JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 1, No. 5, November-December 2002

Access Graphs: Another View on Static Access Control for a Better Understanding and Use

Gilles Ardourel and **Marianne Huchard**
LIRMM, 161 rue Ada, 34392 Montpellier Cedex 5 France

Encapsulation and modularity are supported by various static access control mechanisms that manage implementation hiding and define interfaces adapted to different client profiles. Programming languages use a broad range of different mechanisms, that are sometimes confusing and hard to predict when cumulatively applied. Furthermore, understanding and reasoning about access control independently from the programming languages is quite difficult. We introduce a notation for static access control that we think is adapted for modeling, characterizing, evaluating, comparing and translating access control. Examples of practical applications of access graphs are given. This notation is supported by AGATE, a set of tools designed for access control handling.

1 INTRODUCTION

As noted by [15], "Access control is actually a complex topic that aids in defining well-structured software and therefore in reasoning about correct software". Access control mechanisms indeed play a key role in modular and encapsulated software organizations [16]. The definition of appropriate access rights helps in managing implementation hiding and defining interfaces adapted to different client profiles. In object-oriented programming languages a great share of the mechanisms are *static*, which is an important feature because many accesses are thus checked at compile time, and *class-based*, and should not be confused with object-based security [15].

Object-oriented programming languages offer the most varied static access control mechanisms, and have quite different philosophies. While a language like Eiffel [17] essentially requires the explicit naming of client classes, other languages like Java [5] or C++ [22] give greater importance to the definition of categories of clients for a class, *e.g.* subclasses or classes of the same package. C++ also provides adornment of inheritance links and a *friend*-based mechanism that allows a class to grant special rights to explicitly named classes and methods. Another divergence concerns the granularity of the encapsulated component: the encapsulated item is the instance for Eiffel or Smalltalk [11] (*e.g.* attributes can be written only by the receiver's methods), while it is the class for C++ or Java (*e.g.* a method of a class C has access to private properties of any other instance of C). There is a good remark about this profusion of mechanisms in [15]: "Access control is a good case study of generalization and how a single mechanism or at least a small number of mechanisms can replace multiple mechanisms".

Besides the variety of mechanisms, access control has several other questionable aspects that hinder its understanding and use: languages are evolving fast and documentation is often so informal that several interpretations are possible[1]; compiler and language versions offer subtle variations on important themes; mechanism intrication can lead to access rights that are really difficult to understand and predict; modularity and inheritance have conflicting rules concerning access control, as observed by [20]; there is no general language, independent from the syntax of programming languages, that allows modeling and expression of any access control situation; CASE tools and design methodologies generally propose a rough schema of access control modeling essentially based on the public/private separation.

We believe that addressing these issues requires a language-independent notation, with clear semantics, allowing expression of any set of access rights, and easy to read. Such a formalism should help to express design decisions as well as being an aid in understanding and reasoning about programs. To this end, we introduce *access graphs*, a graph-based notation, that explicitly shows a set of access rights, and that is, from our point of view, adapted for modeling, characterizing, evaluating, comparing and translating sets of access rights.

Section 2 reviews and details several issues raised by access control mechanisms which prompted us to define a general notation. *Access Graphs* are presented and illustrated in Section 3. Examples where the notation is put into practice are presented in Section 4. We close Section 5 by discussing several perspectives.

2 ISSUES IN STATIC ACCESS CONTROL

As previously mentioned, static access control raises two categories of problems: a deep understanding requires more attention than supposed in a first approach; its use for modeling and programming could be enhanced.

Understanding Static Access Control

This section describes a few cases where readability of static access control is questionable.

[1]See Java Spec Report http://www.ergnosis.com/java-spec-report, Sections 15.11.12, 8.2, or 6.6.2.1 for ambiguities in Java access control description.

Intricate mechanisms Even though basic access control mechanisms are usually intelligible when they are considered separately, their intrication often results in access rights that are sometimes hard to foresee. C++ mechanisms that provide access control on properties and on inheritance links are a mine of such examples. Consider the C++ class hierarchy outlined in Figure 1 (left) with a UML-like presentation. Following [21] and its implementation in the GNU2.95 compiler, a method m declared by *Rpro* has access to the inherited protected member a of a *Rpropub* instance[2] (m can contain *Rpropub* $inst1$; ... $inst1.a$), but not to the member a of a *Rpropri* instance, which is rather logical: *private* inheritance makes a a *private* property of *Rpropri* for subclasses as well as for superclasses of *Rpropri*. But what about access inside m to the inherited protected member a of a *Rpropro* instance? Which semantics have really protected inheritance with regard to access from superclasses towards subclasses? The answer could be to consider that a remains *protected* in *Rpropro* like in *Rpropub*, but actually another choice seems to have been made, and *Rpropro* $inst2$; ... $inst2.a$ is an illegal access inside m (thus *Rpro* has lost access to a on *Rpropro*). One interpretation is that protected inheritance acts as a cut for accesses from the superclass involved in the *protected* link. This is not very natural in a first approach, and previous versions of C++ compilers (as GNU2.91) allowed the access $inst2.a$ inside m. It is debatable to what extent such subtle rules must be known by the average C++ programmer and for which design situations they are useful.

... with subtle variations Variations on protected inheritance and packages in Java is another example where subtle changes between versions can disturb programmers. Let's consider a `protected` method m declared in a class $A1$ of the package $P1$, and overridden in $A2$, a subclass of $A1$ located in another package $P2$ (see Figure 1 right). Method p of class B shows accesses to the name m within package $P1$. Access on $a1$ of static type $A1$ is legal in all JDKs, but access on $a2$ depends on the JDK version. In JDK 1.2 and 1.4, accesses from $P1$ to m on an object of static type $A2$ are indeed illegal whereas they can be achieved in the JDK 1.1.8 and JDK 1.3 versions. Such changes can invalidate code and are difficult to track.

Non-unicity of encoding Programming languages usually provide different ways to get a specific set of access rights. In Figure 2, two C++ hierarchies with equivalent allowed accesses are proposed. The trick is to combine `friend` and *private/public* inheritance in such a way that allowed accesses are restricted to accesses to m from instances of A on instances of A, B or C. This feature, *i.e.* achieving a same effect thanks to different syntactic means is quite normal for a language, but unfortunately results in complicating encoding choices and program understanding. Furthermore, despite the equivalence of allowed accesses in the first step of this hierarchy construction, the access control choices made have very different consequences when the hierarchy is extended.

[2]To be more precise, an instance whose static type is *Rpropub*. This remark also applies below.

