JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 4
Special issue: TOOLS USA 2003

Class-based Visibility from an MDA Perspective: From Access Graphs to Eiffel Code

G. Ardourel, LINA, University of Nantes, France
M. Huchard, LIRMM, Montpellier, France

Encapsulation is one of the main principles in object-oriented software construction. Reducing software component coupling and enforcing interface definition facilitates maintenance, reusability and incremental development, thus increasing software quality.
Encapsulation is usually supported by specific access control (visibility) mechanisms as `private` or `protected` in Java and C++ or Eiffel's exporting mechanism. These mechanisms are often static and class-based, sometimes method-based, giving classes the status of both the client and service provider. They help in hiding implementation as well as capturing part of the domain rules on access rights.
We explore solutions for introducing design and implementation of class-based access control from a Model-Driven Architecture (MDA) perspective. As UML proposes only language-dependent solutions for access control representation, we consider extending UML with access graphs, a formalism previously introduced for comparing and reasoning about static access rights.
In this paper, we describe the integration of AGATE, a suite of access graph based tools dedicated to access control, in the MDA process. We focus on two access control design tools operating at the Platform-Independent Model (PIM) level: the Rule Adapter makes access graphs compliant with specific design rules while the Client Discovery Module adds new relevant classes expressing client organization. Mapping to a Platform-Specific Model (PSM) is illustrated with the Eiffel Access Graph Adapter and rules for generating code from this PSM are presented.

1 INTRODUCTION

Encapsulation and the close notions of access control and visibility are key concepts in object-oriented software construction [24, 7]. Reducing software component coupling and enforcing interface definition facilitate maintenance, reusability and incremental development, thus increasing software quality. In this paper, we prefer the term "visibility", used by UML, because "access control" also refers to run-time, object-based security [21], while here we consider class-based, statically checked control of access to class features (attributes and methods). Like many other language concepts, there are several views and implementations of visibility. Kinds of accesses (like read/write) can be distinguished (in Eiffel [23]) or not (in Ada [31]/Smalltalk [14]/Java [15]/C++[30]); access to features on pseudo-variables `self` or `this` can be controlled differently than access to features on any variable. While C++ and

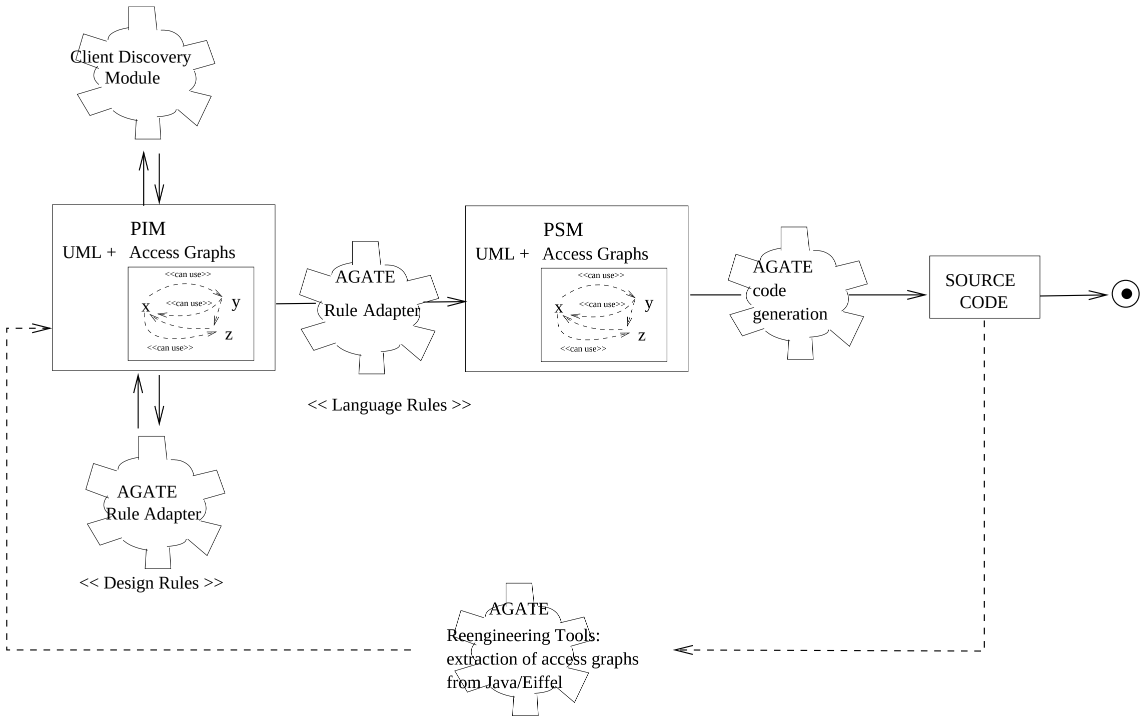

Figure 1: Class-based visibility tools in an MDA process

Java mainly propose the use of anonymous clients for a class [1], like its subclasses or classes of the same package, Eiffel introduces a client clause where clients are explicitly named classes. Eiffel visibility principles enlarge the range of potential visibility use, by letting a class propose different "interfaces" based on the application semantics. In UML, "interfaces" are also used to group externally visible operations [7] but the visibility of operations in interfaces is always `public`, thus without restricting accessibility to explicitly named clients.

The aim of MDA is to get system interoperability through standardization of the design rather than through an unlikely standardization of implementation [27]. The approach consists of first developing a "Platform Independent Model" (PIM) of the application which only involves domain business modeling. This PIM is expressed in UML using high-level core model features. Then the PIM is mapped (by hand or automatically) to a "Platform Specific Model" (PSM) of the application still expressed in UML, but integrating technical aspects of the target platform [29]. A code generation step ends the process. This application construction approach gives rise to many questions about object-oriented programming languages [10]: Have these languages the right concepts for expressing a PIM and recognizing it in the code after the mapping? Conversely, which programming language concepts are required at the PIM level? Considering visibility in the framework of MDA demands, as for other object orientation concepts, to clarify what is visibility independently from specificities of target programming languages. But what shall the visibility standard

[1]If we except the C++ `friend` mechanism.

be: Eiffel-like or UML/C++/Java-like visibility? Are boundaries between visibility and security really clear when we consider modeling high-level requirements? This is an important problem because security, defined at the PIM level as a pervasive service [26], is not yet defined [2], precluding definitive distinction between high-level visibility and access control security concepts.

