PAPER

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 OMAs algorithm and the loss of execution rights. In section 4 we give an introduction to order 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 Funding 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 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 recourses 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 $Cons = \bigcup_{i=1}\{c_i | c_i \in Constraints\}$, where $Constraints = \{count, timed\text{-}count, Date\text{-}time; Interval, True, individual, system, accumulated\}$. 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 $Perm = \bigcup_{i=1}\{p_i | p_i \in Permissions\}$ where $Permissions = \{< p, cont > | p \in \{play, display, print, execute, export\}\}$ 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 rises 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*

Table 1 The OMA Allocation Algorithm

-Only the rights that are valid at the given time should be taken into account from the algorithm. -Rights that are not constrained should always be preferred over constrained rights. -Any right that includes a date-time constraint, and possibly others, should be preferred over rights that are constrained but do not have such a restriction. -If there exists more than one rights with a date-time constraint, the one with the further in the future end element should be preferred. -If there exist a choice between many rights and none of them contains a date-time constraint the one containing an interval constraint should be preferred if there exists such. Rights that contain a count constraint should be preferred after rights that contain a timed-count constraint.

Table 2 Minimal Loss Property

For user request r the algorithm selects license l, containing r_i (satisfying r), such that for all other licenses l' containing $r_j = r_i$ $remnants(ls, l', r) \subseteq remnants(ls, l, r)$.

after the execution of a right, although this is not explicitly stated in [3].

3.3 Loosing Rights and Minimal Loss Properties

There exist some cases that the algorithm fails its purpose. This is caused because the algorithm does not take into account the future requests of the user. A set of licenses *ls* allows a finite set of rights, $rights(ls) = \{r_1, \ldots, r_n\}$. After the execution of a request r, that is satisfied by a license $l \in ls$, the rights that are still available from ls are denoted as $remnants(ls; l; r)$.

Definition 2 (True Loss): A *loss of rights* is defined as the case where $rights(ls) \setminus remnants(ls; l; r) \neq \{\emptyset\}$. A *true loss of rights* on the other hand, is defined as the case where $rights(ls) \setminus remnants(ls; l; r) \supset \{r\}$. i.e., when more rights than the request become unavailable. Using this notation we define the Minimal Loss property of table 2[†].

With the set of licenses from our previous example we have that $rights(ls)$ = { *listen to song A,B,D* }. After a request to listen to song A the original algorithm will select license 1. Since this license apart from the *date-time* constraint is further constrained by a *count* constraint (that only allows one more execution) we have that $remnants(ls, L1, r)$ = { *listen to songs A, D* } the user loses the right of ever listening to song B. But if the algorithm had chosen L2 then we would have $remnants(ls, L2, r)$ = { *listen to songs A, B, D* }. It is easy to see that $remnants(ls, L1, r) \subset remnants(ls, L2, r)$. So the minimal loss property does not hold. This would not occur if the agent had selected L2. 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 $\exists l \in ls$ such that l satifsies r, and $rights(ls) \setminus remnants(ls; l; r) \subseteq \{r\}$ then the algorithm must select $l' \in ls$ containing r, such that for all other licenses $l'' \in ls$ that contain r it holds that $remnants(ls; l''; r) \subseteq remnants(ls; l'; r)$"* [††]. 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 subsort 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, Σ) where S is called the sort set and Σ is an $S^* \times S$-sorted family of functions $\{\Sigma_{w,s} | w \in S^* and s \in S\}$. An order sorted signature is a triple $(S, \leq, \Sigma)$ such that (S, Σ) is a many-sorted signature, $(S, \leq)$ is a poset, and the operations satisfy the following monotonicity condition; $\sigma \in \Sigma_{w1,s1} \cap \Sigma_{w2,s2}$ and $w1 \leq w2$ imply $s1 \leq s2$. Given a many-sorted signature an (S, Σ)-algebra A is a family of sets $\{A_s | s \in S\}$ called the carriers of A, together with a function $A_\sigma : A_w \to A_s$ for each $\sigma \in \Sigma_{w,s}$, where $A_w = A_{s1} \times \ldots \times A_{sn}, w = s1 \ldots sn$.

