply when it automatically selects which license to use when multiple licenses contain rights that can satisfy the user request [3]. These rules, shown in table 1, define an ordering on the constraints of OMA REL that is applied to the constraint permission sets and sublicenses. Though not explicitly stated these rules are inevitably projected to the licenses themselves. So for a fixed user request, applying them produces an ordering on the licenses themselves. We will refer to them as the OMA Allocation Algorithm. In the above example we are posed with a question; what license should the DRM agent choose when requested to play song A? Based on the rules imposed by OMA REL, the agent must select the first license because it contains a date-time constraint (for one month) which is ranked higher in the ordering than the count constraint (ten times) of the second license. The intent behind this is that the ten times can be used whenever the user chooses while the date-time constraint will expire even if it is not exercised. It is clear that this rule was created to benefit the user. In fact all of these rules are of similar intent; unconstraint rights are to be preferred over constraint and so on [3]. So we can argue that the aim of this algorithm, is to allow for an automated decision making process that will result in protecting the interests of the user by choosing to use, the license that maximizes the rights available to him.

†the user could keep rendering the content for an amount of time that surpasses the defined timed limit without remaking a request

††the only constraints that can display such a behavior are count, timed-count and accumulated
This clearly can be seen as against the best interest of the user, which as we argued is the intent of this algorithm and in [7] is characterized as an infuriating situation.

Consider now a new set of licenses. L1: "you can listen to songs A or B once". L2: "you can listen to songs A or C or D once". Assume a user request to listen to song A. Here all the licenses that contain the request cause a true loss of rights. How should an algorithm decide in this case? One option would be to simply allocate the request to the license that causes the smallest loss of rights possible. But the user might value the right to listen to song B more than the rights to listen to songs C or D combined. Thus we believe that in the cases where a true loss is inevitable the final decision should rest on the user.

Based on these observations we argue that a correct allocation algorithm must satisfy the following property: "For a user request r if \( \forall l \in ls \text{ such that } l \text{ satisfies } r, \text{ and } \text{rights}(ls) \setminus \text{remnants}(ls; l; r) \subseteq \text{rights}(ls) \setminus \text{rights}(ls; l; r) \) then the algorithm must select \( l' \in ls \text{ containing } r, \text{ such that for all other licenses } l'' \in ls \text{ that contain } r \text{ it holds that } \text{remnants}(ls; l''; r) \subseteq \text{remnants}(ls; l'; r) \) " \(^\dagger\). We will refer to this property as the Weak Minimal Loss (WML) Property.

4. Redesigning the Algorithm

4.1 Order Sorted Algebra

An order sorted algebra (OSA) [11] is a partial ordering \( \leq \) on a set of sorts, i.e. a set of names for data types. An s-sorted algebra \( A \) is a mapping between the sort names and subsets from the set \( A \). This sortset relation imposes a restriction on an s-sorted algebra \( A \), if \( s \leq s' \) then \( A_s \subseteq A_{s'} \) where \( A_s \) denotes the elements of sort \( s \) in \( A \). Order sorted algebra [11] provides a way for several forms of polymorphism and overloading, error definition, detection and recovery, multiple inheritance, selectors when there are multiple constructors and many more [11]. Formally, given a partially ordered sort set \( S \), an S-sorted set \( A \) is just a family of sets \( A_s \) for each sort \( s \in S \). A many-sorted signature is a pair \((S, \Sigma)\) where \( S \) is called the sort set and \( \Sigma \) is an \( S^* \times S \)-sorted family of functions \( \{ \Sigma_{w,s} | w \in S^* \text{ands } s \in S \} \). An order sorted signature is a triple \((S, \leq, \Sigma)\) such that \((S, \Sigma)\) is a many-sorted signature, \((S, \leq)\) is a poset, and the operations satisfy the following monotonicity condition; \( \sigma \in \Sigma_{s1,1} \cap \Sigma_{s2,2} \) and \( w1 \leq w2 \) imply \( s1 \leq s2 \). Given a many-sorted signature an \((S, \Sigma)\)-algebra \( A \) is a family of sets \( \{A_s | s \in S\} \) called the carriers of \( A \), together with a function \( A_{\sigma} : A_{\omega} \rightarrow A_{\sigma} \) for each \( \sigma \in \Sigma_{w,s} \), where \( A_{\omega} = A_{\sigma1} \times \ldots \times A_{\sigma n}, w = s1 \ldots s n \).