To deal with the problem, we consider here that visibility at a PIM level has to be based on a generalization of current programming language policies, thus more rich than current UML proposals. Our proposal is twofold: first we introduce a language-independent visibility model which has a graphical counterpart easy to integrate into UML (*access graphs*); then, on the basis of the philosophy of the MDA approach, tools that automate visibility model transformations are developed. These tools, which are part of AGATE, a tool suite dedicated to access graph (stored as XML data) manipulation [3], are sketched in Figure 1. The *Rule Adapter* makes access graphs compliant with specific design rules. The *Client Discovery Module* adds new relevant classes expressing client organization. Eiffel is used to illustrate the mapping to a Platform-Specific Model (PSM) by using the *Rule Adapter* with Eiffel visibility rules. Tools that can extract access graphs from Java or Eiffel source code support additional reengineering processes.

Integration of a platform independent visibility concept in UML is outlined in Section 2. PIM-to-PIM transformations, that help in refining the application model, are presented in Section 3. PIM-to-PSM transformations, which map a PIM to a target language, and code generation are illustrated in the case of Eiffel (Section 4). Section 5 concludes the paper, giving the prospects of this work.

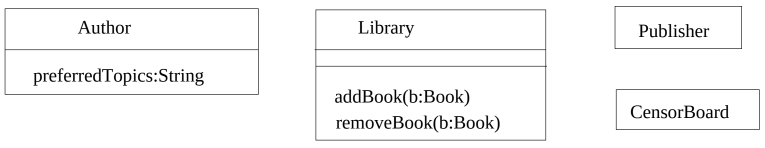

Figure 2: A class diagram for library representation

2 EXTENDING UML WITH ACCESS GRAPH NOTATION

First, weaknesses of the UML notation for class-based visibility are pointed out. Then we propose and illustrate an enhancement of the UML standard notation based on UML extension mechanisms, namely stereotypes and tagged values.

Limits of UML visibility for class-based access control expression

UML considers four possibilities for visibility [25]: *public*, *private*, *protected* and *package*, respectively denoted by +, -, # and ˜ specified before class features. The

[2]http://www.omg.org/mda/faq-mda.html

semantics of these symbols are language dependent, and essentially borrowed from Java and C++: potential clients of the features of a given class C are C, subclasses (not explicitly named) of C, classes in the C package and any other class. This policy drastically restricts the kinds of potential clients. As a drawback, access constraints which would be based on the semantics of the domain classes cannot be expressed. This problem is illustrated by the class diagram in Figure 2. In this diagram, four classes related to the library domain are presented: `Library`, `Author`, `Publisher` and `CensorBoard`. UML visibility notation cannot be used to express two very simple kinds of statements:

- *access limited to method receiver* `preferredTopics` of an author can only be mentioned in methods of `Author` and is only accessible on the method receiver (often known as `this` or `self`);
- *classes are explicitly clients of services from other classes* in methods of `Author`, methods `addBook` and `removeBook` can be invoked on any instance of `Library`; in methods of `Publisher`, only the method `addBook` can be invoked on any instance of `Library`; in methods of `CensorBoard`, only the method `removeBook` can be invoked on any instance of `Library`.

Consider the first statement. In UML (basing our reasoning on Java and C++ language semantics), the first part (“`preferredTopics` of an author can only be mentioned in methods of `Author`”) leads us to choose *private* directives (symbol −), but the second part of the statement (“`preferredTopics` of an author can only be applied to the method receiver”) is not fulfilled.

In the case of the second statement, we could try to use anonymous client classes of UML (subclasses, classes of the same package, etc.) to simulate the expected access rights. `addBook` and `removeBook` cannot be private because they can be called by a method out of `Library`. `addBook` (respectively `removeBook`) cannot be public, otherwise `CensorBoard` (resp. `Publisher`) could use it. Protected does not apply here since there are no inheritance relations. If `addBook` and `removeBook` have package visibility, `Author`, `Publisher` and `Library` need to be in the same package to ensure access rights on `addBook`. Then `Publisher` automatically has access to `removeBook`, which contradicts the initial hypotheses.

These statements are, from our standpoint, important for fine-grained interface definition, and should really be expressed at the same abstraction level as static class diagrams. They concern class description as they describe the range of possible class collaborations.

Access Graph Diagrams

We have previously introduced Access Graph formalism [1, 4] to support language-independent reasoning on access control in programming languages (Java, C++, Eiffel, etc.). This formalism basically describes authorized accesses:

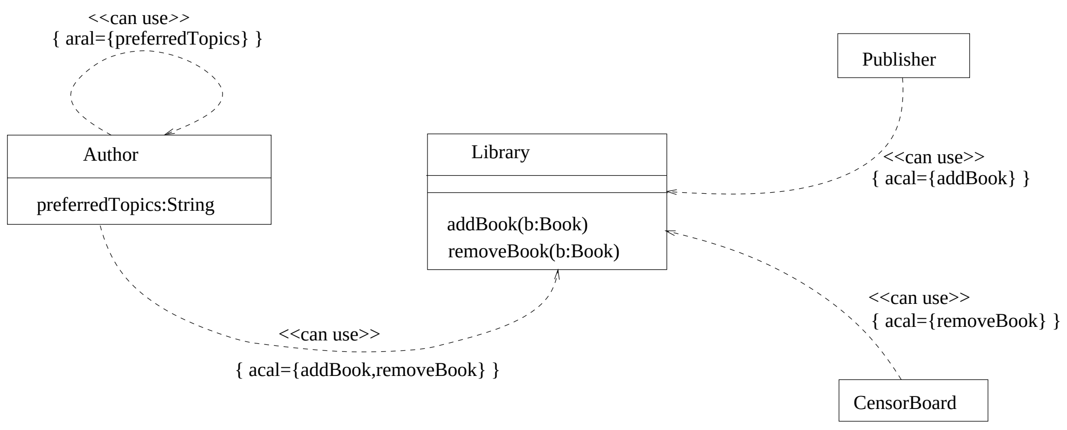

Figure 3: Access graph diagram

Stereotype	**BaseClass**	**Parent**	**Tags**	**Constraints**	**Description**
can use	Dependency	N/A	aral acal isInhS isInhT	tag aral only on loops; origin and destinations of the dependency are classes	Dependencies of this stereotype describe which visibility (authorized accesses) the origin has on features of the destination