4.2 Labeling Licenses and the Proposed Algorithm

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

[†] since we are only interested in the existence of rights and not their quantity, the choice of a license of the form L ={*up to ten times play songs A or B*}, gives the set *rights(l)*={*play A, play B*}. Thus, its use does not cause a loss of rights; $remnants(l; l; r)$ = { *play A, play B* }

[††] 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[†††]. 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 $S3 = \{Count, Timed\text{-}count, Date\text{-}time, Interval, True\}$ the names of the various constraints allowed by OMA REL (we omit the constraints *individual* and *system* because they do not play any part in the decision made by the original algorithm). $S1 = \{Once, Many\}$ denoting whether the license will allow more than one execution of its permissions. Finally $S2 = \{Complex, Simple\}$, denoting if the license contains more than one permissions.

For example, a label $l = Once \times Simple \times Count$, 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: $once < many$ and $simple < complex$. 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 $Once \times Simple \times True$ from $Many \times Simple \times True$. 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.

[†††]as well as *accumulated*, but we equated it with a *count* once constraint

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

Each constraint permission set is a triplet, $CP =< Cons, Perm, label >$, with $CP' =< Cons, Perm >$, a OMA REL constraint permission set and *label* a label. These are retrieved by *Constraints(CP), Perm(CP) and label(CP) respectively.*

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

For example consider the sublicense: *sub-l* ={{*Once before the end of the month*} *either* {*up to ten times play songs A or B*} *or* {*once print document C*} }. Here, *Constraint(sub-l)*={*one time,before the end of the month* }. *CPS(sub-l)*={*CP1;CP2*} where C*P1* ={ *up to ten times, play either song A or B*} and *Constraints(CP1)* = {*up to ten times*}, *Perm(CP1)* = {*play song A, play song B*}. Likewise *CP2* ={*one time,print document C*}. *So here #(CPS (sub-l))* = *2.*

Definition 4 (Satisfiability): For a permission P and a request r we define that $sat(P, r) = true$ iff $P = r$. For a constraint permission set CP, we define that $sat(CP, r) = true$ iff there exists a permission $P \in CP$ such that $sat(P, r)$. For a sublicense $subl$, we define $sat(subl, r) = true$ iff $\exists$ $CP \in subl$ such that $sat(CP, r)$. For a license l $sat(l, r) =$

Table 3 The Proposed Algorithm

1. Input: a user request r, a set of licenses ls
2. Store all licenses $l \in ls$ for which sat(ls,r) holds. If only one such license exists return that license.
3. Else, store all licenses, l, from step 2, for which it holds that $sat(subl, r) \wedge subl \in l \wedge \neg(times(label(subl)) = once \wedge comp(label(subl)) = complex)$.
4. Search the licenses from step 3, and store all licenses l, such that $sat(CP, r) \wedge CP \in subl \wedge subl \in l$ for which $\neg(times(label(CP)) = once \wedge compl(label(CP)) = complex)$.
5. If the result of step 4 is not the empty set, then run the original OMA allocation algorithm on only those licenses and return the result.
6. Else prompt the user to select a license from step 2.

true iff $\exists\, subl \in l$ such that $sat(subl, r)$. Finally for a set of licenses ls, $sat(ls, r) = true$ iff $\exists\, l \in ls$ such that $sat(l, r)$. Also for label $l = complexity \times times \times constraint$, we define $times(l) = times$, $comp(l) = complexity$ and $cons(l) = constraint$.