4.2 Labeling Licenses and the Proposed Algorithm

Licenses are basically a data type. Consequentially, there exists a set of sort names \( S \), that can be used to represent these licenses. In addition, based on the rights object evaluation

\[ \text{remnants}(ls; l; r) = \{ \text{play } A, \text{ play } B \} \]

\( \dagger \)"i.e., that a correct allocation algorithm must satisfy the minimal loss property of table 2, when a true loss of rights is not inevitable.
provided by the OMA Allocation Algorithm an ordering on these sorts can be defined. So if we identify the order sorted algebra that is “hidden” in the definition of licenses we can create an algorithm. The basic idea is to apply this ordering to decide which is the most suitable license to use. For a loss of rights to occur, some constraints of the licenses must no longer hold after the satisfaction of a user request. From the semantics of the constraints supported by OMA REL given in section 2.1, it is clear that the only constraints that can be falsified by the satisfaction of a request are the count and timed-count constraints\textsuperscript{111}. According to this, we can argue that each license should be characterized by the following observations:

- Some part of the license becomes depleted after the execution of a right
- The license contains more than one permission elements
- The characterizing constraint based on the OMA constraint ordering

So in order to meet the conditions of the first bullet a license must contain either a count constraint, or a timed-count constraint (both with only one more execution left). The second bullet can be easily checked at the level of the creation of a license. The characterization of the last bullet can be made with a simple search on the constraints of the licenses. We can now define an order sorted signature for OMA REL licenses as \((s_1 \times s_2 \times s_3, \leq, \Sigma)\) with \(\Sigma = \{\emptyset\}\) and \(\Sigma 3 = \{\text{Count, Timed-count, Date-time, Interval, True}\}\) the names of the various constraints allowed by OMA REL (we omit the constraints \textit{individual} and \textit{system} because they do not play any part in the decision made by the original algorithm). \(s1 = \{\text{Once, Many}\}\) denoting whether the license will allow more than one execution of its permissions. Finally \(s2 = \{\text{Complex, Simple}\}\), denoting if the license contains more than one permissions.

For example, a label \(l = \text{Once} \times \text{Simple} \times \text{Count}\), states that the license allows only one more execution of a right, it contains only one right and the dominant constraint of the license is a count constraint. The ordering comes from the predefined ordering of the rights object evaluation in conjunction with the following definitions: \(\text{once} < \text{many}\) and \(\text{simple} < \text{complex}\). So, formally we have that \(s_1 \times s_2 \times s_3 \leq s'_1 \times s'_2 \times s'_3\) implies that \(s_1 < s'_1\) or \((s_1 = s'_1\) and \(s_2 < s'_2\) or \((s_2 = s'_2\) and \(s_3 \leq s'_3\))\). Using this ordering on licenses we will present in the next section a new algorithm for the decision problem of the optimal license.

We augment a license to contain these sort names by using labels that will be added in two points: the sublicenses as a top label and to the constraint permission sets as a local label. This should be done simultaneously with the creation of the licenses to reduce the computational cost on the mobile devices. In addition we assume that the DRM agent is enhanced so as to be able to update these labels after the execution of permissions as necessary. Meaning that if a sublicense after the execution of right allows only one more use its label should be updated to \(\text{Once} \times \text{Simple} \times \text{True}\) from \(\text{Many} \times \text{Simple} \times \text{True}\). Our approach does not require any knowledge on behalf of the agent on the future or past actions of the users as in [7]. Also we retain the OMA allocation algorithm in the core, so the implementation of the proposed algorithm to the existing DRM agents should have minimal cost.

\textbf{Definition 3 (Labeled Licenses)}: A labeled license consists of a set of sublicenses. Each sublicense is a triplet \(\text{sub-l} = \langle \text{Cons, CPerm, label} \rangle\) such that \(\text{sub-l}' = \langle \text{Cons, CPerm} \rangle\) is an OMA REL sublicense and label a label. We define operators to retrieve them as \(\text{Constraints(sub-l)}, \text{CPS(sub-l)}\) and \(\text{label(sub-l)}\) respectively.