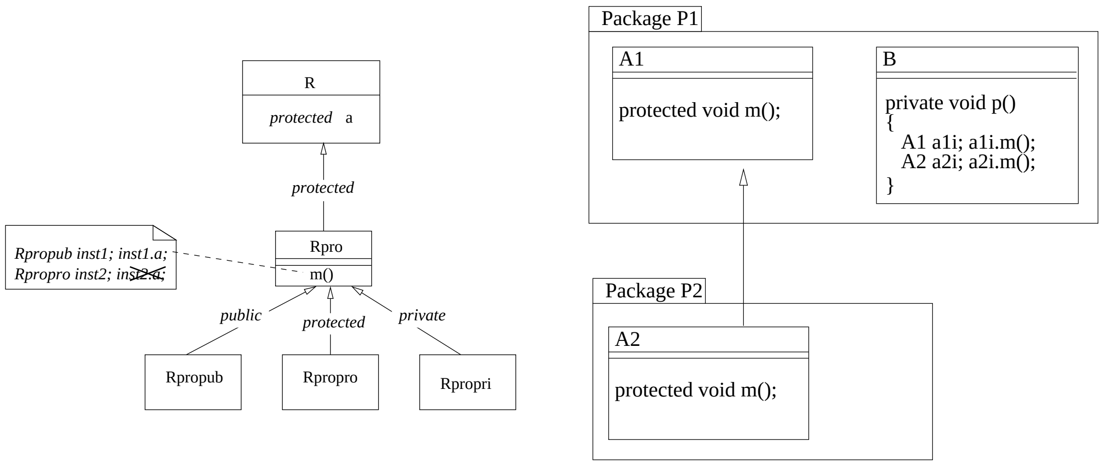

Figure 1: C++ (Left) and Java (Right) hierarchies

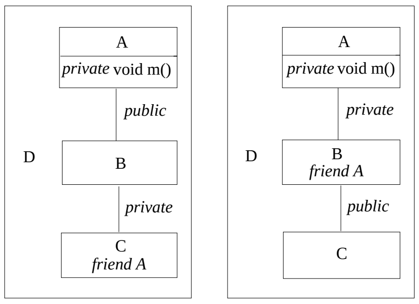

Figure 2: Equivalent allowed accesses in C++

Comparing different policies A problem raised by the variety of access control policies and syntactic structures is that it hides similarities and differences between languages. We propose an example based on three hierarchies (Figure 3), written respectively in C++, Eiffel and Java, which highlights a connection between mechanisms like `friend` (C++), *package* visibility in Java and access rights associated with the `feature` clause in Eiffel. In the figure, the package visibility is mentioned explicitly, even though there is no keyword in Java for it (it is considered the default visibility). f is supposed to be a method. In the Eiffel hierarchy, f is exported to A and C. This also allows B, which is a subclass of A, to have access to f. In the C++ hierarchy, the same accesses are obtained by declaring f protected or private (the figure shows the private case), and A and B friends of C. In the Java hierarchy, a package including A, B and C is created to restrict the access to f to these three classes. f is now declared with protected or package visibility (the figure shows the package case). Clearly understanding the different access control policies would give a more abstract view and encourage better practice. For example, understanding that `friend` is not just an unrecommended trick, but a way to get a package-like visibility (provided that the programmer pays attention to the fact that friendship is neither inherited nor transitive [21]) is useful for getting a better code. Another benefit of establishing connections is to provide actual help for translating hierarchies from one language to another, as the issue approached by [14].

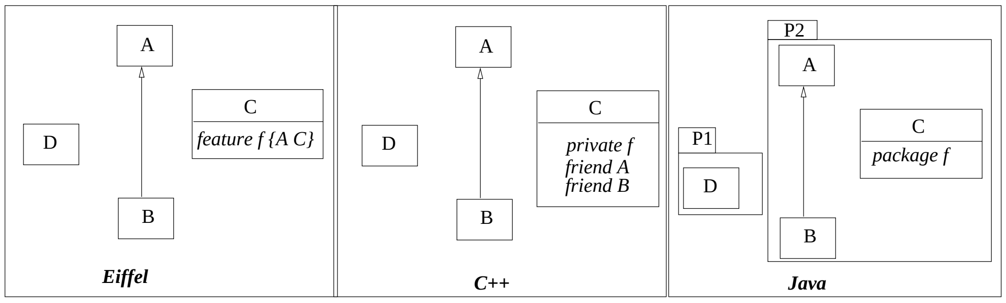

Figure 3: Three close access controls

Modeling and Programming with Static Access Control

Limitations of current modeling languages As we have seen, the very specific idioms of the different languages do not allow designers and programmers to have a clear language-independent representation of allowed accesses. In spite of its claims that "the UML's visibility property matches the semantics common among most programming languages, including C++, Java, Ada and Eiffel" [8], UML mainly borrows syntactic features from C++ or Java, quite far from Eiffel or Ada [23] principles. Four symbols are indeed proposed in UML (+, #, $\sim$, and $-$) which indicate that the concerned property is public, protected, package-level, or private. Unfortunately, the semantics of protected visibility is not the same in C++ and Java, and differs in the successive Java versions. The viewpoint of UML is that the semantics depend on the programming language [18]. The way UML considers access control is very damaging because design as well as documentation require expression of any form of access control, with clear semantics.

Issues in access control design Access Control is unanimously recognized as being an important problem which is not confined to programming phases but has to be considered from the design step [9, 19]: "If a designer encapsulates the parts of the analysis effort that are most volatile, then the inevitably changing requirements become less of a threat" [9]. In particular, access control elements are useful in design pattern specification and verification [7]. Most usual recommendations are quite general [19, 8]: separate interface and implementation (mask the latter), and do not cross associations not directly connected to the current class. A few well-known design rules have an impact on access control policies, for instance the "specialization inheritance rule" requires the access control not to be strengthened when using specialization inheritance. This rule was enforced in Java 1.3 (but unfortunately not in Java 1.2 and 1.4), while it is left to the programmer's skills in C++ and Eiffel. Apart from these generalities, very specific guidelines are associated with (or imposed by) particular constructs of programming languages: to have private attributes provided with accessor methods; to use private/protected inheritance (in C++) for implementation reasons; to use protected methods but preferably not protected attributes [22]. An intermediate level of advice, that would ensure continuity between general ideas and specific constructs of programming languages is missing. This is perhaps explained by the lack of notation supporting such a level.

3 TOWARDS A NOTATION FOR STATIC ACCESS CONTROL

The previous section highlighted the need for a notation independent from programming languages, with clear semantics, and easy to read. To this end, we propose below a graph-based notation which explicitly shows effective or allowed accesses and which we think is well adapted for handling access control in various design and programming situations.

What static access is and isn't

We focus on the main access control policies found in standard languages like Java, C++ or Eiffel. In this framework, static access is basically the fact that a piece of code, part of a method m applied to an object or a class e, has access to a property p of e *via* a name (or signature) x. The property p can be an attribute, a method or a type (class, primitive or constructed type). The access also involves applying to p an operation o that achieves a *kind* of access such as *read/write* if p is an attribute, *call* if p is a method, and *name* if p is a type of variable.

The mechanism is class-based: in the case of access to a property p of an object e, the mechanism only considers the static type (class) of the object e, and ignores the dynamic type and the identity of e. In particular, in a method m, if a variable v is set to the receiver (denoted by *this* or $self$ following the language), later in the code of m, an access $v.p$ will not be statically considered like an access $this.p$, *i.e.* an access to p on the receiver through the special variable *this*. Access control can differ: $this.p$ can be legal, while $v.p$ is forbidden.

A few examples of static accesses are shown Figure 4. The four first accesses are class-level accesses, that are accesses between different instances and accesses from or to class (static) properties. Access A_1 is a write-access from m to the instance attribute x of *inst*. Whether access is allowed or not mainly depends on the class C_1 where m is declared, and on the class C_2 which is the static type of *inst*. Access A_2 is a call-access to a class (static) method x of C_2. A_3 involves using, inside the static method m, the name of C_2 as a type name. A_4 is a classical call-access to an instance method x on *inst*, which here has the same type as the method receiver. The two next accesses are instance-level accesses, namely accesses *via* special variables (sometimes implicit) that designate the method receiver (*this*, $self$, *super*, etc.). A_5 is a read-access to the attribute x, while A_6 is a call-access to the method x. In instance-level accesses, m is necessarily an instance method.

Although the respective roles of the properties and their names (in the broad sense, signatures) are not always clearly identified in the definition of access control mechanisms, the main idea is that (static) *access control applies to names* [21] and not to entities. This is partly due to the fact that, from the static point of view we are not aware of the result of dynamic binding.