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 $subl$, such that either $label(subl) = Once \times Complex \times Constraint$ or $label(CP) = Once \times Complex \times Constraint$. 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 $sat(l, r)$ 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 $sat(l, r)$ 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 $l \in ls$ such that $sat(l, r) = true$, then there exist three cases under which the selection of a license satisfies the WML property:

1. Only one license exists such that $sat(l, r)$.
2. All the licenses for which $sat(l, r)$ cause a true loss of rights.
3. There exists $l \in ls$ such that $sat(l, r)$ 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, *remants(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 ls we define that $\forall p \in rights(ls)$ initially $color(p) = white$. 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 $rights(ls) \setminus remnants(ls, l, r)$ to become unavailable, we define that the color of $p \in \{rights(ls) \setminus remnants(ls, l, r)\}$ becomes *black*: if l is the only license such that $sat(l, r)$ or, if all the other licenses l' for which $sat(l', r)$ 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 $depleted(ls, r^*)$ the set of permissions lost by a sequence of satisfied requests $r^* = r_0, \ldots, r_{n-1}$ from the set ls. Formally, $depleted(ls, r^*) = \{rights(ls) \setminus remnants(ls; l_0; r_0)\} \bigcup \ldots \bigcup \{rights(ls^{(n-1)}) \setminus remnants(ls^{(n-1)}; l_{n-1}; r_{n-1})\}$, where l_i is the license chosen to satisfy the request r_i, Also $ls^{(i)}$ is the set of licenses after the satisfactions of i-1 requests.

Proposition 1: The WML is equivalent to the safety property: $(p \in rights(ls) \wedge p \in depleted(ls, r^*)) \rightarrow \neg(color(p) = white)$

Proof(Sketch) If the safety property does not hold after an arbitrary number of requests n, then $\exists p \in rights(ls) \wedge p \in depleted(ls, r^*)$ such that $color(p) = white$. But, $p \in depleted(ls, r^*)$ implies that $\exists i \leq n$, $l_i \in ls$, such that l_i was selected for a request r_i and some part of l_i got depleted. Also since, $sat(l_i, r_i)$ then $r_i, p \in \{rights(ls^{(i)}) \setminus remannts(ls^{(i)}; l_i; r_i)\}$. However, $color(p) = white$, so from the coloring definition $p \neq r_i \wedge \exists l' \in ls^{(i)}$ such that $sat(l', r_i) \wedge l' \neq l_i \wedge$ $\{rights(ls^{(i)}) \setminus remnants(ls^{(i)}; l'; r_i)\} \subseteq \{r_i\}$. So, $\{rights(ls^{(i)}) \setminus remnants(ls^{(i)}; l'; r_i)\} \subset \{rights(ls^{(i)}) \setminus remnants(ls^{(i)}; l_i; r_i)\}$ which implies that $remnants(ls^{(i)}, l', r_i) \supset remnants(ls^{(i)}; l_i; r_i)$, i.e., the WML property does not hold. If the WML property does not hold, then for some request r_i, $\exists l \in ls^{(i)}$ such that $sat(l, r_i) \wedge \{rights(ls^{(i)}) \setminus remnants(ls^{(i)}; l; r_i)\} \subseteq \{r_i\}$ and the algorithm selects $l' \in ls^{(i)}$ such that $\exists l'' \in ls^{(i)} \wedge sat(l'', r_i) \wedge remnants(ls^{(i)}; l''; r_i) \supset remnants(ls^{(i)}; l'; r_i)$. This implies that $\{rights(ls) \setminus remnants(ls, l'', r_i)\} \subset \{rights(ls) \setminus remnants(ls, l', r_i)\}$. Because $sat(l', r_i)$, then $r_i \in \{rights(ls^{(i)}) \setminus remnants(ls^{(i)}; l'; r_i)\}$. Also, $r_i \in \{rights(ls^{(i)}) \setminus remnants(ls^{(i)}; l''; r_i)\}$. But, $\{rights(ls^{(i)}) \setminus$

$remnants(ls^{(i)}; l''; r_i)\} \subset \{rights(ls^{(i)}) \setminus remnants(ls^{(i)}; l'; r_i)\}$. So there exists $p \in \{rights(ls^{(i)}) \setminus remnants(ls^{(i)}, l', r_i)\}$, $p \neq r_i$. However due to the existence of l, from the definition of the coloring we have that $color(p) = white$, so the safety property does not hold.