Each constraint permission set is a triplet, \(CP = \langle \text{Cons, Perm, label} \rangle\), with \( CP' = \langle \text{Cons, Perm} \rangle\), a OMA REL constraint permission set and label a label. These are retrieved by \(\text{Constraints(CP)}, \text{Perm(CP)}\) and \(\text{label(CP)}\) respectively.

Assuming variables complexity \(\epsilon S_1\), times \(\epsilon S_2\) and constraint \(\epsilon S_3\), we define that, \(\text{label(CP)} = \text{times} \times \text{Simple} \times \text{constraint}\) iff \(#\text{Perm(CP)}\) = 1. Meaning that a set of constraint permissions is labeled as \(\text{times} \times \text{Simple} \times \text{constraint}\) iff it contains only one permission. Else it is labeled \(\text{times} \times \text{Complex} \times \text{constraint}\). Also, \(\text{label(CP)} = \text{Once} \times \text{complexity} \times \text{constraint}\) if the execution of any permissions in \(\text{Perm(CP)}\) causes some constraint in \(\text{Constraints(CP)}\) to fail. Denoting that CP can only be used one more time. Else \(\text{label(CP)} = \text{Many} \times \text{complexity} \times \text{constraint}\). A sublicense now is labeled as, \(\text{label(sub-l)} = \text{times} \times \text{Simple} \times \text{constraint}\) if \(#\text{CPS(sub-l)}\) = 1. This means that the sub-license contains only one constraint permission set. Else \(\text{label(sub-l)} = \text{times} \times \text{Complex} \times \text{constraint}\). Finally \(\text{label(sub-l)} = \text{Once} \times \text{complexity} \times \text{constraint}\) if the execution of a permission belonging to any of the constraint permission sets of the sub-license, \(\text{CPS(sub-l)}\), causes \(\text{Constraints(sub-l)}\) to no longer hold. Else \(\text{label(sub-l)} = \text{Many} \times \text{complexity} \times \text{constraint}\).

For example consider the sublicense: \(\text{sub-l} = \{\langle \text{Once before the end of the month} \rangle \text{either \{up to ten times play songs A or B\} or \{once print document C\}\}}\) Here, \(\text{Constraint(sub-l)}\) = \{\langle \text{one time, before the end of the month} \rangle\}. \(\text{CPS(sub-l)} = \{\langle \text{CP1, CP2}\rangle\) where \(\text{CP1} = \{\langle \text{up to ten times, play either song A or B}\rangle\) and \(\text{Constraints(CP1)} = \{\langle \text{up to ten times}\rangle\), \(\text{Perm(CP1)} = \{\text{play song A, play song B}\}\). Likewise \(\text{CP2} = \{\langle \text{one time, print document C}\rangle\) So here \#\(\text{CPS (sub-l)}\) = 2.

\textbf{Definition 4 (Satisfiability)}: For a permission \(P\) and a request \(r\) we define that \(\text{sat}(P, r) = \text{true}\) iff \(P = r\). For a constraint permission set \(CP\), we define that \(\text{sat}(CP, r) = \text{true}\) iff there exists a permission \(P \in CP\) such that \(\text{sat}(P, r)\). For a sublicense \(\text{sub-l}\), we define \(\text{sat}(\text{sub-l}, r) = \text{true}\) iff \(\exists CP \in \text{sub-l}\) such that \(\text{sat}(CP, r)\). For a license \(l\) sat\((l, r) = \text{true}\) if all permissions in \(l\) are satisfied.
true iff \( \exists \text{subl} \in l \) such that \( \text{sat}(\text{subl}, r) \). Finally for a set of licenses \( \text{ls} \), \( \text{sat}(l_s, r) = \text{true} \) iff \( \exists l \in \text{ls} \) such that \( \text{sat}(l, r) \) holds. Also for label \( l = \text{complexity} \times \text{times} \times \text{constraint} \), we define \( \text{times}(l) = \text{times} \), \( \text{comp}(l) = \text{complexity} \) and \( \text{cons}(l) = \text{constraint} \).