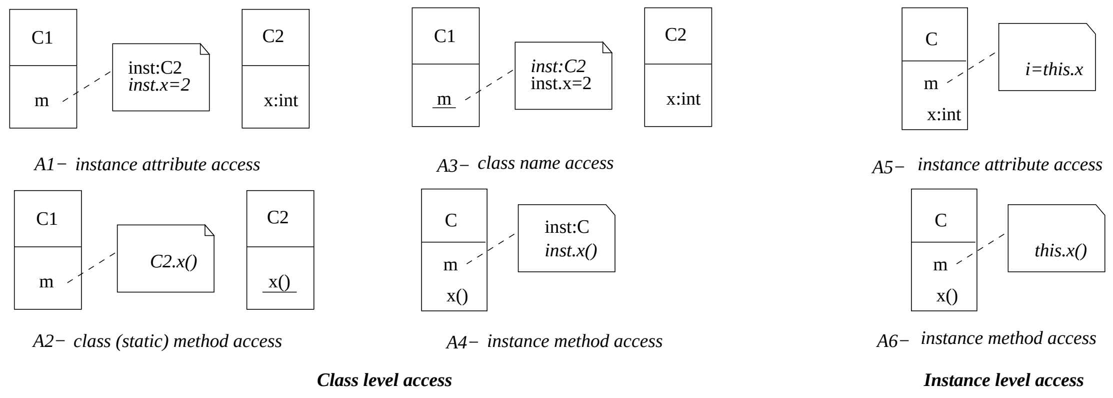

Figure 4: Examples of static access situations

An intuitive graph-based representation

We aim to help intuition and reasoning on static access by providing an easy-to-read representation, giving access control a visual notation with precise semantics. The idea is to provide access control with the same readability that given by UML [8] to other aspects of design (classes, associations, collaborations, etc.). Accesses presented in Figure 4 are highlighted by the diagrams shown in Figure 5. Nodes are labelled by classes, while edges correspond to accesses. A 3-tuple (m, sa, x) on an edge (C_1, C_2) represents the fact that C_1 has access through its method m to the property called x, with the access kind sa (read, write, call, name). Assuming that accesses of Figure 4 are buried in the code, access graphs bring even more readability.

This representation may evoke call graphs and collaboration diagrams, but there are significant differences. As opposed to call graphs[3], we take access to attributes into account, and we do not focus on the sequential aspect of the calls in our effective access graphs. We only need to know that a call is effective, not how many times it happened, so we can factorize access information at a class level. Hence, access graphs nodes are classes and not methods. Access graphs are closer to collaboration diagrams[4] of UML, but note that our notation focuses on access to various entities, including (but not reduced to) methods, and nodes correspond to classes rather than

[3] "A call graph (CG) describes the relationship between routines. Its nodes represent routines, its edges represent routine calls and returns" [24].

[4] UML diagrams that show the structural organization of objects that send and receive messages [8].

instances. Besides, contrary to collaboration diagrams, our point of view is static and not dynamic.

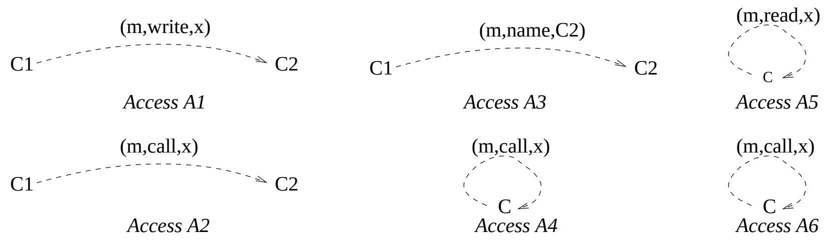

Figure 5: A graph-based representation of accesses

Nevertheless, the main target is not effective accesses, but rather *legal accesses*, which are encoded into syntactic access control constructs as well as in implicit language rules, with a significant loss of clarity. Let us take for example the following Eiffel program (and recall that Eiffel is one of the most explicit languages concerning access control):

```
class A                      class B                        class C
feature {NONE}               inherit A                      feature {NONE} r is
  f0:INTEGER;                    rename p as q               do inst:A;
  m is do n end;                 redefine m                     -- create inst as a B
  n is do end;                   export{C} n                    ...
feature {A} f1:INTEGER;      feature {NONE} m is do end;        inst.q;
feature{C}  p is do end;     end -- B                        end
end -- A                                                     end -- C
```

The corresponding effective accesses and legal accesses are appropriately represented in the graph-based notation in Figure 6. Instance-level accesses are gathered into sets called I_a while class-level accesses are gathered into sets called C_a. The principle is the same as in Figure 5, except that we have simplified edge labels in the graph which represents legal accesses: when an access to a property x is allowed for all methods of a class and with all possible access kinds (*e.g.* read and write for an attribute), the 3-tuple representing the access is reduced to x. In the graph associated with legal accesses, class name accesses are all supposed to be allowed and are not represented for the sake of simplicity.

Notations

Several useful notations are now introduced in order to formalize the notion of static access control. Notations are shortened when there is no ambiguity.

- The set of classes is denoted by $\mathcal{C}$ and ordered by the inheritance order $\leq_H$. In this order, $B \leq_H A$ if B is a subclass of A.

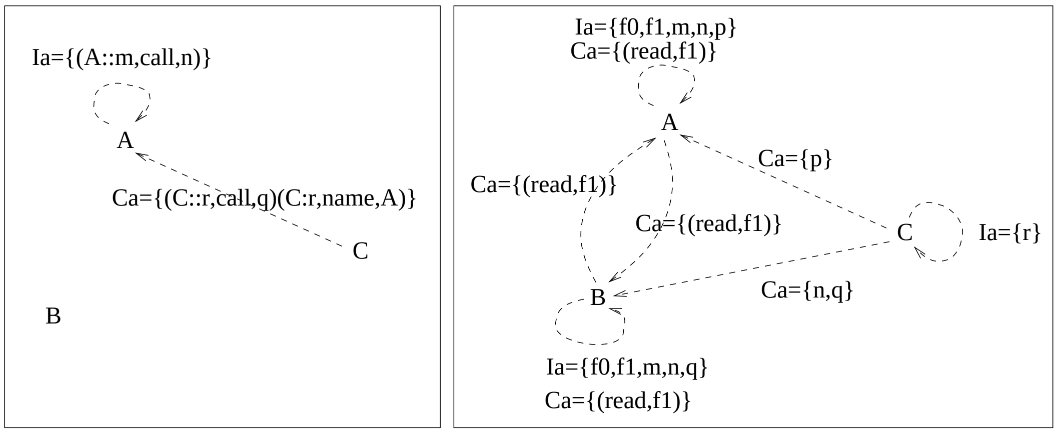

Figure 6: Effective and legal (allowed) accesses in Eiffel

- $staticType : \mathcal{V} \rightarrow \mathcal{C}$, associates with each variable the static type given when the variable is declared, in any hypothetic code, which contains variables of a set $\mathcal{V}$.

- The set of properties is denoted by $\mathcal{P}$ and contains all syntactically declared properties. An attribute or a method will be denoted by its declaring signature, where the name is prefixed by the declaring class name and the characters "::". The name of a class C will be denoted by $name(C)$. $\mathcal{P}$ is decomposed into: $\mathcal{N}_\mathcal{C}$, the class or type names (for nested classes or type declarations), $\mathcal{P}_A = \mathcal{P}_{CA} \cup \mathcal{P}_{IA}$, the set of class and instance attributes, and $\mathcal{P}_M = \mathcal{P}_{CM} \cup \mathcal{P}_{IM}$, the set of class and instance methods.

- The map $properties : \mathcal{C} \rightarrow 2^{\mathcal{P}}$, associates with a class C its property set, while the map $declared : \mathcal{C} \rightarrow 2^{\mathcal{P}}$, associates with a class C the properties it syntactically declares. The set $properties(C)$ contains the properties the language assigns to a class C, which are (in a first approximation) the properties declared by C and the properties inherited by C (declared by a superclass and not overridden).

- A map $sig : \mathcal{C} \times \mathcal{P} \rightarrow \mathcal{S}$ associates a signature with each pair (C, p), where $p \in properties(C)$. $\mathcal{S}$ is defined as the signatures in a UML-like notation `name([parameter-type-list])[:return-type]` (simplifications are made when there is no ambiguity). We suppose that a property has a unique signature in a class (otherwise the model can be extended without difficulty).