Table 1: Stereotype definition

- receiver-level accesses, namely tuples (C_l, m, sa, p), where p is a feature accessible in method m declared in C_l, on the receiver of m, and the access kind is sa (taken among read/write/call/etc.); m is often omitted because usually all methods of a class have the same access rights; when there is no ambiguity, sa can also disappear, for example when there is only one kind of access right for a feature (*e.g. call* for a method);
  receiver-level access $(Author, preferredTopic)$ would represent our first statement;

- class-level accesses, tuples (C_1, C_2, m, sa, p) where m, declared in C_1, has access to p on an expression of type C_2, the access kind being sa; as previously, m and sa can be omitted in non-ambiguous situations;
  class-level access $(Author, Library, addBook)$ would represent part of our second statement.

This formalism has a graphical counterpart which we already used in previous work for the analysis of static access control in software [16]. The graphical form of access graphs makes them relevant for introduction in UML using standard extension mechanisms. Figure 3 shows the main elements of access graph diagrams (here combined with the class diagram). A new stereotype $<<$ *can use* $>>$ is attached to dependencies, and tagged values describe authorized accesses.

Table 1 details the stereotype while the tag definitions are given in Table 2. The two tables respect the format suggested in [25]. The first two tag definitions refer to access right representation: tag *aral* describes the authorized receiver-level access list attached to a << *can use* >> dependency (necessarily a loop); tag *acal* describes the authorized class-level access list. The third and fourth tags illustrate possible diagram simplifications that can be done for readability's sake: accesses can be source inherited (tagged value *isInhS=true*), or target inherited (tagged value *isInhT=true*).

Figure 4 presents a more detailed access graph diagram which will be used afterwards. Receiver-level access is omitted as we will focus, for simplicity's sake, on class-level accesses in the next sections. In this diagram, librarians can call `addBook` and `removeBook` on library sections as well as on "book of the month" sections since the access is target inherited (tagged value `inhT` for `isInhT=true`[3]). Only librarians have this access to `addBook`, on "book of the month" sections, since adding books to such sections has to be moderated. Publishers have access to method `addBook` restricted to library sections, and this access is source inherited (tagged value `inhS` for `isInhS=true`), for example science publishers also benefit from this access to `addBook` in library sections. Authors and fantasy authors have access to `addBook` and `removeBook` in library sections, but only to `removeBook` in "book of the month" sections. Censor boards have only access to `removeBook` in library sections and "book of the month" sections.

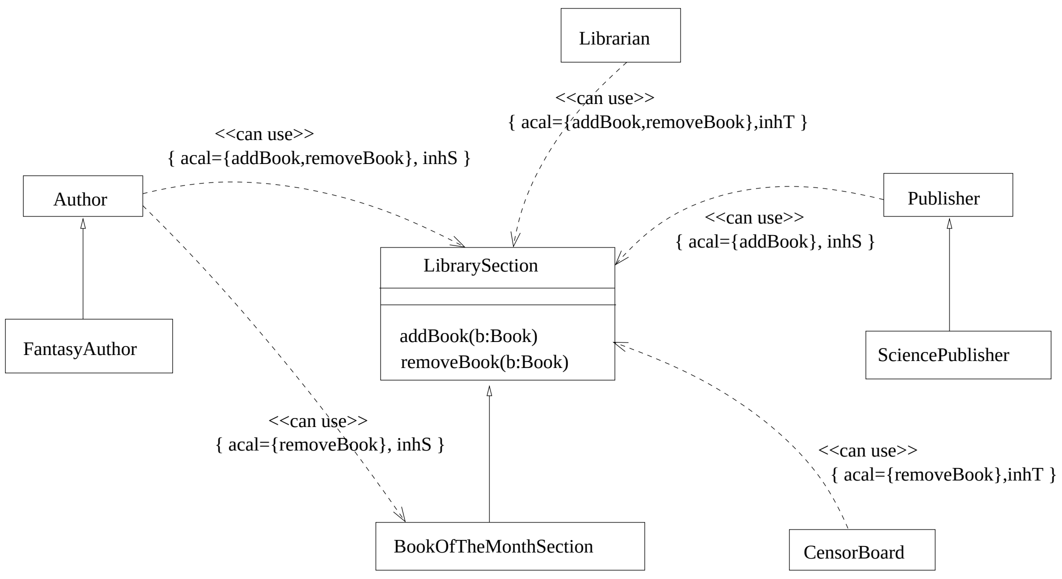

Figure 4: Detailed Access graph diagram for libraries

Several access dependencies can be drawn between two classes if this can improve readability. This paper will not go into all the subtleties of the notation, but we think that it is important to note that the fundamentals of access graphs are not very far from a graphical translation of the client clause of Eiffel. A difference is

[3]As recommended for Boolean properties in UML [25] paragraph 3.17.2.

Tag	Stereotype	Type	Mult.	Description
aral	can use	(m, sa, p) where m is a method, $sa \in \{call, read, write, ...\}$, and p is a feature. When non-ambiguous, m and sa can be omitted.	*	`authorized receiver-level access list` On loop concerning class C_l, indicates that the p —feature of C_l—is accessible for access kind sa through method m —declared by C_l— and on its receiver (e.g. in m the code $this.p$ is authorized access)
acal	can use	(m, sa, p) where m is a method, $sa \in \{call, read, write, ...\}$, and p is a feature. When non-ambiguous, m and sa can be omitted.	*	`authorized class-level access list` On edge from C_1 to C_2, indicates that the p —feature of C_2—is accessible for access kind sa through method m —declared by C_1— (e.g. in m the code $inst.p$, where $inst$ has type C_2, is authorized access)
isInhS	can use	Boolean	1	`Source Inherited` Written on edge from C_1 to C_2, indicates whether access right extends or not to subclasses C_3 of C_1 (the source). Dependencies from C_3 to C_2 with same tagged values can be omitted.
isInhT	can use	Boolean	1	`Target Inherited` Written on edge from C_1 to C_2, indicates whether access right extends or not to subclasses C_3 of C_2 (the target). Dependencies from C_1 to C_3 with the same tagged values can be omitted.