An abstract version of the proposed algorithm using this notation is shown in Table 3. A true loss of rights will occur when the selected license contains the user request in a constraint permission set \( CP \), of a sublicense \( \text{subl} \), such that either \( \text{sat}(\text{subl}) \) or \( \text{subl} \in l \) and \( \neg \text{times}(\text{label}(\text{subl})) \) or \( \text{comp}(\text{label}(\text{subl})) = \text{complex} \). The goal of the algorithm is to avoid such selections if that is possible. Else, the user is prompted to ensure the satisfaction of his preference on the available rights.

Using the above algorithm there exist only two ways for a permission other than the request to get lost. The first case is when all the licenses for which \( \text{sat}(l, r) \) holds cause a true loss. In this case the user is prompted by the algorithm as to which rights he prefers to lose. Here it is clear that the algorithm protects the preferences of the users and we consider this loss as intentional. The second case occurs when there exists only one license \( l \) such that \( \text{sat}(l, r) \) and \( l \) causes a true loss of rights. But as we must always satisfy the request if there exists a suitable license, this loss is also considered intentional.

An implementation of this algorithm for labeled licenses was created in Java and several case studies were conducted. In all cases when a true loss of rights was not inevitable the algorithm correctly selected the license that caused no true loss.

### 5. Verification of the proposed algorithm

In this section we present a formal proof that the proposed algorithm of table 3, satisfies the WML property. The proof was conducted by specifying an arbitrary OMA DRM system that uses this algorithm, as an Observation Transition System (OTS) [12] expressed in CafeOBJ terms [13].

If for a request \( r \) there exists at least one \( l \in \text{ls} \) such that \( \text{sat}(l, r) = \text{true} \), then there exist three cases under which the selection of a license satisfies the WML property:

1. Only one license exists such that \( \text{sat}(l, r) \).
2. All the licenses for which \( \text{sat}(l, r) \) cause a true loss of rights.
3. There exists \( l \in \text{ls} \) such that \( \text{sat}(l, r) \) and \( l \) does not cause a true loss of rights. Then one of the following must hold:
   a. No part of \( l \) gets depleted.
   b. If 3a) does not hold then, \( \text{remnants}(\text{ls}, l, r) = \{ r \} \).

Based on this observation we define a coloring on the rights, that changes every time a request is satisfied by the DRM agent. Using this coloring we then transform the WML property into a formula that is easier to verify with the OTS/CafeOBJ method.

**Definition 5** (Coloring): For a set of licenses \( \text{ls} \) we define that \( \forall p \in \text{right}(\text{ls}) \) initially \( \text{color}(p) = \text{white} \). After a request \( r \), the selection of license \( l \) by the algorithm and the execution of a right if no part of \( l \) is depleted, then the coloring of the rights remains unchanged. If some part of \( l \) gets depleted, causing the rights \( \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}, l, r) \) to become unavailable, we define that the color of \( p \in \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}, l, r) \} \) becomes black: if \( l \) is the only license such that \( \text{sat}(l, r) \) or, if all the other licenses \( l' \) for which \( \text{sat}(l', r) \) holds also cause a true loss of rights or, if \( p \) matches the user request \( r \). Else the color of \( p \) remains unchanged.

We denote by \( \text{depleted}(l_s, r') \) the set of permissions lost by a sequence of satisfied rights \( r' = r_0, \ldots, r_{n-1} \) from the set \( \text{ls} \). Formally, \( \text{depleted}(l_s, r') = \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}; l_0; r_0) \} \cup \ldots \cup \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}; l_{n-1}; r_{n-1}) \} \). Where \( l_i \) is the license chosen to satisfy the request \( r_i \). Also \( \text{ls}^0 \) is the set of licenses after the satisfactions of i-1 requests.

**Proposition 1:** The WML is equivalent to the safety property: \( (p \in \text{right}(\text{ls}) \land p \in \text{depleted}(l_s, r')) \rightarrow \neg \text{color}(p) = \text{white} \)