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 $S =< O, I, T >$ 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 a state of the system and possibly a series of other data type values (visible sorts) and returns a value of a data type that is characteristic to that state of the system. Given an OTS S and two states $u_1, u_2 \in Y$, the equivalence $(u_1 =_s u_2)$ between them w.r.t. S is defined as $\forall o \in O, o(u_1) = o(u_2)$. The previous equality creates the equivalence classes, $Y/ =_S$, on the states of an OTS. Finally, T is the set of transition, conditional functions (or actions). Each transition takes as input a state of the system (hidden sort) and possibly a set of data-type values and returns a new state of the system. If $\tau \in T$ then $\tau(u_1) =_s \tau(u_2)$ for each $u_1, u_2 \in Y/ =_S$. For each $u \in Y$, $\tau(u)$ is called the successor state of u. The condition c_τ of τ is called the effective condition. Also, for each $u \in Y$,$\tau(u) = u$ if $\neg c_\tau(u)$.

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, where $k = i_1, \ldots, i_m$, and a hidden sort H, the observation operator denoting $o_{i1,\ldots,im}$ is declared as follows; bop o: $V_{i1} \ldots V_{im} H \rightarrow V$. Any state in I is denoted by a constant, say init, which is declared as: op init: $\rightarrow$ H. A transition $\tau_{j1,\ldots,jn} \in T$ is denoted by a CafeOBJ action operator; bop $\tau : V_{j1} \ldots V_{jm}\, H \rightarrow H$, with V_k a visible sort corresponding to the data type D_k and $k = j_1, \ldots, j_n$. Each transition is defined by stating the value returned by each observer in the successor state, when $\tau_{j1,\ldots,jn}$ is applied in a state u when $c\text{-}\tau_{j1,\ldots,jn}(u)$ holds. The value returned by $o_{i1,\ldots,im}$ is not changed if $\tau_{j1,\ldots,jn}$ is applied in a state u such that $\neg c\text{-}\tau_{j1,\ldots,jn}(u)$. 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 datatypes (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 `]` (`*[`and `]*`). 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.

Table 4 CafeOBJ module defining the ADT of Labels

```
mod* Label{ [type1 , type2 ,type3 < label]
op _=_ : label label -> Bool {comm}
ops simple complex  : -> type1
ops count datetime true : -> type2
ops once many : -> type3
op _ & _ & _ : type1 type2 type3 -> label
op type1?_ : label -> type1
op type2?_ : label -> type2
op type3?_ : label -> type3
var t1 : type1
var t2 : type2
var t3 : type3
eq type1?(t1 &  t2 & t3) = t1 .
eq type2?(t1 & t2 &  t3) = t2 .
eq type3?(t1 & t2 &  t3) = t3 .
eq (t1 = t1 ) = true .
eq (t2 = t2 ) = true .
eq (t3 = t3 ) = true . }
```

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 S_1, S_2 and S_3 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

Table 5 Observers and Transitions of the component OTS

Observers	
Signature	Description
validsublic : *Lsys subLic* → *Bool*	Returns true if the constraints of the sublicense are met at the given state
validCP : *Lsys cPerm* → *Bool*	Returns true if the constraints of the constraint permission set are met at the given state
installed : *Lsys* → *licSet*	Returns the set of installed licenses
Transitions	
depleteSL : *Lsys subLic* → *Lsys*	Models the transition that occurs when the whole of the given license becomes depleted
depleteCP : *Lsys cPerm* → *Lsys*	Models the transition that occurs when a constraint permission set of a sublicense becomes depleted.

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 sate 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:

```
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

Signature	Description
$licIns : sys \rightarrow licSet$	Returns the set of installed licenses
$useReq : sys \rightarrow reqErr$	Returns the request of the user at the given state, if it exists. Else returns an error constant
$best : sys \rightarrow lic$	Returns the license the algorithm selects for the user request
$color : sys\ perm \rightarrow color$	Returns the color of the given permission
$possLic : sys \rightarrow licSet$	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.
$finalLic : sys \rightarrow licSet$	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.
$allowed : sys \rightarrow permSet$	Returns the set of permissions allowed by the installed licenses initially.
$depleted : sys \rightarrow permPSet$	Returns the set of permissions lost after the satisfaction of a request
$license : sys \rightarrow Lsys$	The projection operatro

Table 7 Transition use4 in CafeOBJ