Table 2: Tag definition

that Eiffel imposes a few limitations to the client clause that access graphs ignore: mainly Eiffel imposes that accessibility is inherited [23], *i. e.* `isInhS` is always true. This Eiffel restriction is certainly a good usual design rule, but we think that access graph notation has to be general enough to take other languages that do not enforce this rule into account. Further development of the access graph notation should consider applying access dependencies to other elements or element groups [1], like packages for example.

3 CONSTRUCTING THE PIM: TOOLS FOR CLASS-BASED VISIBILITY DESIGN

We consider access graph diagrams to express high-level, expert domain-based, design decisions like those presented about the library system. Like any design diagram, access graph diagrams need to be elaborated from scratch, and then revised several times before obtaining domain expert and designer approval. In this process, tools that assist reflection about the artifacts are welcome. In the case of access graph diagram design, we propose two kinds of such tools: a first proposal is an automated access graph adaptation to general design rules selected by the designer (the *Rule Adapter*); a second proposal involves adding new classes, relevant abstractions of clients, which are integrated within the class hierarchy (the *Client Discovery Module*). Designers are expected to use these automated tools and then select and adapt the results that seem relevant.

The Rule Adapter

The Rule Adapter is a tool that transforms an access graph diagram according to a design rule. The following rules are examples that will add some accesses to the graphs:

- a subclass can have class-level access to at least all methods its super-classes have access to.
- if a class has access to a method m in a class B, then it should have access to the method m in every subclasses of B.
- a method beginning with "get" should be callable by the subclasses of the class declaring it.

The first rule is a very useful one to apply when considering the extensibility of classes, and is enforced in Eiffel. The second is called "preserving interfaces when sub-classing" and ensures that dynamically allowed accesses are statically allowed too. Java (using JDK 1.4) almost enforces this rule [2]. The third and last one is an illustration of naming conventions that can be used when designing and programming. Such conventions, while not specifically tied to access control, are quite usual and necessary.

The following are some rules that can remove accesses:

- every method beginning with “getOwn...” should only be accessible at the receiver-level,
- all attributes can be written by and only by the receiver.

These latter rules can be more problematic than the first ones because they restrict access, thus potentially prohibiting an operation that the access graph designer thought necessary. When an access is removed upon applying a rule, a warning is issued to notify the designer which design rule he initially broke. As our tools are not currently integrated in case tools, there is no user interaction yet. Some useful options in this case, besides automatic removal, could include:

- providing automatic replacement solutions (like `set` methods when attribute writing is necessary),
- letting the user keep the access if he feels the rule is more harmful than useful in this case while ensuring that the documentation will clearly state the anomaly.

In the Rule Adapter Tool, the rules are encoded in methods modifying the access graph. We are currently working on defining a simple language that will be expressive enough to declare such rules and generate the corresponding checking and modifying code.

The Client Discovery Module

The goal of the *Client Discovery Module* is to identify new classes which correspond to presumably relevant potential clients and to correctly insert these classes into the class hierarchy. These clients can be identified by an analysis of groups of authorized accesses. For example, authors (including specializations), publishers (including specializations) and librarians have all the access to `addBook` on library sections, suggesting the *contributor* notion. Our example was very simplified and, in the general case, `addBook` would be found within a group of features corresponding to library section contribution, like `listOfContributions(e:Entity)`, `conditionsForContributing`, etc. This identification of groups of common accesses can be done in a systematic way with the help of *formal concept analysis* (also known as Galois lattice/concept lattice construction). We briefly review the foundations of formal concept analysis applied in our context and explain how the tool works with them. Note that this module can be directly applied when authorized accesses are source inherited or from classes without subclasses, otherwise some adaptations might be necessary.

	LS|addBook	LS|removeBook	BS|addBook	BS|removeBook
Author	X	X		X
FantasyAuthor	X	X		X
Publisher	X			
SciencePublisher	X			
CensorBoard		X		X
LibrarySection				
Librarian	X	X	X	X
BookOfTheMonthSection				

Figure 5: A binary relation R describing authorized accesses for the library system

Galois lattices [6] are based on Galois connections very likely introduced in [8] as a generalization of works of the mathematician E. Galois. Their use for concept formation appears in [6] and they also have been disseminated under the name of concept lattices, or formal concept analysis [32, 11]. They are used in several domains such as knowledge representation, machine learning (conceptual clustering), data mining, classification and software engineering for different purposes including class hierarchy construction based on common features [12, 9, 13, 20, 19] and class hierarchy analysis based on feature usage [28] or method call patterns [5].

Formal concept analysis works with a context composed of three parts:

- a set E of *formal entities*; in our case the classes,
- a set F of *formal attributes*; in our case the targets of authorized accesses, that are generally the tuples (m, sa, p) for receiver-level accesses or the tuples (C_2, m, sa, p) for class-level accesses; simplified forms of accesses can also be considered, for example reduced to p;
- a binary relation R which associates entities with their attributes (classes to their authorized accesses).

Figure 5 shows such a context for our library example, with the simplified form of access targets (reduced to properties). In this Figure, LS stands for *LibrarySection* while BS replaces *BookOfTheMonthSection*.

Concepts are pairs (X, Y), where:

- X, the *extent*, is a subset of formal entities which are “covered” by the concept;
- Y, the *intent*, is a subset of formal attributes which are satisfied by all entities of the concept.

In our case, a concept associates a group of classes with the accesses authorized for all classes in the group. Each formal attribute will be composed of a class name and a property name separated by | respectively called *class part* and *property part*.

Constructing the whole concept (Galois) lattice exhaustively gives us the concepts together with an organization based on extent inclusion (or equivalently intent containment). A simplified form of formal concept analysis which only produces a relevant part of the lattice (namely the Galois sub-hierarchy) considers concepts that can be obtained using one of the following processes [17].