- $accessKinds = \{read, write, call, name\}$ denotes the set of access kinds that are considered here. $read, write$ are possible access kinds for attributes, while $call$ is used for methods, and $name$ for the use of type names. This set is extensible, for example if there is a need to differentiate between using a type name and instantiating a class, or between specializations of call access (*e.g.* in Eiffel to make a distinction between calling a creation method at the instantiation or calling it like a normal method).

- We consider the set $\mathcal{O}(x)$ of operations (assignments, functions, operators or methods) that may be applied to x. $\mathcal{O}(x)$ can be decomposed into $\mathcal{O}_T(x)$ (operations Transforming x), and $\mathcal{O}_{NT}(x)$ (operations Not Transforming x).

Definition of static access

First instance-level accesses are defined, that are, for a given instance method m, accesses inside m to properties of the receiver of m *via* `this`, `self` and `super` variables (or implicitly).

Definition 3.1 (Instance-level Access)
A well-formed instance-level access is a 4-tuple $(C_l, m, sa, x) \in \mathcal{C} \times \mathcal{P}_{IM} \times accessKinds \times \mathcal{S}$ *such that the following conditions are satisfied:*
▷ $\exists p \in properties(C_l), sig(C_l, p) = x$,
▷ $m \in declared(C_l)$,
▷ $p \in \mathcal{P}_{IA} \cup \mathcal{P}_{IM}$,
▷ *x appears*[5] *inside the code of m in an access expression X where x is applied to the receiver, implicitly or explicitly like in: $X = super.x$ or $X = self.x$,*
▷ *one of the three following conditions is true: $p \in \mathcal{P}_{IA}$, $sa = read$, and an operation $o \in \mathcal{O}_{NT}(x)$ is applied to X; $p \in \mathcal{P}_{IA}$, $sa = write$, and an operation $o \in \mathcal{O}_T(x)$ is applied to X; $p \in \mathcal{P}_{IM}$ and $sa = call$.*

For example, the set of effective instance-level accesses in the Eiffel example is $\mathcal{A}_{ie} = \{(A, A :: m, call, n)\}$. The set of allowed instance-level accesses for A through the method n is $\mathcal{A}_A = \{(A, A :: n, read, f_0), (A, A :: n, read, f_1), (A, A :: n, write, f_0), (A, A :: n, write, f_1), (A, A :: n, call, m), (A, A :: n, call, n), (A, A :: n, call, p), \}$.

The next definition describes class-level accesses, that are accesses between (statically) different instances and accesses from or to class (static) properties.

Definition 3.2 (Class-level access)
A well-formed class-level access is a 5-tuple $(C1, C2, m, sa, x) \in \mathcal{C} \times \mathcal{C} \times \mathcal{P}_M \times accessKinds \times \mathcal{S}$ *such that:*
▷ $\exists p \in properties(C2), sig(C2, p) = x$,
▷ $m \in declared(C1)$,
▷ *x appears inside the code of m in the access expression X detailed below,*
▷ *if $p \in \mathcal{P}_{IA} \cup \mathcal{P}_{IM}$, X has the form $v.x$, where v is a variable such that $staticType(v) = C2$ (v is neither $self$, nor $super$),*
▷ *if $p \in \mathcal{P}_{CA} \cup \mathcal{P}_{CM}$, X has the form $C2.x$*[6],
▷ *if $p \in \mathcal{P}_A$, an operation $o \in \mathcal{O}_T(x)$ (resp. $\mathcal{O}_{NT}(x)$) is applied to X if $sa = write$ (resp. $sa = read$),*

[5]In fact, an invocation of x.
[6]We do not consider degenerated invocation forms like $v.x$, where v refers to an instance.

- ▷ *if* $p \in \mathcal{P}_M$, $sa = call$,
- ▷ *if* $p \in \mathcal{N}_\mathcal{C}$, $sa = name$, *and* x *appears as a type name.*

The set of effective class-level accesses in the Eiffel example is $\mathcal{A}_{ce} = \{(C, A, C :: r, call, q), (C, A, C :: r, name, name(A))\}$ and contains an access which is not legal in Eiffel: the set of allowed class-level accesses from C to properties of A is indeed $\mathcal{A}_{CA} = \{(C, A, C :: r, name, name(A)), (C, A, C :: r, call, p)\}$ and does not contain $(C, A, C :: r, call, q)$. This set indicates in particular that $C :: r$ can call $A :: p$ on an object of static type A *via* the name p (but not *via* the name q). The set of class-level accesses allowed from C to properties of B is $\mathcal{A}_{CB} = \{(C, B, C :: r, name, name(B)), (C, B, C :: r, call, n), (C, B, C :: r, call, q)\}$. It indicates that $C :: r$ can call $A :: p$ on an object of static type B, and in this case only *via* the name q.

Access Graphs

The graph-based representation of accesses associates nodes with classes and edges with accesses.

Definition 3.3 *Let* $(\mathcal{C}, \leq_H)$ *be an inheritance hierarchy, the access graph associated with the class-level access set* $\mathcal{A}_C$ *and the instance-level access set* $\mathcal{A}_I$ *is the graph* $(\mathcal{C}, \mathcal{E})$ *such that* $(C1, C2) \in \mathcal{E}$ *if there is a 5-tuple* $(C1, C2, m, sa, x)$ *in* $\mathcal{A}_C$ *or a 4-tuple* $(C1 = C2, m, sa, x)$ *in* $\mathcal{A}_I$.

Edges are labelled by the following maps:

- ▷ $Ia : \mathcal{E} \rightarrow 2^{\mathcal{P}_{IM} \times accessKinds \times \mathcal{S}}$
  $(C_l, C_l) \longrightarrow \{(m, sa, x)/(C_l, m, sa, x) \in \mathcal{A}_I\}$
- ▷ $Ca : \mathcal{E} \rightarrow 2^{\mathcal{P}_M \times accessKinds \times \mathcal{S}}$
  $(C1, C2) \longrightarrow \{(m, sa, x)/(C1, C2, m, sa, x) \in \mathcal{A}_C\}$

Figures 5 and 6 gave examples of access graphs with simplifications of labels as explained above.

4 ACCESS GRAPHS IN USE

This view of static access control offers many prospects for improving specification, understanding and use of static access control mechanisms. In this section, we show application of access graphs in access control mechanism formalization or understanding, and how they help in reasoning and modeling. Finally AGATE (*Access Graph Based Tools for Encapsulation*), a set of tools dedicated to static access control management is described.

Formalizing Static Access Policies

A first application of access graphs is to give a description of access control mechanisms available in a programming language. The result (the family of access graphs allowed by the rules of the language) has the advantage of being easy to understand and independent of the source languages. Below we give several rules that partially show how Eiffel and Java access control mechanisms can be expressed through access graphs.

Access Graphs for Eiffel Access Control The main access control mechanisms of Eiffel [17] are simple enough to be described by the six following rules, excluding only descriptions of class variable, class method, and class name accessibility, as well as specificities of constructor methods.These rules define the sets $\mathcal{A}_I$ and $\mathcal{A}_C$ of legal (allowed) accesses associated with a given Eiffel hierarchy $(\mathcal{C}, \leq_H)$. A simplified view of rules is given in Figure 7.

Rule 4.1
" Attributes can be read through instance-level access."
For all $C \in \mathcal{C}$, $a \in properties(C) \cap \mathcal{P}_{IA}$, $meth \in declared(C) \cap \mathcal{P}_{IM}$,
we have $(C, meth, read, sig(C, a)) \in \mathcal{A}_I$.

