Formal Analysis of Access Control Policies

Jeremy W. Bryans
School of Computing Science
Newcastle University
UK

Abstract

We present a formal (model-based) approach to describing and analysing access control policies. This approach allows us to evaluate access requests against policies, compare versions of policies with each other and check policies for internal consistency. Access control policies are described using VDM, a state-based formal modelling language. Policy descriptions are concise and may be easily manipulated. The structure of the VDM description is derived from the OASIS standard access control policy language XACML. It is therefore straightforward to translate between XACML policies and their corresponding models.

1 Introduction

In many market sectors, business affiliations are increasingly volatile. Use of technologies such as web services can allow companies to rapidly join forces in *virtual organisations* (VOs), tailored to meet specific market opportunities. The structure of these VOs may be constantly changing, as companies join to provide necessary new skills, or members complete their part of a task and withdraw from the VO.

In order for the VO to operate efficiently, each partner must expose relevant business information to the other members. This may be done using a centralised database, to which all partners contribute, or it may be done by each partner retaining their own business information and allowing other partners access as necessary.

Parts of the overall access control policy may be developed and maintained by each partner, and it may be that no one person is responsible for the final resultant policy. Either way, the relevant access control policies need to be updated carefully, to ensure that legitimate accesses are not blocked (which would slow down the operation of the VO) and that illegitimate accesses are not permitted. In this environment there is a clear need to understand the behaviour of the policy and in particular to understand how changes to small parts of the policy will affect the behaviour of the overall policy.

The GOLD project [14] has been researching into enabling technology to support the formation, operation and termination of VOs. In this paper we present a formal way of analysing access control policies which have been written in XACML. We translate policies into the formal modelling language VDM. Simple operations within this formal context permit us to test the behaviour of these policies, compare policies with each other and check the internal consistency of policies. The structure of the XACML is preserving and therefore a faulty policy can be fixed within the VDM framework and translated back to XACML.

VDM is a model oriented formal method with good tool support: VDMTools [3].

In Section 2 we give an overview of XACML, the OASIS access control language. Section 3 gives a VDM description of the data types and algorithms common to all access control policies. Section 4 shows how these data types may be populated to describe a specific policy. In Section 5 we show how the VDMTools framework may be used to test policies, and

compare them with each other.

2 XACML and access control

XACML [13] is the OASIS standard for access control policies. It provides a language for describing access control policies, and a language for interrogating these policies, to ask the policy if a given action should be allowed.

A simplified description of the behaviour of an XACML policy is as follows. An XACML policy has an associated *Policy Decision Point* (PDP) and a *Policy Enforcement Point* (PEP) (See Figure 1.) Any access requests by a user are intercepted by the PEP. Requests are formed in the XACML request language by the PEP and sent to the PDP. The PDP evaluates the request with respect to the access control policy. This response is then enforced by the PEP.

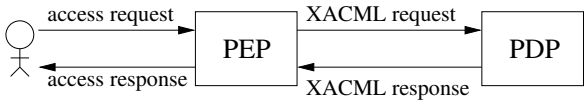

Figure 1: XACML overview.

When a request is made, the PDP will return exactly one of:

- PERMIT: if the subject is permitted to perform the action on the resource,
- DENY: if the subject is not permitted to perform the action on the resource, or
- NOTAPPLICABLE: if the request cannot be answered by the service.

The full language also contains the response INDETERMINATE. This is triggered by an error in evaluating the conditional part of the rule. Since we assume rules to be environment independent, evaluating the rules will not require evaluation of any conditional statements. We therefore do not use INDETERMINATE.

A full XACML request includes a set of *subjects* (e.g. a user, the machine the user is on, and the applet the user is running could all be subjects with varying access rights) to perform a set of *actions* (e.g. read, write, copy) on a set of *resources* (e.g. a file or a disk) within an environment (e.g. during work hours or from a secure machine). It may also contain a *condition* on the environment, to be evaluated when the request is made. We will assume that requests (and later, rules) are environment independent, and omit the condition and environment components.

We will assume that a PDP contains a set of *Policies*, each of which contain a set of *Rules*[1]. Rules in XACML contain a *target* (which further contains sets of resources, subjects and actions) and an *effect* (permit or deny). If the target of a rule matches a request, the rule will return its *effect*. If the target does not match the rule NOTAPPLICABLE is returned.