**Proof(Sketch)** If the safety property does not hold after an arbitrary number of requests \( n \), then \( \exists p \in \text{right}(\text{ls}) \land p \not\in \text{depleted}(l_s, r') \) such that \( \text{color}(p) = \text{white} \). But, \( p \not\in \text{depleted}(l_s, r') \) implies that \( \exists i \leq n \), \( l_i \in \text{ls} \), such that \( l_i \) was selected for a request \( r_i \) and some part of \( l_i \) got depleted. Also since, \( \text{sat}(l_i, r_i) \) then \( r_i, p \in \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}^0; l_i; r_i) \} \). However, \( \text{color}(p) = \text{white} \), so from the coloring definition \( p \neq r_i \land \exists l' \in \text{ls}^0 \) such that \( \text{sat}(l', r_i) \land l' \neq l_i \land \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}^0; l'; r_i) \} \subseteq \{ r_i \} \). So, \( \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}^0; l'; r_i) \} \subseteq \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}^0; l_i; r_i) \} \). Which implies that \( \text{remnants}(\text{ls}^0; l_i', r_i') \supseteq \text{remnants}(\text{ls}^0; l_i; r_i) \). I.e., the WML property does not hold. If the WML property does not hold, then for some request \( r_i \), \( \exists l_i \in \text{ls}^0 \) such that \( \text{sat}(l_i, r_i) \land \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}^0; l_i; r_i) \} \subseteq \{ r_i \} \) and the algorithm selects \( l' \in \text{ls}^0 \) such that \( l' \neq l_i \land \text{sat}(l', r_i) \land \text{remnants}(\text{ls}^0; l'; r_i) \supseteq \text{remnants}(\text{ls}^0; l_i; r_i) \). This implies that \( \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}^0; l_i, r_i) \} \subseteq \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}^0; l'; r_i) \} \). Because \( \text{sat}(l', r_i) \), then \( r_i \in \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}^0; l'; r_i) \} \). Also, \( r_i \in \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}^0; l'; r_i) \} \). But, \( \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}^0; l_i, r_i) \} \subseteq \{ \text{right}(\text{ls}) \backslash \text{remnants}(\text{ls}^0; l_i, r_i) \} \).
5.1 Observational Transition Systems and CafeOBJ

An Observation Transition System (OTS) is a transition system that can be written in terms of equations. Assuming a universal state space \( Y \), an OTS \( S \) is a triplet \( S = \langle O, I, T > \) where \( I \subseteq Y \) is the set of initial states of the system and \( O \) is a set of observation operators. Each observer in \( O \) is a function that takes as input elements \( \tau \) of \( Y \) and possibly a set of data-type values and returns a new state of the system. If \( \tau \in \tau \), \( \tau(u_1, u_2) \) is defined as \( \tau \subseteq O, o(u_1) = o(u_2) \). The previous equality creates the equivalence classes, \( Y \| \|_\tau \), on the states of an OTS. Finally, \( T \) is the set of transitions, conditional functions (or actions). Each transition takes as input a state of the system (hidden sort) and possibly a set of data-type values (visible sorts) and returns a new state of the system. If \( \tau \in T \) then \( \tau(u_1) \subseteq \tau(u_2) \) for each \( u_1, u_2 \in Y \). For each \( u \in Y \), \( \tau(u) \) is called the successor state of \( u \). The condition \( \tau \subseteq \tau \), \( \tau(u) \) is applied in a state \( u \) if \( \neg \tau \).

An OTS defines a Behavioral Object (BO), BO Composition has been defined formally in [14]. From the state of the composite object we can retrieve the state of the component objects via Projection Operators [14]. There are several ways to compose an object from component objects. Parallel Composition without Synchronization, if the changes on the states of an object do not affect the states of the other objects of the same level. Parallel Composition with Synchronization when the changes in the state of one object may alter the state of an object in the same level. In respect to the number of objects that compose a composite object, we have Dynamic Composition if that number of component objects is not fixed and Static if it is.