```
op c-use4 : sys -> Bool
eq c-use4(S) =  ((not(useReq(S) = null) and
   (not (best(S) = emptyLic))) and (type3?(labelCP?(
   find3(useReq(S),best(S)))) = once))and not(type3?(
   label?(find4(useReq(S),best(S)))) = once) and ( not
   (# build2(useReq(S),licIns(S),license(S))== 1) and
   (not(possLic(S) = emptyLic) and  not(finalLic(S) =
   emptyLic))) .
ceq license(use4(S)) = depleteCP(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) .
```

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, `labelCP?` 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 cafeobjntua.wordpress.com/

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 $u \in I$ or for a reachable state u', $u =_s \tau(u')$ for $\tau \in T$. The first step of the verification is to define the property in CafeOBJ terms (usually in a module called INV that imports the OTS):

```
module INV{
pr(OTS)
op inv1 : sys perm  -> Bool
var S : sys
var P : perm
eq eq inv1(S,P) =  (P /in allowed(S)) and
(P /in depleted(S)) implies not (color(S,P) = white)  .
```

The next step is to show that the property holds for an arbitrary initial state, the inductive base. This is achieved by opening the module that defines the invariant and asking the CafeOBJ system to reduce the given expression using the equations that constitute the specification to either true or false, denoting that the property holds or not respectively. This is done using the `red` command of CafeOBJ. Here true was returned for the following proof passage, where `p` is a CafeOBJ constant denoting an arbitrary permission:

```
open INV
red  inv1(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.

```
module ISTEP{
pr(INV)
ops s' s : -> sys
op p : -> perm
op r : -> reqErr
op l : -> lic
op istep1 :  -> Bool
eq istep1 = inv1(s,p) implies inv1(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 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):

```
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 belong7?(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 of those equations can hold simultaneously. Attention must be taken though as those lemmas must now be verified as well as new invariant properties. During the verification of the safety property the state that is defined by the following equations was reached: $c - use1(s) \wedge (p \in allowed(s)) \wedge \neg(p = perm3?(useReq(s), find3(useReq(s), best(s)))) \wedge (belong3?(makeReq(p), find3(useReq(s), best(s)))) \wedge$ (color (s , p) = white) $\wedge \neg p \in depleted(s) \wedge \neg(p \in buildPS1$ (find3 (useReq (s) , best(s)))) and CafeOBJ returned true for 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.