Rule 4.2
"Attributes can be written through instance-level access."
For all $C \in \mathcal{C}$, $a \in properties(C) \cap \mathcal{P}_{IA}$, $meth \in declared(C) \cap \mathcal{P}_{IM}$,
we have $(C, meth, write, sig(C, a)) \in \mathcal{A}_I$.

Rule 4.3
"Methods can always be called through instance-level access."
For all $C \in \mathcal{C}$, $m \in properties(C) \cap \mathcal{P}_{IM}$, $meth \in declared(C) \cap \mathcal{P}_{IM}$,
we have $(C, meth, call, sig(C, m)) \in \mathcal{A}_I$.

Rule 4.4
"A property accessible by a class is also accessible by its subclasses."
For all $C1, C2, C3 \in \mathcal{C}, C2 \neq C1, C3 \leq_H C1$, *if there are* $sa \in accessKinds$, $meth1 \in (declared(C1) \cap \mathcal{P}_M)$ *and* $x \in \mathcal{S}$ *such that* $(C1, C2, meth1, sa, x) \in \mathcal{A}_C$,
then $\forall meth2 \in (declared(C3) \cap \mathcal{P}_M)$ $(C3, C2, meth2, sa, x) \in \mathcal{A}_C$

`Export` and `feature` clauses are translated in our formalism according to the following two rules. The first rule describes the strict application of the export mechanism.

Rule 4.5 (class C1 contains `export {X1, ..Xn} m` or `feature {X1, ..Xn} m`)
implies:
Let $L = \{X1, X2, ...Xn\}$, *for all* $Xi \in L$, $meth \in declared(Xi) \cap \mathcal{P}_{IM}$, *and let* $m = sig(C1, C1 :: m)$ *we have* $(Xi, C1, meth, call, m) \in \mathcal{A}_C$ *if* $C1 :: m \in \mathcal{P}_{IM}$, *and* $(Xi, C1, meth, read, m) \in \mathcal{A}_C$ *if* $C1 :: m \in \mathcal{P}_{IA}$.

The second rule states that if a class $C3$ has access to a property m named $p1$ in a class $C1$ and if $C1$ has a subclass $C2$ which knows the same property with the name $p2$ (we may have $p1$=$p2$), if m has been exported explicitly neither by $C2$ nor by a subclass of $C1$ which is also a superclass of $C2$, then $C3$ also has access to m in $C2$ (through the name $p2$).

Rule 4.6 *Let* $C1 \in \mathcal{C}$*,* $m \in properties(C1)$*, with* $p1 = sig(C1, m)$*, let* $C2 \in \mathcal{C}$*,* $C2 \leq_H C1$*, with* $p2 = sig(C2, m)$*, if* m *has been exported explicitly neither by* $C2$ *nor by a subclass of* $C1$ *which is also a superclass of* $C2$ *(through an* `export` *clause), then for all* $C3 \in \mathcal{C}$*, and for all* $meth \in declared(C3) \cap \mathcal{P}_{IM}$*, such that* $(C3, C1, meth, sa, p1) \in \mathcal{A}_C$*, we also have* $(C3, C2, meth, sa, p2) \in \mathcal{A}_C$*.*

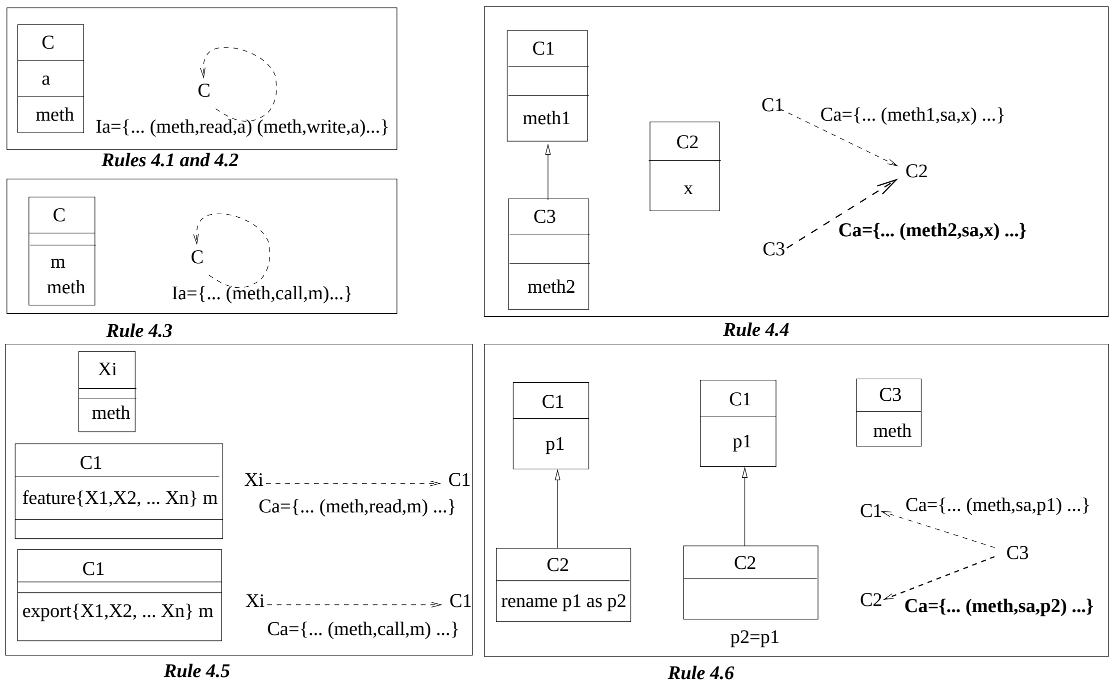
C
a
meth
C
Ia={... (meth,read,a) (meth,write,a)...}
Rules 4.1 and 4.2
C
m
meth
C
Ia={... (meth,call,m)...}
Rule 4.3
C1
meth1
C3
meth2
C2
x
C1
Ca={... (meth1,sa,x) ...}
C2
C3
Ca={... (meth2,sa,x) ...}
Rule 4.4
Xi
meth
C1
feature{X1,X2, ... Xn} m
Xi
C1
Ca={... (meth,read,m) ...}
C1
export{X1,X2, ... Xn} m
Xi
C1
Ca={... (meth,call,m) ...}
Rule 4.5
C1
p1
C2
rename p1 as p2
C1
p1
C2
p2=p1
C3
meth
C1
Ca={... (meth,sa,p1) ...}
C3
C2
Ca={... (meth,sa,p2) ...}
Rule 4.6

Figure 7: Translation of Eiffel rules into Access Graphs

Access Graphs for Java Access Control In order to show how access graph notation fits Java mechanisms, a few Java rules are extracted from the Java language Specification. Here these rules partially define the set $\mathcal{A}_C$ of legal (allowed) accesses associated with a given Java hierarchy $(\mathcal{C}, \leq_H)$.

A first rule establishes that *"a member (class, interface, field, or method) of a reference (class, interface, or array) type or a constructor of a class type is accessible only if the type is accessible."*[12]

Rule 4.7 $\forall C1, C2 \in \mathcal{C}, \forall meth \in declared(C1) \cap \mathcal{P}_M$,
$(C1, C2, meth, name, name(C2)) \notin \mathcal{A}_C \Rightarrow \nexists (sa, x) \in accessKinds \times \mathcal{S}$ *s.t.*
$(C1, C2, meth, sa, x) \in \mathcal{A}_C$

"If the member or constructor is declared public, then access is permitted."[12]

Rule 4.8 `class C2 {public m}` *implies:*
$\forall C1 \in \mathcal{C}, \forall meth \in declared(C1) \cap \mathcal{P}_M, (C1, C2, meth, name, name(C2)) \in \mathcal{A}_C \Rightarrow \forall sa$ *possible access kind for m,* $(C1, C2, meth, sa, sig(C2, m)) \in \mathcal{A}_C$

"If the member or constructor is declared private, then access is permitted if and only if it occurs within the body of the top level class (7.6) that encloses the declaration of the member."[12]

Rule 4.9 (private member) `class C2 {private m}` *implies:*
$\forall C1 \in \mathcal{C}, C1 = C2 \Leftrightarrow \forall meth \in declared(C1) \cap \mathcal{P}_M, \forall sa$ *possible access kind for m,* $(C1, C2, meth, sa, sig(C2, m)) \in \mathcal{A}_C$

For the next rules, we introduce the map *package* which associates with each class C the smallest package which contains C.