CafeOBJ is an algebraic specification language [13]. An OTS can be written in CafeOBJ in a natural way. Moreover, hierarchical behavioral object composition, has already been defined in [14] with the use of CafeOBJ. The universal state space \( Y \) is denoted in CafeOBJ by a hidden sort, while each observer by an observation operator. Assuming visible sorts \( V_{ij} \), \( V \) that correspond to the data types \( D_{k, D} \), \( k = i, \ldots, m \), and a hidden sort \( H \), the observation operator denoting \( o_{m, \ldots, i} \) is declared as follows; bop o: \( V_{1, \ldots, m} H \rightarrow V \). Any state in \( I \) is denoted by a constant, say init, which is declared as; op init: \( \rightarrow H \). A transition \( \tau_{j_1, \ldots, j_m} \in T \) is denoted by a CafeOBJ action operator; bop \( \tau: V_{j_1, \ldots, j_m} H \rightarrow H \), with \( V \) a visible sort corresponding to the data type \( D_k \) and \( k = j_1, \ldots, j_m \). Each transition is defined by stating the value returned by each observer in the successor state, when \( \tau_{j_1, \ldots, j_m} \) is applied in a state \( u \) when \( \neg \tau_{j_1, \ldots, j_m}(u) \) holds. The value returned by \( o_{1, \ldots, m} \) is not changed if \( \tau_{j_1, \ldots, j_m} \) is applied in a state \( u \) such that \( \neg \tau_{j_1, \ldots, j_m}(u) \). The basic building blocks of a CafeOBJ specification are modules. Each module defines a sort. CafeOBJ provides built in modules for the most commonly used data-types (visible sorts) like BOOL, NAT and so on. An underscore _ in the definition of an operator indicates the place where an argument is put. The keyword mod! (mod\*) indicates that the module defined has tight (loose) semantics. Visible (hidden) sorts are denoted by enclosing them within [ ] and _ respectively. The keyword eq is used to denote an equation and ceq to denote a conditional equation. Modules can be imported using the keyword pr. Finally the key word bop is declares observation and action operators.

5.2 OTS specification of a DRM agent using the proposed Algorithm

Before introducing the OTSs that specify the system, we need first to specify the data types required. These specifications consist of modules that define visible sorts, corresponding to a data type. The following data types were required: Content, Permission, Request, Colors, Label, SET, Constraint, ConstraintSet, ConstraintPermission, SetOfCP, License and finally LicSet. In table 4 the specification of the ADT of Labels is shown. The operator \( \& \) takes as input elements of \( L_{i, j, k} \) and \( S_{i, j, k} \) respectively and returns a label. Operators type1?, type2? and type3? can be used to retrieve these elements and are the specifications of the operators comp, cons and times from section 4.2 respectively.

Using the above ADTs we define the hidden sorts for our OTS. Each such sort defines the state space of an abstract machine. In our specification we used two such sorts. The first sort, Lsys, specifies a system that consists of a set of
licenses that can deplete a whole sublicense or a constraint permission set after a relative request. This system is defined in module LOTS. The observers and transitions used to define it can be found in Table 5. In this OTS all the installed licenses have their constraints met at the initial state of the system.

To specify a DRM agent that uses the new algorithm this OTS does not suffice. A DRM system that uses the proposed algorithm should be able to handle:

- the receipt of a user request
- the selection of an appropriate license (using the proposed algorithm)
- the satisfaction of the request.

We define an OTS specifying the above using the OTS defined in module LOTS as a component object. This new composite OTS defines a novel state space denoted by the hidden sort sys and for its definition the observers of Table 6 were used.

In order to derive the state of the component object from the composite object we use projection operators. Since in this specification we have one component we only define one projection: bop license_ : sys → Lsys. For example assuming that init is a CafeOBJ constant that denotes an arbitrary initial state of the composite OTS, we derive the state of the component system with the following equation, stating that the component object will be at an arbitrary initial state as well: eq license(init) = initl.

In OTS terms the functionalities required by a DRM system that uses the proposed algorithm are naturally expressed and modeled as transitions. In our specification the first two are defined through the request and choose transitions. The request transition can successfully change the state of the system only if there is no pending user request. If the transition is successful it stores the new request, using the observer useReq. The second transition defines the selection process of the proposed algorithm. This is achieved using the possLic and finalLic observers. The first observer returns the set of licenses from the third step of the algorithm. The second observer returns the set of licenses from the fourth step of the algorithm. The values of these observers are calculated using the operators build3 and build respectively. These take as input a set of licenses and check each license to see if some conditions hold. Finally, they return those that satisfy them. The selection of the algorithm is the application of the original ordering to the finalLic set, if it is not empty (using the OMA operator that simulates the selection of the original algorithm). This was defined with the following equation:

\[ \text{ceq best(choose(S)) = OMA(useReq(S), finalLic(choose(S)))} \]

if (#finalLic(choose(S)) >= 1) and c-choose(S).