- Let C be a class, consider $\mathcal{A}(C)$ as the accesses authorized to C as an intent; to form the appropriate associated extent, we just have to pool all the classes that also have $\mathcal{A}(C)$ accesses;
  For example, let us consider the class $Author$. *Author* has access to $\mathcal{A}(Author) = \{$*LS*|*addBook,LS*|*removeBook, BS*|*removeBook*$\}$ which is the intent of a concept. We gather the classes which also have these accesses to obtain the associated extent: $\{Author, FantasyAuthor, Librarian\}$.
  ($\{$*Author*, *FantasyAuthor*, *Librarian*$\}$, $\{$*LS*|*addBook,LS*|*removeBook, BS*|*removeBook*$\}$) is a concept C_a.

- Let a be an access, consider $\mathcal{C}(a)$ as the classes that have a as an extent; the appropriate corresponding intent is then obtained by gathering all the accesses that share $\mathcal{C}(a)$ classes;
  For example, let us consider the access (to) *LS*|*removeBook*. The classes which have this access are $\mathcal{C}(a) =\{$*Author, FantasyAuthor, CensorBoard, Librarian*$\}$. These classes share not only *LS*|*removeBook*, but also *BS*|*removeBook*.
  ($\{$*Author, FantasyAuthor, CensorBoard, Librarian*$\}$, $\{$*LS*|*removeBook,BS*|*removeBook*$\}$) is another concept C_b.

Some concepts are indifferently obtained by the first or the second item of the process, but in some cases only one part of the process can lead to them [17]. As noted before, they are organized by intent inclusion, for example C_a is a sub-concept of C_b. The Galois sub-hierarchy (shown in Figure 6 for relation R) preserves the more informative concepts: the least concept where a given class appears and the highest concept where a given access is declared. It also has a node number bounded by $|E| + |F|$ which is very low compared to the node number of the whole lattice in $\mathcal{O}(2^{min(|E|,|F|)})$. For our purpose, the main point is that the Galois sub-hierarchy contains all possible groups of authorized accesses, thus revealing all potential client abstractions (see Figure 6) which we can interpret as new classes.

As we want to smoothly insert the new classes into the current class hierarchy, initial classes are linked to them in a simple way. An initial class is attached to the least new classes that have it in their extent. For example, *Author* is attached as a subclass of *LSContributor+LS&BSErasers*; *Librarian* is assimilated to the lower new class because the extent is reduced to $\{$*Librarian*$\}$. Transitivity edges are removed. The resulting hierarchy, integrating client abstraction and current classes

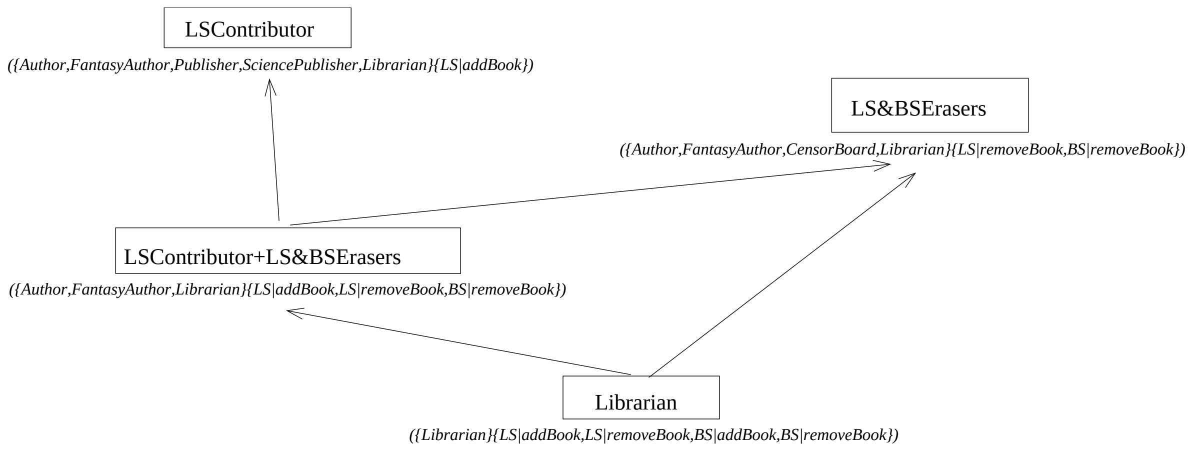
LSContributor
({Author,FantasyAuthor,Publisher,SciencePublisher,Librarian}{LS|addBook})
LS&BSErasers
({Author,FantasyAuthor,CensorBoard,Librarian}{LS|removeBook,BS|removeBook})
LSContributor+LS&BSErasers
({Author,FantasyAuthor,Librarian}{LS|addBook,LS|removeBook,BS|removeBook})
Librarian
({Librarian}{LS|addBook,LS|removeBook,BS|addBook,BS|removeBook})

Figure 6: Classes suggested by the Galois sub-hierarchy

is shown in Figure 7. On the access graph diagram, most accesses are shifted to higher classes (client abstractions). This simplification (due to access factorization) is the first advantage of the method. The second advantage is that the new concepts can have a relevant meaning for the ongoing design and include features (attributes or methods) in next design steps.

Figure 8 shows how the tool works: a pre-processing step extracts relevant access information of the access diagram and constructs a binary relation; this binary relation is processed by CERES [22, 18], a tool dedicated to Galois sub-hierarchy construction; then a post-processing step exploits the Galois sub-hierarchy to insert the new classes in the hierarchy and to factorize accesses, thus clarifying the access diagram. Accesses are generated by creating `can use` dependencies linking the concepts (interpreted as client classes) to the *class part* of the formal attributes of their concept. The *property part* of the formal attributes is added in the `acal` tag of the created dependency. This process is automatic in the current version of our tools. User selection of classes to be integrated in the hierarchy is possible and will be encouraged in future versions. For every concept class not selected by the user, the accesses and inheritance links just have to be redistributed to their subclasses, thus no data is lost.