"Default access, (...) is permitted only when access occurs from within the package in which the type is declared."[12]

Rule 4.10 (default member) `class C2 {          m}` *implies:*
$\forall C1 \in \mathcal{C}, package(C1) = package(C2) \Leftrightarrow \forall meth \in declared(C1) \cap \mathcal{P}_M, \forall sa$ *possible access kind for m,*$(C1, C2, meth, sa, sig(C2, m)) \in \mathcal{A}_C$

When inheritance is not concerned, `protected` visibility is reduced to `package` visibility.

Rule 4.11 (protected member (reduced)) `class C2 {protected m}` *and there is no inheritance relation between C1 and C2 implies:*
$\forall C1 \in \mathcal{C}, package(C1) = package(C2) \Leftrightarrow \forall meth \in declared(C1) \cap \mathcal{P}_M, \forall sa$ *possible access kind for m,* $(C1, C2, meth, sa, sig(C2, m)) \in \mathcal{A}_C$

Help in Understanding

Examples of Section 2 are revisited to show how access graphs make them more readable. Figure 8 (left) shows for Figure 1 (left) differences between the compilers GNU2.91 and GNU2.95 (following the standard ANSI/ISO C++ in this situation) in a more efficient way than a lengthy textual description. Figure 8 (center) gives a straightforward overview of the accesses allowed in the two hierarchies of Figure 2. Figure 8 (right) highlights the allowed accesses in the three hierarchies of Figure 3.

The differences shown in Figure 1 (right) between the JDK versions are also better explained with the graph of allowed accesses, as in Figure 9. Thanks to its readability, this notation was found to be very useful for teaching students access control mechanisms [4].

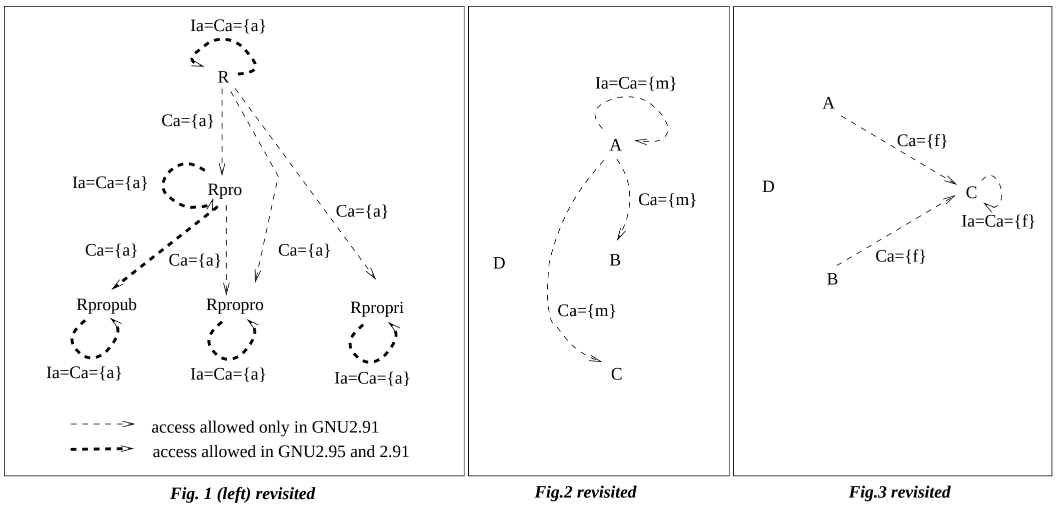

Figure 8: Access graph associated with Figures 1 (left), 2 and 3

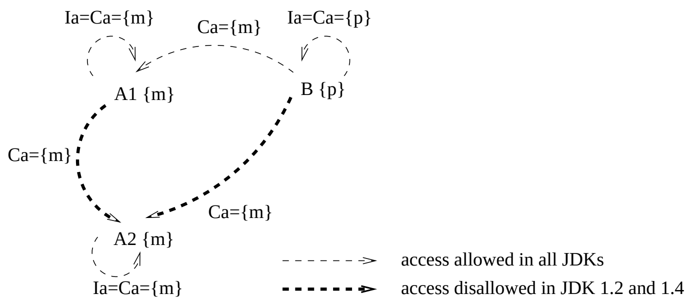
Ia=Ca={m}
Ca={m}
Ia=Ca={p}
A1 {m}
B {p}
Ca={m}
Ca={m}
A2 {m}
Ia=Ca={m}
access allowed in all JDKs
access disallowed in JDK 1.2 and 1.4

Figure 9: Access graph showing an oddity of the JDK

Reasoning with access graphs

Access graph notation is a good common framework for reasoning about access control. As an example, we show its suitability for establishing that Eiffel and Java have uncomparable access control mechanisms: each language can express an access graph that the other cannot. We restrict comparison to cases where neither the property sets of classes, nor the inheritance relations, nor the nesting relations between classes can be changed. For simplicity, we do not mention the accessing method.

Figure 10 shows two simple hierarchies together with their associated access graphs. The hierarchy and the access graph on the left of the figure are admitted by Eiffel, with the following schematic code:

```
class A              class B                                         class C
end -- A                feature {A,B}  x is do end;                  end -- C
                        feature {B,C}  y is do end;
                     end -- B
```

Reasoning on the access graph, we show that there is no Java code giving the same access rights. Let us consider the method $B :: x$ of class B such that $sig(B, B :: x) = x$. In Java, $B :: x$ could be either `public`, `private`, `protected` or *default*. Since there are no inheritance links between classes in this example, only Rules 4.8, 4.9, 4.11, or 4.10 have to be considered.

- If $B :: x$ would be `public`, as for any $m \in declared(C)$, $(C, B, m, call, x) \notin \mathcal{A}_C$ Rule 4.8 implies that we should have $(C, B, m, name, name(B)) \notin \mathcal{A}_C$. Using Rule 4.7, we would have $Ca(C, B) = \emptyset$. This contradicts the access graph since it establishes that $\forall m \in declared(C)$, $(C, B, m, call, y) \in \mathcal{A}_C$. By similar reasoning, $B :: y$ cannot be `public`.
- If $B :: x$ would be `private`, as for any $m \in declared(A)$, $(A, B, m, call, x) \in \mathcal{A}_C$ we should have by Rule 4.9 $A = B$. Contradiction. By similar reasoning, $B :: y$ cannot be `private`.
- The above points imply that $B :: x$ and $B :: y$ can only be `protected` or *package* $(default)$. $B :: y$ `protected` or *default* and $(C, B, meth, call, y) \in \mathcal{A}_C$ for any $meth \in declared(C)$ implies by Rules 4.11 or 4.10 that $module(B) = module(C)$. Now $B :: x$ `protected` or *default* and $module(B) = module(C)$ implies, by the same rules 4.11 or 4.10, that $(C, B, meth, call, x) \in \mathcal{A}_C$. Contradiction with the access graph.

In the second example (at the right of Figure 10), the hierarchy and access graph may be encoded in Java, for example with the following code:

```
package P1;                 package P1;           package P2;
public class A              public class B {}     public class C extends B {}
{protected void x(){} }
```

On the access graph we can firstly see that $(B, A, meth, call, x) \in \mathcal{A}_C$, $\forall meth \in declared(B)$, secondly that $(C, A, meth, call, x) \notin \mathcal{A}_C, \forall meth \in declared(C)$ and also that $C \leq_H B$. This clearly contradicts Eiffel Rule 4.4 and therefore this example has no possible translation in Eiffel.