As well as a set of rules, a policy contains a *rule combining algorithm*. Like rules, they also contain a target. All rules within a policy are evaluated by the PDP. The results are then combined using the rule combining algorithm, and a single effect is returned.

The PDP evaluates each policy, and combines the results using a *policy combining algorithm*. The final effect is then returned to the PEP to be enforced. If PERMIT is returned then the PEP will permit access, any other response will cause access to be denied.

3 VDM

VDM is a model oriented formal method incorporating a modelling or specification language (VDM-SL) with formal semantics [1], a proof theory [2] and refinement rules [11].

A VDM-SL model is based on a set of data type definitions, which may include invariants (arbitrary predicates characterising properties shared by all members of the type). Functionality is described in terms of functions over the types, or operations which may have side effects on distinguished state variables. Functions and operations may be restricted by preconditions, and may be defined in an explicit algorithmic style or implicitly in terms of postcondi-

[1] Strictly, it contains a set of policy sets, each of which contain a set of policies. Policies then contain a set of rules. Extending our model to include this additional complexity would be straightforward.

tions. The models presented in this paper use only explicitly-defined functions. We remain within a fully executable subset of the modelling language, allowing our models of XACML policies to be analysed using an interpreter.

VDM has strong tool support. The CSK VDM-Tools [3] include syntax and type checking, an interpreter for executable models, test scripting and coverage analysis facilities, program code generation and pretty-printing. These have the potential to form a platform for tools specifically tailored to the analysis of access control policies in an XACML framework.

An access control policy is essentially a complex data type, and the XACML standard is a description of the evaluation functions with respect to these data types. Thus VDM, with its focus on datatypes and functionality, is a suitable language to describe access control policies.

This section describes in detail the data types and functionality of the Policy Decision Point. The description is presented in VDM. We impose the simplifications mentioned in the previous section. In particular, we limit targets to sets of subjects, actions and resources, and exclude consideration of the environment. This means we only consider the effects PERMIT, DENY and NOTAPPLICABLE.

Elements of the type *PDP* are pairs. The first field has label *policies* and contains a set of the elements of the type *Policy*. The second field contains one value of the type *CombAlg*, defined immediately below.

$$\begin{array}{rrcl} PDP :: & policies & : & Policy\text{-}\mathbf{set} \\ & policyCombAlg & : & CombAlg \end{array}$$

$$CombAlg = \textsc{DenyOverrides} \mid \textsc{PermitOverrides}$$

DENYOVERRIDES and PERMITOVERRIDES are enumerated values. They will act as pointers to the appropriate algorithms, which are defined later. Other possible combining algorithms are given in [13] but for simplicity we will model only these two here.

A policy contains a *target* (the sets of *subjects*, *resources* and *actions* to which it applies), a set of *rules*, and a *rule combining algorithm*.

$$\begin{array}{rrcl} Policy :: & target & : & Target \\ & rules & : & Rule\text{-}\mathbf{set} \\ & ruleCombAlg & : & CombAlg \end{array}$$

$$\begin{array}{rrcl} Target :: & subjects & : & Subject\text{-}\mathbf{set} \\ & resources & : & Resource\text{-}\mathbf{set} \\ & actions & : & Action\text{-}\mathbf{set} \end{array}$$

Each rule has a target and an *effect*. If a request corresponds to the rule target then the rule effect is returned. The brackets [..] denote that the target component may be empty. In this case, the default value is the target of the enclosing policy.

$$\begin{array}{rrcl} Rule :: & target & : & [Target] \\ & effect & : & Effect \end{array}$$

The effect of the rule can be PERMIT, DENY, or NOTAPPLICABLE. These are modelled as enumerated values.

$$Effect = \textsc{Permit} \mid \textsc{Deny} \mid \textsc{NotApplicable}$$

Requests are simply *target*s:

$$Request :: target : Target$$

The functionality of a PDP is captured in the following set of functions. We begin with the function *targetmatch*, which takes two targets and returns true if the targets have some subject, resource and action in common.