4 PIM TO PSM TRANSFORMATION AND CODE GENERATION

In this section, we illustrate the mapping to a Platform-Specific Model with the Rule Adapter using Eiffel language rules and code generation from the obtained PSM.

Mapping to a PSM

As seen in the previous section, access graphs are useful for expressing domain-based and design-based platform-independent decisions concerning visibility. Then, these

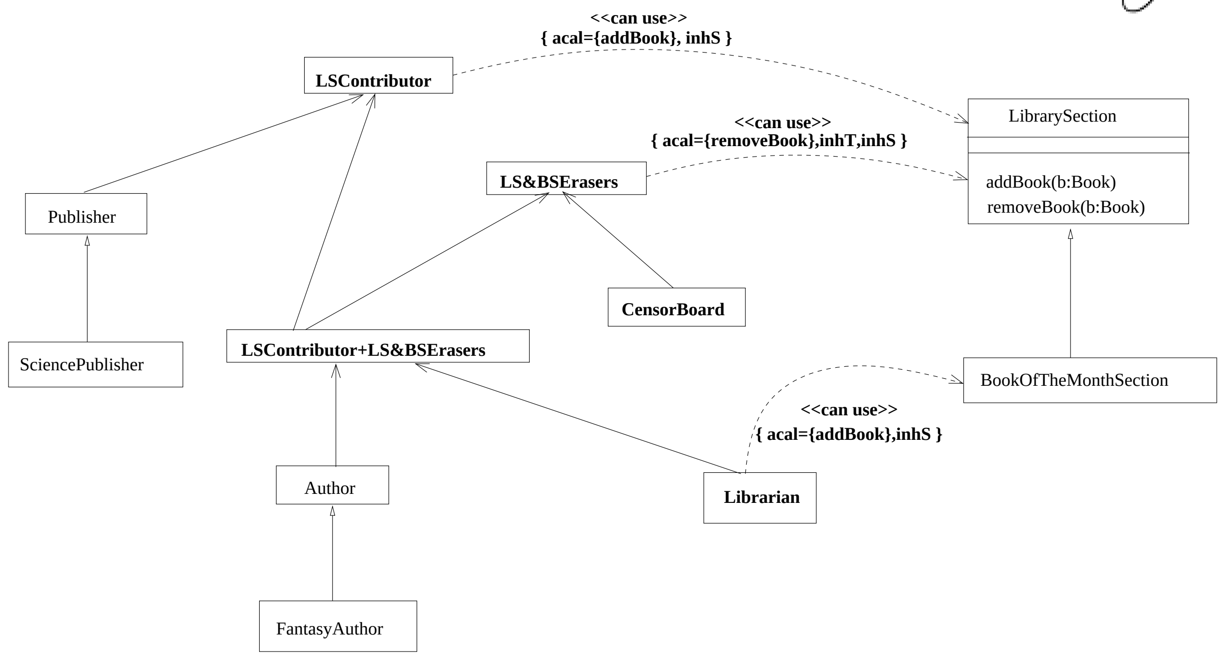

Figure 7: A new class hierarchy with client abstractions and access factorization

decisions are mapped to a platform-specific model that integrates specific constraints related to the target language. We will show here the mapping to an Eiffel-compliant model because of the expressiveness and simplicity of Eiffel.

Some very simple rules[4] must be verified for an access graph diagram to be compatible with Eiffel:

1. Attributes can be read and written at the receiver-level.
2. Attributes can only be written at the receiver-level.
3. Methods can always be called at the receiver-level.
4. A property accessible by a class A is also accessible by subclasses of A.
5. When not otherwise specified, the clients of a feature f in a class B are also clients of the feature f in subclasses of B.

As you can see, some of these rules where already mentioned in Section 3. Applying them for mapping to a platform-specific model does not differ from applying them on the Platform-Independent Model. Thus, we also use the Rule Adapter as the mapping tool.

Rules 1, 3, 4 5 only add accesses, but Rule 2 might remove some, and might require user interaction, as noted in Section 3.

[4]A more detailed description of these rules in the access graph formalism can be found in [4].

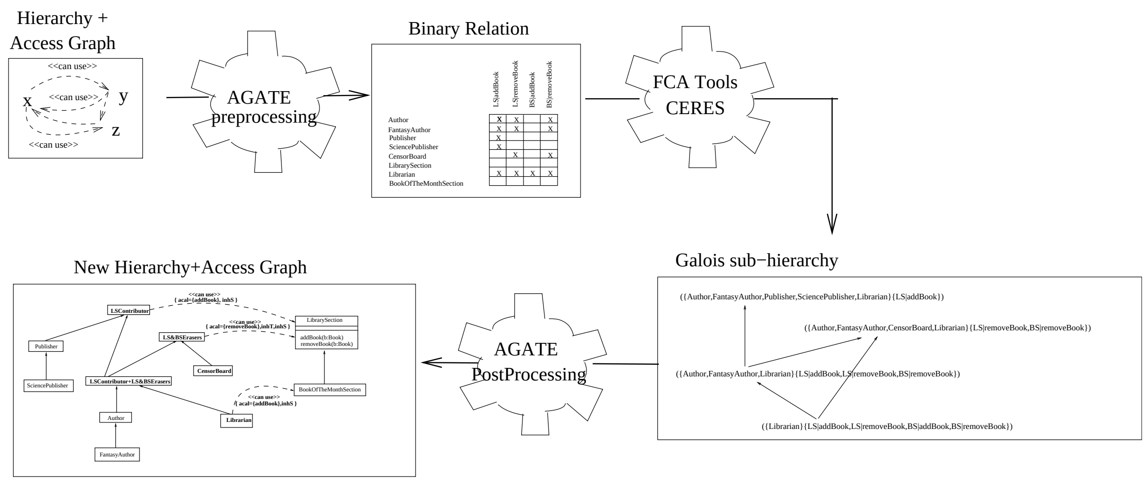

Figure 8: Tools for the Client Discovery Module

The example from Figure 7 contains only methods, thus rules 1 and 2 would not be used. Rule 3 would add the tagged values `aral={addBook,removeBook}` on arrows from `LibrarySection` and `BookOfTheMonthSection` to themselves. Rule 4 is already used by all `can use` dependencies because their tagged values have the `inhS` tagged value. Rule 5 is already applied because the `addBook` feature has its clients (`LSContributor` and `Librarian`) specified for both its owner classes, and the `removeBook` feature also because of the `inhT` tagged value on the `can use` dependency that targets it.