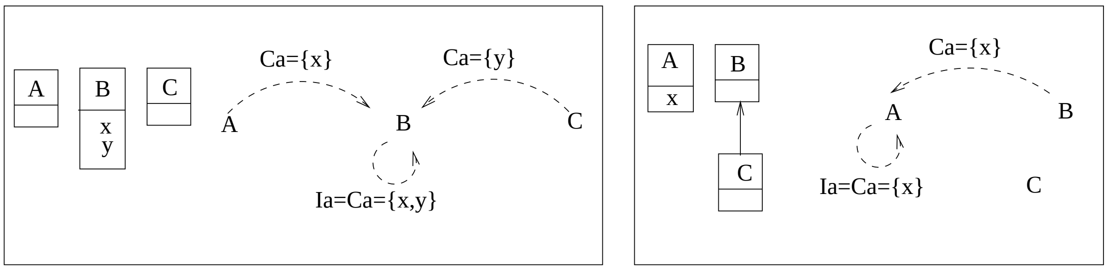

Figure 10: Comparing Java and Eiffel mechanisms

Under the same hypotheses, we have also shown that the C++ access control mechanisms are not comparable with Java or Eiffel mechanisms.

Support for modeling

We illustrate, through a short example, the role that access graphs can play in the design of static accesses. Let us consider the class diagram of Figure 11, which describes suppliers, retailers and persons. The class `Supplier` provides two methods, `supply` which is intended for *retailers* to obtain a given quantity of products of a given reference, and `retailerList` which is designed for *persons* that would like to know the retailers approved by the supplier. The class `Retailer` offers the method `sell`, which is intended for *persons* who buy products. `Retailer` admits two subclasses that respectively represent retailers of alcohol and retailers of candies. A subclass `Child` of `Person` has the right to buy only specific products, namely candies, so can have access to `sell` only on `RetailerOfCandies`. Access graph notation can express requirements in terms of access rights, as in Figure 12 (left). We think that this representation (maybe simplified or considered from different views) is the right support for elaborating encoding choices.

For simplicity, and since Ia and Ca on loops do not pose actual problems in this example, the discussion only deals with cross accesses: loops are not considered in the access graph and its encoding.

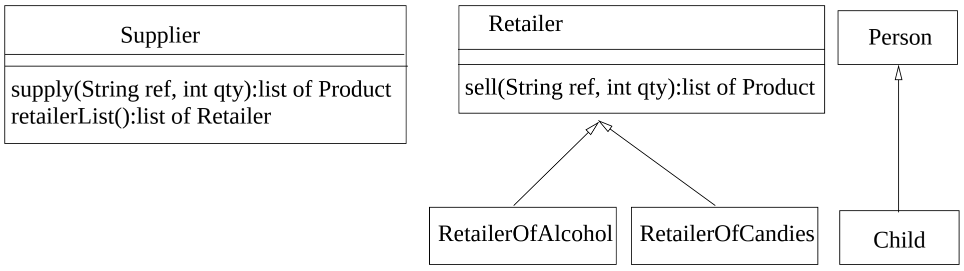
Supplier
supply(String ref, int qty):list of Product
retailerList():list of Retailer
Retailer
sell(String ref, int qty):list of Product
Person
RetailerOfAlcohol
RetailerOfCandies
Child

Figure 11: Suppliers, Retailers and Persons

Encoding the access rights determined at the design step requires finding, in the target language, access controls that will ensure the required set of accesses, and as few additional (not required) accesses as possible.

The required access rights can be approximated in Eiffel with the clauses out-

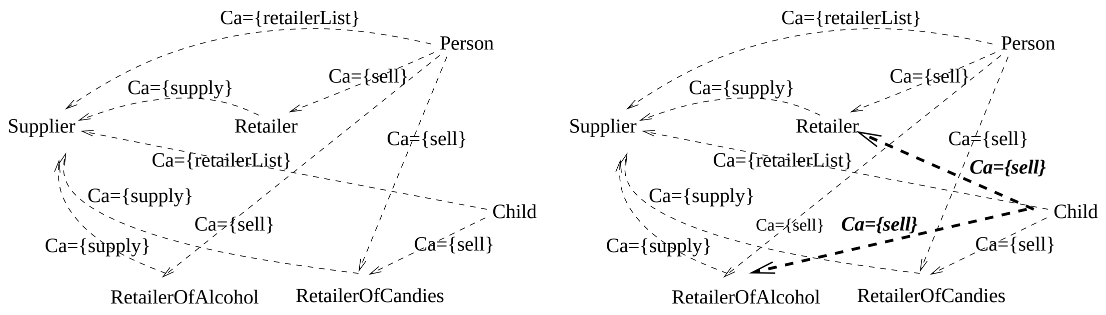

Figure 12: *left* Required accesses; *right* Least access set in Eiffel

lined in Figure 13. Clause `feature{Retailer} supply` provides only accesses required for `supply`: access is allowed explicitly to `Retailer`, and Rule 4.4 extends it to its subclasses `RetailerOfAlcohol` and `RetailerOfCandies`. For similar reasons, Clause `feature{Person} retailerList` provides only accesses required for `retailerList`. Applied to clause `feature{Person} sell`, Rule 4.4 gives unfortunately access for `Child` to `sell` on `RetailerOfAlcohol`. The associated access graph is shown in Figure 12 (right): the two edges added by the adaptation to Eiffel are thickened.

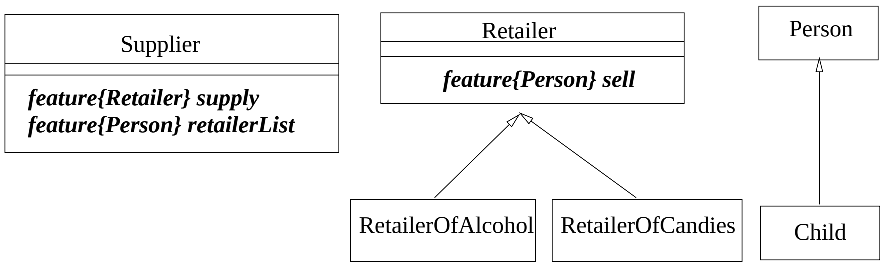

Figure 13: A solution in Eiffel

Let us now consider the case of Java encoding. Note that we are in a situation quite close to that in Figure 10: if we do not accept to modify the design of classes, the best choice is to include all classes in the same package, and to make *package* (*default*) the three methods `supply`, `retailerList` and `sell`. Indeed, making them *private* is impossible, *public* seems too liberal and *protected* does not make sense here since the rights we want to adapt do not concern access from classes to their own (possibly inherited) properties. The associated access graph is not represented but is simple to design, as it contains all possible cross accesses within the package `Trade`.

In this example, Eiffel access rights allow an encapsulation closer to the initial design. Dynamic checking of instance type can be used afterward in order to avoid children, for example, to buy alcohol.

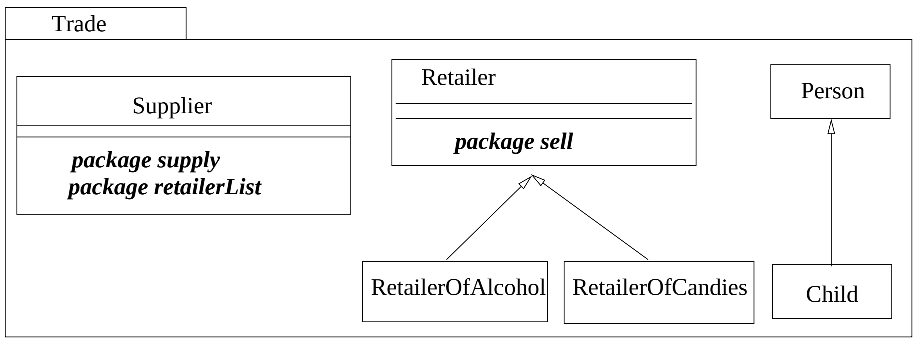

Figure 14: A solution in Java

As the access graph can rapidly increase as other classes are added, we think it is beneficial to have an automatic help for choosing a suitable encoding among several possibilities, and for tracing which alterations the encoding brings to the initial design. Such help is one of the functions of AGATE, *i.e.* a set of tools which are presented in the next section.