Please note that because CafeOBJ is executable if we defined a set of licenses and request in its terms by using these operators CafeOBJ would return a license matching the output of the algorithm, thus simulating its execution.

We need one final transition that models the satisfaction of the request. This must deplete the chosen licenses as necessary and change the color of the rights based on the defined coloring. However, the complexity of the verification of this transition would be very high. We can however split it into sub transitions based on the conditions of the depletion of the chosen license and the coloring of the lost rights.

That is, we use some of these conditions to define the effective conditions of the transitions. In this way we reduce the complexity of the observers’ definition.

Assume that the value of the observer (best) returning the output of the algorithm is L, and that the request belongs to the subl, sublicense of L and in the cp, constraint permission set of subl. For the coloring of the rights the following sentence will either hold or not: "there is only one license satisfying the request or all such licenses cause a true loss of rights" (we abbreviate this property as Q). Also, for the depletion of the license we want to discriminate whether the constraint permission set, cp, and the sublicense, subl, become depleted. Namely, if Q holds we have the following cases: times(cp) = Once and times(subl) = Many. This situation is defined by the transition use1. Next, times(cp) = Once and times(subl) = Once. This is defined by the use3 transition. Also, times(cp) = Many and times(subl) = Once, is defined by the use3 transition as well. Finally, times(cp) = Many and times(subl) = Many, is defined by the use2 transition. Now, if Q does not hold, the corresponding subcases are defined using the transitions use4, use5, use5 and use2 respectively.

So in this way we split the desired transition into five sub-transitions. However, these sub-transitions are characterized by the conditions we would have to consider in the equations defining the coloring and the depletion of the selected license for the original transition. Also, please note that these conditions are checked against the license that the OTS selects as the optimum to use (as part of the effective conditions of the transition rules). Thus, these sub-transitions are simply a convenient way to define after the selection of the optimum license by the algorithm, what parts of it become depleted and also the coloring of the lost rights. Concluding no additional information is used by these sub-transitions and thus the OTS remains a strict
Table 6 Observers of the composite OTS

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>licIns : sys → licSet</td>
<td>Returns the set of installed licenses</td>
</tr>
<tr>
<td>useReq : sys → reqErr</td>
<td>Returns the request of the user at the given state, if it exists. Else returns an error constant</td>
</tr>
<tr>
<td>best : sys → lic</td>
<td>Returns the license the algorithm selects for the user request</td>
</tr>
<tr>
<td>color : sys perm → color</td>
<td>Returns the color of the given permission</td>
</tr>
<tr>
<td>possLic : sys → licSet</td>
<td>Returns a set of licenses that contain the user request in a constraint permission set that is not labeled as Once or is labeled as Simple.</td>
</tr>
<tr>
<td>finalLic : sys → licSet</td>
<td>Returns a set of licenses that belong to possLic and the labels of the sublicenses where the request belongs are not labeled as Once or are labeled as Simple.</td>
</tr>
<tr>
<td>allowed : sys → permSet</td>
<td>Returns the set of permissions allowed by the installed licenses initially.</td>
</tr>
<tr>
<td>depleted : sys → permPSet</td>
<td>Returns the set of permissions lost after the satisfaction of a request</td>
</tr>
<tr>
<td>license : sys → Lsys</td>
<td>The projection operator</td>
</tr>
</tbody>
</table>

Table 7 Transition use4 in CafeOBJ

```c
op c-use4 : sys -> Bool

eq c-use4(S) = ((not(useReq(S) = null) and (not(best(S) = emptyLic)) and (type3?(labelS?(find3(useReq(S),best(S)))))) = once) and not(possLic(S) = emptyLic)) = 1) and (not(finalLic(S) = emptyLic)) and (not(allowedLic(S) = emptyLic)) .

ceq license(use4(S)) = depletedCP(license(S),
find3(useReq(S),best(S))) if c-use4(S) .

ceq useReq(use4(S)) = null if c-use4(S) .

ceq color(use4(S),P) = black if
(P = perm3?(useReq(S),find3(useReq(S),best(S))))
and (P /in allowed(use4(S))) and c-use4(S) .

ceq color(use4(S),P) = color(S,P) if not
(P = perm3?(useReq(S),find3(useReq(S),best(S))))
and (P /in allowed(use4(S))) and c-use4(S) .
```