$$targetmatch : Target \times Target \to Bool$$

$$\begin{array}{l} targetmatch(target1, target2) \triangleq \\ \quad (target1.subjects \cap target2.subjects) \neq \{\} \wedge \\ \quad (target1.resources \cap target2.resources) \neq \{\} \wedge \\ \quad (target1.actions \cap target2.actions) \neq \{\} \end{array}$$

A request is evaluated against a rule using *evaluateRule*. If the rule target is NULL the targets are assumed to match, since the parent policy target must match. If the targets match the effect of the rule is returned, otherwise NOTAPPLICABLE is returned.

$$evaluateRule : Request \times Rule \to Effect$$

$$\begin{array}{l} evaluateRule(req, rule) \triangleq \\ \quad \textbf{if } rule.target = \textsc{Null} \\ \quad \textbf{then } rule.effect \\ \quad \textbf{else if } targetmatch(req.target, rule.target) \\ \qquad \textbf{then } rule.effect \\ \qquad \textbf{else } \textsc{NotApplicable} \end{array}$$

A policy is invoked if its target matches the request. It then evaluates all its rules with respect to a request, and combines the returned effects using its *rule combining algorithm*. In the following DENYOVERRIDES is abbreviated to D-OVER, and PERMITOVERRIDES to P-OVER.

evalPol : *Request* × *Policy* → *Effect*

evalPol(*req*, *pol*) $\triangleq$
  if *targetmatch*(*req.target*, *pol.target*)
  then if (*pol.ruleCombAlg* = D-OVER)
    then *evalRules-DO*(*req*, *pol.rules*)
    else if (*pol.ruleCombAlg* = P-OVER)
      then *evalRules-PO*(*req*, *pol.rules*)
      else NOTAPPLICABLE
  else NOTAPPLICABLE

The deny overrides algorithm is implemented as

evalRules-DO : *Request* × *Rule*-**set** → *Effect*

evalRules-DO(*req*, *rs*) $\triangleq$
  if $\exists r \in rs \cdot evaluateRule(req, r) =$ DENY
  then DENY
  else if $\exists r \in rs \cdot evaluateRule(req, r) =$ PERMIT
    then PERMIT
    else NOTAPPLICABLE

If any rule in the policy evaluates to DENY, the policy will return DENY. Otherwise, if any rule in the policy evaluates to PERMIT, the policy will return PERMIT. If no rules evaluate to either PERMIT or DENY, the policy will return NOTAPPLICABLE.

The permit overrides rule combining algorithm (omitted) is identical in structure, but a single PERMIT overrides any number of DENYs.

The evaluation of the PDP and its rule combining algorithms has an equivalent structure to the policy evaluation functions already presented.

evaluatePDP : *Request* × *PDP* → *Effect*

evaluatePDP(*req*, *pdp*) $\triangleq$
  if (*pdp.policyCombAlg* = D-OVER)
  then *evalPDP-DO*(*req*, *pdp*)
  else if (*pdp.policyCombAlg* = P-OVER)
    then *evaluatePDP-PO*(*req*, *pdp*)
    else NOTAPPLICABLE

evaluatePDP-DO : *Request* × *PDP* → *Effect*

evaluatePDP-DO(*req*, *pdp*) $\triangleq$
  if $\exists p \in pdp.policies \cdot evalPol(req, p) =$ DENY
  then DENY
  else if $\exists p \in pdp.policies \cdot evalPol(req, p) =$ PERMIT
    then PERMIT
    else NOTAPPLICABLE

The above functions and data types are generic. Any XACML policy in VDM will use these functions. In the next section we show how to instantiate this generic framework with a particular policy.

4 An example policy

In this section we present the initial requirements on an example policy and instantiate our abstract framework with a policy aimed at implementing these requirements. In Section 5 we show how we can use the testing capabilities of VDMTools [3] to find errors in these policies.

This example is taken from [5], and describes the access control requirements of a university database which contains student grades. There are two types of resources (*internal* and *external* grades), three types of *actions* (*assign*, *view* and *receive*), and a number of subjects, who may hold the roles *Faculty* or *Student*. We therefore define

Action = ASSIGN | VIEW | RECEIVE

Resource = INT | EXT

Subjects are enumerated values,

Subject = ANNE | BOB | CHARLIE | DAVE

and we populate the Student and Faculty sets as

Student : *Subject*-**set** = {ANNE, BOB}
Faculty : *Subject*-**set** = {BOB, CHARLIE}

so Bob is a member of both sets. In practice, an access control request is evaluated on the basis of certain attributes of the subject. What we are therefore saying here is that the system, if asked, can produce evidence of Anne and Bob being students, and of Bob and Charlie being faculty members.