AGATE, *Access Graph Based Tools for Encapsulation*

Object-Oriented Case tools, being mainly based on UML, are not sufficient to handle properly access control design. In order to test new strategies in access control handling, we devised **AGATE** [2, 13] a prototype developed to help in design, understanding and managing static access.

- The Access Graph *Editor/Viewer* is a graphical tool for helping designers and programmers to understand and specify access control in a simple way [3, 13]. In a future work, some accesses will be generated automatically according to general rules and could be filtered out to keep only relevant accesses.
- The Access Graph *Extractor* is currently used to extract the graph representing the allowed accesses from Java and Eiffel source code. Extracting required access graphs from collaboration diagrams and extracting effective access graphs from source code are under way.
- The Access Graph *Adapter* takes an access graph and adapts it to a specific language. It is currently fully automated and implemented for Eiffel. Java will require user interaction, especially for adapting packages. It can help in preserving access control when translating from one language to another, a difficult task as shown in [1].
- The Access Graph *Design Checker* can show differences between a design and either the adaptation to a specific language (and thus, help to choose a target language) or the implementation. It can also give advice and warnings, even concerning restrictions not expressible in the target language.
- The Access Graph *Rule Checker* will check if access control policies are addressed by a program, as for example the specialization inheritance rule previously presented.
- The *Code Generator* produces the code (currently Eiffel code) corresponding to an access graph in a specific language, provided the access graph can be expressed in this language.

5 CONCLUSION

Access Graphs, a graph-based notation for static access control in object-oriented languages has been presented. We believe it should help in reasoning about access control in the different steps of object-oriented development: modeling, characterizing, evaluating, comparing sets of accesses, as well as preserving access control when translating from one language to another. We also presented AGATE, the kernel of a framework dedicated to access control handling.

Even though this approach encounters difficulties due to intrication of access control with other features of languages (status of accessor methods, calling an inherited method *via* `super` vs. explicit class naming), we think that most situations can be fruitfully encoded in access graphs. In the future, we plan to study the usefulness of our approach in broader issues concerning access control, for instance evaluating the interaction between encapsulation and immutability, two closely related topics, as shown in [6].

Design is concerned about both access control and implementation [9, 19] and adds its own problems, such as defining specific access control for whole-part associations [10]. One of our prospects is to study integration of access graphs with UML diagrams. One goal is to define a UML profile based on access graphs, adapted to static access control design and which could be proposed as an extension in UML-based Case Tools. Simplified views on access graphs and extension of the access graph notation to denote patterns of accesses would further improve their interest and handling.

We also plan to develop a simple and general model for static access control in the object-oriented framework. We think access graphs could help in such research work by providing simple and intuitive support.

REFERENCES

[1] M. Abadi. Protection in programming-language translations. In *Automata, languages and programming: 25th international colloquium, ICALP'98*, Lecture Notes in Computer Science – 1443, pages 868–883. Springer verlag, Berlin, 1998.

[2] G. Ardourel and M. Huchard. AGATE, Access Graph bAsed Tools for handling Encapsulation. In *Proceedings of the 16th IEEE International Conference Automated Software Engineering (ASE) 26-29 Nov. 2001 San Diego California*, pages 311–314, 2001.

[3] G. Ardourel and M. Huchard. Representing static access control in object oriented software engineering. Workshop of Visualization of Relationnal Data WVRD 2002 ftp://ftpdim.uqac.ca/pub/ychirico/wvdr2002/ardourel.pdf, 2002.

[4] G. Ardourel and M. Huchard. Teaching encapsulation and modularity in object-oriented languages with access graphs. Sixth Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts, ECOOP 2002. http://prog.vub.ac.be/ecoop2002/ws03, 2002.

[5] K. Arnold and J. Gosling. *The Java Programming Language Second Edition.* Addison Wesley, 1998.

[6] M. Biberstein, J. Gil, and S. Porat. Sealing, encapsulation, and mutability. In *ECOOP 2001 Proceedings Budapest Hungary*, 2001.

[7] A. Blewitt, A. Bundy, and I. Stark. Automatic verification of Java Design Patterns. In *Proceedings of the 16th IEEE International Conference Automated Software Engineering (ASE) 26-29 Nov. 2001 San Diego California*, pages 324–327, 2001.

[8] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[9] P. Coad and E. Yourdon. *Object-Oriented Design*. Prentice-Hall, 1991.

[10] R. K. Ege. Encapsulation — the key to re-usable software components. http://tools.fiu.edu/encaps-one.ps, 1997.

[11] A. Golberg and D. Robson. *Smalltalk-80, the Language and its Implementation*. Addison Wesley, Reading, Massachusetts, 1983.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (Second Edition)*. Addison Wesley, 2000.

[13] O. Gout, G. Ardourel, and M. Huchard. Access Graph Visualization: A step towards better understanding of static access control. Electronic Notes in Theoretical Computer Science vol 72, no.2, http://www.elsevier.nl/, 2002.

[14] J. Jorgensen. A Comparison of the Object Oriented Features of Ada 9X and C++. In *ADA-Europe 93*, pages 125–141, 1993.

[15] I. Joyner. *Objects Unencapsulated, Java, Eiffel and C++*. Prentice Hall, 1999.

[16] B. Meyer. *Object-oriented Software Construction*. Englewood Cliffs NJ: Prentice Hall, 1988.

[17] B. Meyer. *Eiffel, The Language*. Prentice Hall — Object-Oriented Series, 1992.

[18] Rational Software Corporation. *UML v 1.3 Documentation*. http://www.rational.com/uml/resources/documentation/index.jtmpl.

[19] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice Hall Inc. Englewood Cliffs, 1991.

[20] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *Special issue of Sigplan Notice — Proceedings of ACM OOPSLA'86*, 21(11):38–45, 1986.

[21] I. standard ISO/IEC 14882. *Programming Languages - C++*. Information Technology Council, 1998. http://www.ncits.org/cplusplus.htm.

[22] B. Stroustrup. *The C++ programming language, Third Edition.* Addison Wesley, 1997.

[23] S. T. Taft and R. A. Duff. *The Ada 95 Reference Manual*, volume 1246. Lecture Notes in Computer Science, Springer-Verlag, 1997.

[24] H. Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju-do, South Korea, December 2000.

ACKNOWLEDGEMENTS

The authors would like to thank M. Boulet, C. Class, B. Dufes, O. Gout, S. Renoux and S. Tenon, who contributed to develop part of the Eiffel tools during their Master's dissertations.

ABOUT THE AUTHORS

Gilles Ardourel (ardourel@lirmm.fr) is about to complete his PhD Thesis at the LIRMM (Lab. CNRS & Science University of Montpellier) on the subject of access control modeling in Object-Oriented Languages. He is interested in language design and modeling. http://www.lirmm.fr/∼ardourel

Marianne Huchard (huchard@lirmm.fr) is an assistant professor at the LIRMM (Lab. CNRS & Science University of Montpellier) since 1992. Her research interests include various aspects on class hierarchies (multiple inheritance, conflict resolution based on linearization methods, inheritance graph decomposition, algorithms for class hierarchy construction), and more recently access control mechanisms and exception modeling. http://www.lirmm.fr/∼huchard