5.3 Verification of the Safety Property

The safety property of Proposition 1 (that is equivalent to WML) was verified for the system described in the previous section, using the proof score method [12] for the CafeOBJ specification of the composite OTS. A safety or invariant [15] property in the OTS/CafeOBJ framework is a property that holds for all reachable states of an OTS. A state u is reachable for an OTS S =< O.I.T > if u ∈ I or for a reachable state u’, u =, τ(u’) for τ ∈ T. The first step of the verification is to define the property in CafeOBJ terms (usually in a module called INV that imports the OTS):

```c
module INV{

var S : sys
var P : perm

eq eq invl(S,P) = (P /in allowed(S)) and
(P /in depleted(S)) implies not (color(S,P) = white) .

open INV
red invl(init,p) .
close
```

To complete the verification we must show that the safety property is preserved by the inductive steps, i.e. by all of the transition functions. In a module, usually called ISTEP (importing INV) we define a generic operator to denote this. Next we must instantiate that operator for each transition and reduce it to true or false.

```c
module ISTEP{

ops s' s : -> sys
op p : -> perm
op r : -> reqErr
op l : -> lic

op istep1 : -> Bool

eq eq istep1 = invl(s,p) implies invl(s',p) .
```

When we instantiate s’ and ask CafeOBJ to reduce the inductive step, it is possible that it will return neither true nor false. Instead an expression might be returned that signifies it cannot reduce some of the equations required. The user must then select one of these equations and use it to split this case to two subcases, one denoting that the equation holds and one that it does not. The most typical example of this procedure is the effective condition of the transition specification of a DRM system that uses the proposed algorithm.

For example the definition of the observers that are changed when use4 is applied as well as its effective condition is given in table 7 †. In table 7, labelS? is an operator that returns the label of a constraint permission set, find3 an operator that returns the constraint permission set that the request belongs given a license. Similarly, find4 returns the sub-license that the request belongs to. Also, build2 returns the set of licenses that contain a permission matching the request, # returns the number of elements in a set and finally, perm3? returns the permission matching a request.

† the full specification can be found at cafeobjintua.wordpress.com/
rule. Usually CafeOBJ cannot reduce it to either true or false when the transition is applied to an arbitrary state. For example during the reduction of the request transition the case was split into two subcases (lines beginning with -- are comments ignored by CafeOBJ):

```plaintext
open ISTEP
op r : -> req .
  -- CASE SPLITTING
  eq c-request(r,s) = false .
  eq s' = request(r,s) .
  red istep1 .
  close

open ISTEP
op r : -> req .
  -- CASE SPLITTING
  eq c-request(r,s) = true .
  eq useReq(s) = null .
  eq belong??(r,licIns(s)) = true .
  eq (r = null) = false .
  eq best(s) = emptyLic .
  eq s' = request(r,s) .
  red istep1 .
  close
```

During the verification of a property, it is likely that we reach a case where CafeOBJ returns false. This means that either we have reached a state where the desired property does not hold, or the state that returned false is not reachable w.r.t. our OTS. In the first case we are presented with a counterexample, in the second case we must prove it is not reachable. This is done by using the equations defining this state to create a lemma that usually states that not all symmetrical subcases and false for this case. Due to the interactive nature of the proving procedure we were able to deduce that this case should not be reachable under our OTS and lemma inv2 defined below was used to discard the case using the following proof score.

```plaintext
eq inv2(P,R,L) = P /in buildPS1(find3(useReq(s),best(s))) implies
  belong3?(makeReq(P),find3(useReq(s),best(s)))) .
```


Nikolaos Triantafyllou
received his Diploma from the School of Mathematical and Physical Sciences of the National Technical University of Athens in 2008 and is a PhD student in the school of Electrical and Computer Engineering of the National Technical University of Athens since 2008. His research interests include DRM, semantic web, rule languages, formal methods, and algebraic specifications and verifications techniques.

Petros Stefaneas
is Lecturer at the National Technical University of Athens (Department of Mathematics, School of Mathematical and Physical Sciences). He has studied at the University of Athens, Oxford University (Programming Research Group) and the National Technical University of Athens. His research interests include logic and computation, formal metho-