Informally, we can argue that populating the student and faculty sets so sparsely is adequate for testing purposes. All rules we go on to define apply to roles, rather than to individuals, so we only need one representative subject holding each possible role combination.

The properties to be upheld by the policy are

1. No students can assign external grades,
2. All faculty members can assign both internal and external grades, and
3. No combinations of roles exist such that a user with those roles can both receive and assign external grades.

Our initial policy (following the example in [5]) is

> *Requests for students to receive external grades, and for faculty to assign and view internal and external grades, will succeed.*

Implementing this policy naïvely leads to the following two rules, which together will form our initial (flawed) policy. Students may receive external grades,[2]

$$StudentRule : Rule = ((Student, \text{EXT}, \text{RECEIVE}), \text{PERMIT})$$

and faculty members may assign and view both internal and external grades.

$$FacultyRule : Rule = ((Faculty, \{\text{INT}, \text{EXT}\}, \{\text{ASSIGN}, \text{VIEW}\}), \text{PERMIT})$$

The policy combines these two rules using the PERMITOVERRIDES algorithm. The target of the policy is all requests from students and faculty members.

$$PolicyStuFac : Policy = ((Student \cup Faculty, \{\text{INT}, \text{EXT}\}, \{\text{ASSIGN}, \text{VIEW}, \text{RECEIVE}\}), \{StudentRule, FacultyRule\}, \text{P-OVER})$$

[2]The correct VDM description is

$$StudentRule : Rule = mk\_Rule(mk\_Target(Student, \{\text{EXT}\}, \{\text{RECEIVE}\}), \text{PERMIT})$$

For ease of reading, we omit the $mk\_$ constructs and brackets around singleton sets.

In fact, this policy would have the same behaviour if the two rules were combined using the DENYOVERRIDES algorithm. This is because both rules have the effect PERMIT.

The PDP is a collection of policies; in this case only one.

$$PDPone : PDP = (PolicyStuFac, \text{D-OVER})$$

We use the deny overrides algorithm here, but in this case it has the same behaviour as permit overrides, because there is only one policy. In the next section we examine this and other specifications using VDM-Tools.

5 Testing the specification

The VDM toolset VDMTools described in [7] provides considerable support for testing VDM specifications, including syntax and type checking, testing and debugging. Individual tests can be run at the command line in the interpreter. The test arguments can be also read from pre prepared files, and scripts are available to allow large batches of tests to be performed.

A systematic approach to testing requires that we have some form of oracle against which to judge the (in)correctness of test outcomes. This may be a manually-generated list of outcomes (where we wish to assess correctness against expectations) or an executable specification (if we wish to assess correctness against the specification). In this section we use a list of expected results as our oracle. Section 5.1 uses one version of a VDM-SL specification as an oracle against another version.

Tests on *PDPone* are made by forming requests and evaluating the PDP with respect to these requests, using the function *evaluatePDP* from Section 3.

Below we show four example requests and the results from *PDPone*.

Request	Result from *PDPone*
(ANNE, EXT, ASSIGN)	NOTAPPLICABLE
(BOB, EXT, ASSIGN)	PERMIT
(CHARLIE, EXT, ASSIGN)	PERMIT
(DAVE, EXT, ASSIGN)	NOTAPPLICABLE

In the first test, *PDPone* returns NotApplicable when user Anne (a student) asks to assign an external grade. This is because there is no rule which specifically covers this situation. The PEP denies any request which is not explicitly permitted, and so access is denied.

The second test points out an error, because Bob (who is both a faculty member and a student) is allowed to assign external grades, in violation of rule one, which states that no student may assign external grades. This policy has been written with an implicit assumption that the sets student and faculty are disjoint. Constraining these sets to be disjoint when we populate them allows us to reflect this assumption. In practice this constraint would have to be enforced at the point where roles are assigned to individuals rather than within the PDP.