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

open ISTEP
eq s' = use1(s) .
eq (useReq(s) = null) = false .
eq (best(s) = emptyLic) = false .
eq type3?(labelCP?(find3(useReq(s),best(s))))= once .
eq (type3?(label?(find4(useReq(s),best(s))))= once)= false .
eq possLic(s) = emptyLic .
eq p /in allowed(s) = true .
eq (p= perm3?(useReq(s),find3(useReq(s),best(s))))= false .
eq (belong3?(makeReq(p),find3(useReq(s),best(s))))= false .
eq color(s,p) = white   .
eq p /in depleted(s) = false .
eq p /in buildPS1(find3(useReq(s),best(s)))= true .
  red inv2(p,useReq(s),best(s)) implies istep1 .
close
```

For the complete proof four lemmas were required to discard cases that returned false and for the verification of those lemmas thirteen new lemmas were needed. Those lemmas were verified as well, which concludes the verification that the proposed algorithm satisfies the WML property†.

6. Conclusions

We redesigned the OMA rights allocation Algorithm by introducing labels on some parts of the licenses that allow the algorithm to enjoy a monotonic behavior when such behavior is desired. Non-monotonic behavior may only occur when all licenses that satisfy the request cause a loss. Then the user is prompted to ensure that the rights lost will be the ones valued less by him, which in our opinion is preferable to full monotonicity. Such an algorithm would be NP-complete [7], but we shift the extra computational cost to the creation of the licenses, which are produced by computers with much higher computational power. Tools that take as input a license written in OMA REL and produce labeled licenses will be easy to implement. To prove that the algorithm behaves as desired, an Observational Transition System (OTS) specification written in CafeOBJ terms of a DRM system using the suggested algorithm was presented. This allowed the verification of a property that expresses the desired behavior of such an algorithm. With small alterations the same algorithm could be used to solve the problem of monotonically allocating a user request to a license in other DRM enviroments as well, since their licenses share common attributes. This work is in the spirit of our previous efforts in the formal verification of mobile systems and algorithms [8, 9 and 10].

References

[1] Open Mobile Alliance, OMA home page, 2010
[2] Open Mobile Alliance Digital Rights Management 2.1, 2008.
[3] Open Mobile Alliance, DRM Rights Expression Language 2.1, 2010.
[4] R. Iannella, Open Digital Rights Ranguage version 1.1, 2002.
[5] ContentGuard, Xrml 2.0 technical overview version 1.0, 2007.
[6] Rightscom, The MPEG-21 rights expression language, 2007.
[7] A. Barth, J. C. Mitchell, Managing digital rights using linear logic, 21st Symposium on Logic in Computer Science, 2006.
[8] I. Ouranos, P. Stefaneas, P. Frangos, An algebraic framework for

† the full proof score can be found at cafeobjntua.wordpress.com

modeling of mobile systems, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science, volume E90-A, 2007.

[9] N. Triantafyllou, I. Ouranos, P. Stefaneas, Algebraic specifications for oma rel licenses, IEEE International Conference on Wireless and Mobile Computing, Networking and Communication, 2009.

[10] K. Ksystra K, P. Stefaneas, N, Triantafyllou, I. Ouranos, An algebraic specification for the mpeg-2 encoding algorithm, South-East European Workshop on Formal Methods, 2009.

[11] J. A. Goguen, Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations, Theoretical Computer Science, pp. 217-273, 1992.

[12] K. Ogata, K. Futatsugi, Some Tips on Writing Proof Scores in the OTS/CafeOBJ Method, Algebra, Meaning, and Computation, Lecture Notes in Computer Science, Volume 4060, pp. 596-615, 2006.

[13] R. Diaconescu, K. Futatsugi, CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification, World Scientific Pub Co Inc, 1998.

[14] R. Diaconescu, Behavioral specification for hierarchical object composition, Journal of Theoretical Computer Science - Formal methods for components and objects, Volume 343 Issue 3, 2005.

[15] K. Ogata, K. Futatsugi, Proof Score Approach to Verification of Liveness Properties. IEICE Transactions on Fundamentals of Software and Theory of Programs, Volume 91-D, pp. 2804-2817, 2008.

[16] G. Bruns, M. Huth, Access-Control Policies via Belnap Logic: Effective and Efficient Composition and Analysis, 21st IEEE Computer Security Foundations Symposium, pp. 163-176, 2008.

[17] A. K. Bandara, E. C. Lupu, A. Russo, Using Event Calculus to Formalise Policy Specification and Analysis, 4th IEEE International Workshop on Policies for Distributed Systems and Networks, pp. 26-40, 2003.

[18] A. Sasturkar, P. Yang, S. D. Stoller, C.R. Ramakrishnan, Policy Analysis for Administrative Role Based Access Control, 19th IEEE Computer Security Foundations Workshop, pp.124-138, 2006.

[19] D. J. Dougherty, K. Fisler, S. Krishnamurthi, Specifying and Reasoning about Dynamic Access-Control Policies, Third international joint conference on Automated Reasoning, pp. 632-646, 2006.

[20] M.Y. Becker, Specification and Analysis of Dynamic Authorisation Policies, 22nd IEEE Computer Security Foundations Symposium, pp. 203 - 217, 2009.

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 methods, web and philosophy, information society and new media.

Panayiotis Frangos received the Diploma in Electrical and Computer Engineering from National Technical University of Athens, Greece, in 1983, and the M.Sc. and Ph.D. degrees from the Moore School of Electrical Engineering, University of Pennsylvania, USA, in 1985 and 1986 respectively. Since 1989 he has been a Faculty Member at the School of Electrical and Computer Engineering, National Technical University of Athens, currently a full Professor (since 2000). His research interests include planning of mobile radio communication systems and radar signal processing.

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 Funding Program: THALIS