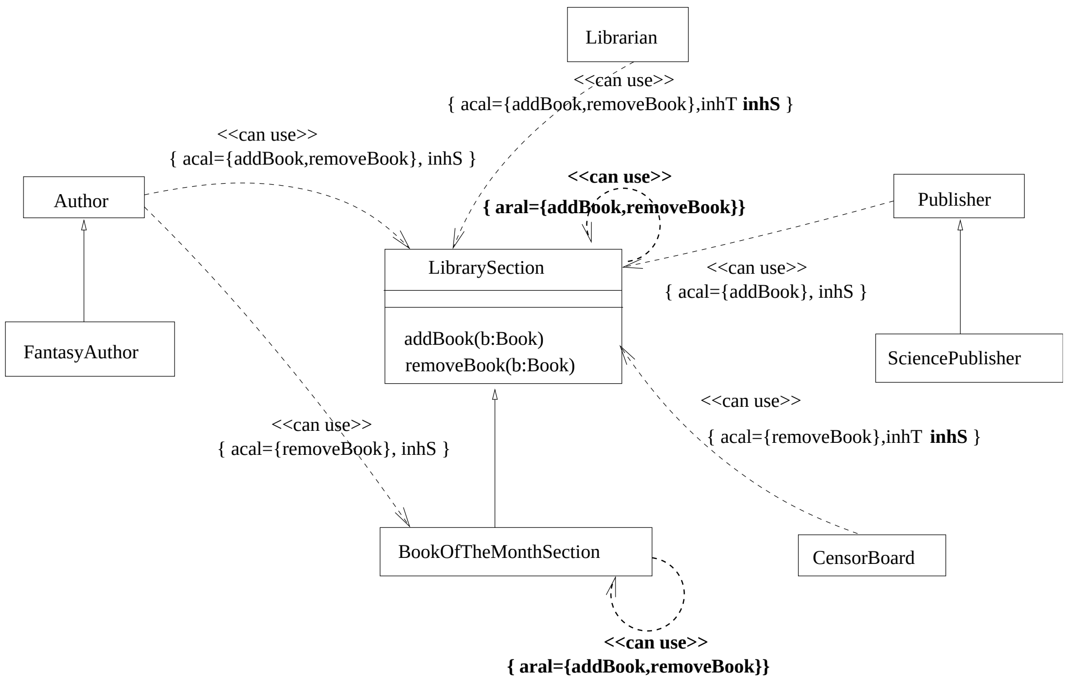

Figure 9: PSM: Access graph diagram for libraries in Eiffel

If the designer did not use the classes generated by the client discovery module, he can still adapt its original graph (cf. Figure 4) to the Eiffel language. The corresponding graph would be as shown in Figure 9.

Code generation

As stated in [27], "in a mature MDA environment, code generation will be substantial, or, perhaps in some cases, even complete". We will now describe the Eiffel code generation process which is currently implemented in our tool.

Generating Eiffel code from the above PSM is rather straightforward. The uniqueness principle advocated by B. Meyer [24] is very useful for such a transformation since it ensures that there are not numerous different ways to achieve the same thing, in our case visibility constraints.

The first step after adapting the graph to Eiffel is to generate a client clause for each feature. If a feature is inherited without being redefined and changes its visibility, then the `export` keyword is used to declare the client clause in the `inherit` clause.

This client clause contains all classes that can have access to the feature. If the `inhS` was not used in a `can use` dependency, then the corresponding client clause should be simplified by removing all classes that have a superclass in the client clause (*e.g.* a client clause containing `{Author FantasyAuthor}` is reduced to `{Author}`).

The code generated from the previous example is as follows :

```
class LIBRARY_SECTION
feature {LS_BSERASER} removeBook(b:BOOK) is
do
-- code
end;
feature {LSCONTRIBUTOR} addBook(b:BOOK) is
do
-- code
end;
end --LIBRARY_SECTION
```

```
class BOOK_OF_THE_MONTH_SECTION
inherit
  LIBRARY_SECTION
  export {LIBRARIAN} addBook
end
end --BOOK_OF_THE_MONTH_SECTION
```

The current version of the tool only generates skeleton code concerning visibility, but we plan on integrating it to a CASE tool and modifying the abstract syntax tree of the corresponding code, if it exists.

An algorithm for generating Java code from access graphs is described in [1] but has not yet been implemented. User interaction or automatic decision mechanism are needed because some visibility constraints where the `inhT` tag is not used can be encoded in different ways in Java.

5 CONCLUSION AND FUTURE WORKS

In this paper, we focused on visibility handling in the MDA process. We proposed access graph diagrams, an UML extension for expressing precise visibility constraints in models. Then we described two design tools operating on visibility at the Platform-Independent Model (PIM) level: the Rule Adapter applies design rules to the model while the Client Discovery Module detects new client classes which enhance the diagram and factorize potential collaborations. We illustrated the next steps in the MDA process with the mapping to an Eiffel-Specific Model (Access Graph Diagrams integrating Eiffel visibility rules) and Eiffel code generation.

This process has been tested using small-sized design diagrams. Since we currently do not have access graph design diagrams of realistic size, we plan to use our reengineering tool to extract them from Eiffel or Java applications. We envision two strategies: either using authorized accesses (which will be much more informative in Eiffel than in Java) or using effective accesses (method calls or attribute accesses).

In future works, we plan to extend our graphical notation for visibility by admitting groups of elements as source and target of << can use >> dependencies and to develop a simple declarative language for expressing design rules.

Our tools are currently part of the AGATE platform, but we intend to integrate them into a CASE tool to assist potential users in developing visibility constraints in a seamless process.

ACKNOWLEDGEMENTS

The authors would like to thank Thérèse Libourel for fruitful discussions about UML extensions and Jean Privat for his help in the development of Eiffel-specific tools. CERES is the result of research supported by France Télécom R&D.