The third test is permitted, as expected, since Charlie is a member of faculty, and the fourth test returns NotApplicable, because Dave is not a student or a faculty member.

Multiple requests

The policy as defined can be broken if multiple access control requests are combined into one XACML request. For example the request below (identified in [5])

(Anne, {Ext}, {Assign, Receive})

is permitted. As pointed out in [5], this breaks the first property, because Anne (a student) is piggybacking an illegal request (assigning an external grade) on a legal one (receiving an external grade). In future, therefore, we make the assumption that the PEP only submits requests that contain singleton sets. Given this assumption, we can limit the test cases we need to consider to those containing only single subjects, actions and resources.

5.1 Comparing specifications of PDPs

We now suppose (following [5]) that teaching assistants (TAs) are to be employed to help with the internal marking. They are not, however, allowed to help with external marking. A careless implementation, that merely included the names of the TAs as faculty members, would overlook the fact that students are often employed as TAs.

A more robust implementation, that makes TAs a separate role and develops rules specific for them, is given below. Note that in *TArule*2, TAs are explicitly forbidden to assign or view external grades; their role is restricted to dealing with the internal grades.

$$TArule1 : Rule = ((TA, \{\text{Int}\}, \{\text{Assign}, \text{View}\}), \text{Permit})$$

$$TArule2 : Rule = ((TA, \{\text{Ext}\}, \{\text{Assign}, \text{View}\}), \text{Deny})$$

The rules are combined into a (TA-specific) policy

$$PolicyTA : Policy = ((TA, \{\text{Int}, \text{Ext}\}, \{\text{Assign}, \text{View}, \text{Receive}\}), \{TArule1, TArule2\}, \text{PermitOverrides})$$

which is combined with *PolicyStuFac* from Section 4 to give a new Policy Decision Point:

$$PDPtwo : PDP = (\{PolicyTA, PolicyStuFac\}, \text{D-Over})$$

This new PDP can of course be tested independently, but it can also be compared with the previous one. We do this with respect to a test suite. Because the policies are small, this test suite can be comprehensive. We now populate the roles as

$$\begin{array}{lcl} Student : Subject\text{-}\mathbf{set} & = & \{\text{Anne}, \text{Bob}\} \\ Faculty : Subject\text{-}\mathbf{set} & = & \{\text{Charlie}\} \\ TA : Subject\text{-}\mathbf{set} & = & \{\text{Bob}, \text{Dave}\} \end{array}$$

taking care that there is a person holding each possible combination of roles that we allow. Every request that each person can make is considered against each PDP. This is easily automated using a simple shell script. The observed changes are summarised below.

Request	*PDPone*	*PDPtwo*
(Bob, Int, Assign)	NotApp	Permit
(Bob, Int, View)	NotApp	Permit
(Bob, Ext, Assign)	NotApp	Deny
(Bob, Ext, View)	NotApp	Deny
(Dave, Int, Assign)	NotApp	Permit
(Dave, Int, View)	NotApp	Permit
(Dave, Ext, Assign)	NotApp	Deny
(Dave, Ext, View)	NotApp	Deny

As a *TA*, Bob's privileges now include assigning and viewing internal grades, as well as all the privileges he has as a student. Everything else is now explicitly denied.

All requests from Dave, who is a now TA but not a student, are judged NotApplicable (and consequently denied) by the first policy, but the second policy allows him to view and assign internal grades. It explicitly forbids him to assign or view external grades.

Internal consistency of a PDP

We consider a set of rules to be consistent if there is no request permitted by one of the rules which is denied by another in the set. A set of policies is consistent if there is no request permitted by one of the policies which is denied by another in the set.

Rule consistency within a policy and policy consistency within a PDP can each be checked using the method outlined above, using the functions *evaluateRule* and *evaluatePol* from Section 3.

6 Related Work

An important related piece of work (and indeed a major source of inspiration for this work) is [5]. Here the authors present Margrave, a tool for analysing policies written in XACML. Our intentions are almost identical but there are some important differences. Margrave transforms XACML policies into Multi-Terminal Binary Decision Diagrams (MTBDDs) and users can query these representations and compare versions of policies. Our work allows the possibility of the user manipulating the VDM representation of a policy then translating the changed version back into XACML. We test policies against requests where Margrave uses verification.

In [8, 15] the access control language RW is presented. It is based on propositional logic, and tools exist to translate RW programs to XACML, and to verify RW programs.

Alloy [10] has been used [9] to verify access control policies. XACML policies are translated into the Alloy language and partial ordering between these polices can be checked.

7 Conclusions and Further Work

We have presented a formal approach to modelling and analysing access control policies. We have used VDM, a well-established formal method, as our modelling notation. This has allowed us to use VDMTools to analyse the resultant formal models. We have shown that rigorous testing of these policies is possible within VDMTools, and further that policies may be checked for internal consistency.

Ongoing work seeks to represent rules that are dependent on context. This will require extending the VDM model with environmental variables, and allowing rules to query these variables. This will allow us to model a much broader range of policies, including *context-based authorisation* [4] and delegation.

With larger policies, testing all possible requests may become time consuming. Further work will look at techniques and tools for developing economical test suites for access control policies.

Using attributes of subjects instead of subject identities would allow a greater range of policies to be implemented, and this is something we are currently working on.

A full implementation of the translation from XACML to VDM and vice versa is under development.

Delegation could then also be modelled. The delegator could alter a flag in the environment (perhaps by invoking a certain rule in the PDP) and the delegate is then only allowed access if the flag is set.

Following the RBAC profile of XACML [12], the

rules we have considered so far all contain a role as the *subject*-**set** in the target. However in the core specification of XACML [13] the *subject*-**set** of a rule can be an arbitrary set of subjects. If this is the case, then in general *every* possible combination of subject, resource and action would need to be tested, rather than just a representative from each role combination.

8 Acknowledgments

Many thanks to Joey Coleman for guiding me through shell scripting, and John Fitzgerald for guiding me through VDM.

This work is part-funded by the UK EPSRC under e-Science pilot project GOLD, DIRC (the Interdisciplinary Research Collaboration in Dependability) and DSTL.

References

[1] D.J. Andrews, editor. *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*. International Organization for Standardization, December 1996. International Standard ISO/IEC 13817-1.

[2] J. C. Bicarregui, J.S. Fitzgerald, and P.A. Lindsay et. al. *Proof in VDM: A Practitioner's Guide*. Springer-Verlag, 1994.

[3] CSK. VDMTools. available from `http://www.vdmbook.com`.

[4] Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Access control: principles and solutions. *Software - Practice and Experience*, 33:397–421, 2003.

[5] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 196–205, New York, NY, USA, 2005. ACM Press.

[6] J. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors. *International Symposium of Formal Methods Europe. Newcastle, UK, July 2005*, volume 3582 of *LNCS*. Springer-Verlag, 2005.

[7] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques in Software Developement*. Cambridge University Press, 1998.

[8] D. Guelev, M. Ryan, and P. Schobbens. Model-checking Access Control Policies. In *ISC'04: Proceedings of the Seventh International Security Conference*, volume 3225 of *LNCS*, pages 219–230. Springer, 2004.

[9] Graham Hughes and Tevfik Bultan. Automated Verification of Access Control Policies. Technical Report 2004-22, University of California, Santa Barbara, 2004.

[10] D. Jackson. *Micromodels of software: Modelling and analysis with Alloy*. `http://sdg.lcs.mit.edu/alloy/reference-manual.pdf`.

[11] Cliff B. Jones. *Systematic Software Developement using VDM*. International Series in Computer Science. Prentice-Hall, 1990.

[12] OASIS. Core and heirarchical role based access control (RBAC) profile of XACML v2.0. Technical report, OASIS, Feb 2005.

[13] OASIS. eXtensible Access Control Markup Language (XACML) version 2.0. Technical report, OASIS, Feb 2005.

[14] The GOLD project. `http://gigamesh.ncl.ac.uk/`.

[15] N. Zhang, M. Ryan, and D. Guelev. Evaluating access control policies through model checking. In J. Zhou, J. Lopez, R.H. Deng, and F. Bao, editors, *Eighth Information Security Conference (ISC'05)*, volume 3650 of *LNCS*, pages 446–460. Springer-Verlag, 2005.