REFERENCES

[1] G. Ardourel. *Modélisation des mécanismes de protection dans les langages à objets*. PhD thesis, Université Montpellier 2, 2002.

[2] G. Ardourel and M. Huchard. Formalizing static access control via access graphs: application to Java and Eiffel. Technical report, LIRMM UM2/CNRS, 2000. http://www.lirmm.fr/~ardourel/publis.html.

[3] G. Ardourel and M. Huchard. AGATE, Access Graph bAsed Tools for handling Encapsulation. In *ASE'2001 (International Conference Automated Software Engineering)*, pages 311–314, 2001.

[4] G. Ardourel and M. Huchard. Access graphs: Another view on static access control for a better understanding and use. *Journal of Object Technology*, 1(5):95–116, November 2002, http://www.jot.fm/issues/issue_2002_11/article1.

[5] G. Arevalo and T. Mens. Analysing Object-Oriented Application Frameworks Using Concept Analysis. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems - OOIS 2002 Workshops*, number 2426 in LNCS, pages 53–63. Springer, 2002.

[6] M. Barbut and M. Monjardet. *Ordre et Classification. Algèbre et Combinatoire*, volume 2. Hachette, 1970.

[7] S. Bennett, S. McRobb, and R. Farmer. *Object-Oriented Systems Analysis and Design using UML*. Mc Graw Hill, 2002. Second Edition.

[8] G. Birkhoff. *Lattice theory*. AMS Colloquium Publication, vol. XXV, 1940.

[9] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'96*, 31(10):251–267, 1996.

[10] M. Eichberg. MDA and Programming Languages. In J. Bettin, Ghica van Emde Boas, C. Cleaveland, and K. Czarnecki, editors, *Workshop "Generative Techniques in the context of Model-Driven Architecture", OOPSLA 2002*, nov 2002. http://www.softmetaware.com/oopsla2002/mda-workshop.html.

[11] B. Ganter and R. Wille. *Formal Concept Analysis, Mathematical Foundations*. Springer-Verlag, 1999.

[12] R. Godin and H. Mili. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. In *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'93*, volume 28, pages 394–410, 1993.

[13] R. Godin, H. Mili, G. Mineau, R. Missaoui, A. Arfi, and T. Chau. Design of Class Hierarchies Based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.

[14] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Computer Science. Addison-Wesley Publishing Co., 1984.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* Addison - Wesley, June 2000. http://java.sun.com/docs/books/jls/.

[16] O. Gout, G. Ardourel, and M. Huchard. Access Graph Visualization: a step towards better understanding of static access control. In Tom Mens, Andy Schürr, Gabriele Taentzer, editors, *proceedings of the International (ICGT 2002) Workshop on Graph-Based Tools (GraBaTs), Barcelona, Spain, October 7 - 8, 2002 Electronic Notes in Theoretical Computer Science, 72(2), Elsevier Science B. V.* http://www.elsevier.nl/locate/entcs/volume72.html

[17] M. Huchard, H. Dicky, and H. Leblanc. Galois lattice as a framework to specify algorithms building class hierarchies. *Theoretical Informatics and Applications*, 34:521–548, January 2000.

[18] M. Huchard and H. Leblanc. Computing Interfaces in Java. In *Proc. IEEE International conference on Automated Software Engineering (ASE'2000)*, pages 317–320, 11-15 September, Grenoble, France, 2000.

[19] P. Valtchev, M. Rouane Hacene, M. Huchard, and C. Roume. Extracting formal concepts out of relational data In *Proceedings of Fourth International Conference JIM'2003, September 3-6, 2003, Metz , France* Pages 37-48, INRIA publications, ISBN 2-7261-1256-0, 2003.

[20] M. Huchard, C. Roume, and P. Valtchev. When concepts point at other concepts: the case of UML diagram reconstruction. In V. Duquenne, B. Ganter, M. Liquiere, E. M. Nguifo, and G. Stumme, editors, *FCAKDD 2002, Advances in Formal Concept Analysis for Knowledge Discovery in Databases, Int. workshop ECAI 2002*, pages 32–43, Lyon, juillet 2002. http://www.lattices.org/FCAKDD2002.htm

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

[22] H. Leblanc. *Sous-hiérarchies de Galois : un modèle pour la construction et l'évolution des hiérarchies d'objets (Galois sub-hierarchies : a model for construction and evolution of object hierarchies).* PhD thesis, Université Montpellier 2, 2000.

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

[24] B. Meyer. *Object-Oriented Software Construction.* Professional Technical Reference. Prentice Hall, 2nd edition, 1997.

[25] Object Management Group. *Unified Modeling Language Specification (UML)*, September 2001. Version 1.4, http://www.omg.org/technology/uml/.

[26] OMG Architecture Board MDA Drafting Team. Model Driven Architecture. OMG document ormsc/2001-07-01, 2001.

[27] J. Siegel and the OMG Staff Strategy Group. Developping in OMG's Model-Driven Architecture. OMG document omg/2001-12-01, 2001.

[28] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22(3):540–582, May 2000.

[29] R. Soley and the OMG Staff Strategy Group. Model Driven Architecture. OMG document omg/2000-11-05, 2000.

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

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

[32] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets, in I. Rivals (Eds)*, 23, 1982.

ABOUT THE AUTHORS

Gilles Ardourel is an assistant professor at the LINA (Lab. CNRS & Science University of Nantes (ex-IRIN)). He received a PhD Degree in Computer Science from the University of Montpellier II in 2002. He is interested in object-oriented language design and modeling, primarily in the static access control subject. He can be reached at ardourel@lina.univ-nantes.fr. See also http://www.sciences.univ-nantes.fr/info/perso/permanents/ardourel/.

Marianne Huchard is an assistant professor at the LIRMM (Lab. CNRS & Science University of Montpellier). Her research interests include various aspects of inheritance (conflict resolution based on linearization methods, inheritance graph decomposition, formal concept analysis based techniques for class hierarchy and UML class diagrams restructuring) and static access control mechanisms. She leads at the LIRMM a working group dedicated to object orientation. She can be reached at huchard@lirmm.fr. See also http://www.lirmm.fr/˜huchard.