Electronic Notes in Theoretical Computer Science 74 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume74.html 12 pages

# High-level Transformations to Support Framework-Based Software Development

Tom Tourwé

*Programming Technology Lab*
*Vrije Universiteit Brussel*
*Pleinlaan 2, 1050 Brussel*
*Belgium*

Tom Mens

*Programming Technology Lab*
*Vrije Universiteit Brussel*
*Pleinlaan 2, 1050 Brussel*
*Belgium*

**Abstract**

In this paper, we show how elaborate support for framework-based software development can be provided based on explicit documentation of the hot spots of object-oriented application frameworks. This support ranges from automatically verifying whether appropriate design constraints are preserved, over providing high-level transformations that guide a developer when instantiating applications from a framework, to supporting software upgrades based on these transformations. The hot spots are documented by means of design patterns, and we use metapatterns as an abstraction to define the associated design constraints and high-level transformations.

## 1 Introduction

Over the past years, object-oriented software development based on framework technology has become extremely popular and has gained widespread acceptance. The major reason is that such development method offers significant software engineering benefits: it allows design reuse, as opposed to

[1] Email: tom.tourwe@vub.ac.be
[2] Email: tom.mens@vub.ac.be. Tom Mens is a postdoctoral fellow of the Fund for Scientific Research - Flanders (Belgium)

©2002 Published by Elsevier Science B. V.

mere code reuse, reduces application development time, promotes consistency between applications, and so on [5,10].

The most important asset offered by an application framework is its design, that should be flexible and reusable to allow developers to build numerous applications within the same application domain. The design defines the specific places where the framework can be extended with application-specific code (the so-called *hot spots* of the framework [11]) and imposes particular constraints upon the application's implementation. In order to behave as intended, the applications need to *fill in* the appropriate hot spots and adhere to the intended design, to ensure that no constraints are violated.

In practice, it turns out that the design of the framework is not adequately documented, and as a consequence, neither are the hot spots and the constraints [12,3]. As a result, the latter are only implicitly present in the implementation. It should thus come as no surprise that instantiating a correct application from a given framework is a complex and error-prone task. Many times, applications do not fill in the appropriate hot spots, or use these hot spots in the wrong way, and thereby violate the intended design of the framework.

To alleviate these problems, we propose to explicitly document a framework's hot spots. To a considerable extent, this can be achieved by using information about the design patterns used in the framework [3,7,2,9]. Not only are design patterns extremely well suited to ensure that the framework implements the appropriate hot spots in a flexible way, the information conveyed within a design pattern can also be used to document accurately these hot spots and to specify in which ways they can be filled in. As such, each design pattern comes with a number of high-level transformations, that prescribe the specific changes that should be applied to fill in its hot spots. These transformations can be explicitly and formally defined in terms of metapatterns and can be automated. They thus serve as a basis for an approach that supports building concrete applications.

In what follows, we will first show how to document the hot spots of a framework by means of design patterns. Then, we will explain how these design patterns impose design constraints and how they enable us to provide high-level transformations that can be used to support instantiating correct applications. Last, we will elaborate upon the actual implementation of the constraints, transformations and upgrading algorithm, by means of metapatterns.

## 2 Documenting Hot Spots with Design Patterns

### *2.1 Motivation*

Our choice for documenting the hot spots of a framework by means of the design patterns it uses is motivated by various considerations.

First of all, the authors of the famous design pattern book [6] state that "a design pattern allows a particular aspect of a system to vary independently of other aspects". In other words, design patterns are extremely well suited to implement the hot spots of a framework in a flexible way.

Second, design patterns are well known, well documented and expose a lot of useful and important information. They document the rationale behind a design, improve understandability of the design and the corresponding implementation, define a common vocabulary among developers, and so on.

Based on the two observations above, it should come as no surprise that design patterns are extermely popular, and that most of a framework's hot spots are implemented by using one or more design patterns. This leaves us quite confident that design patterns can be used to document the majority of a framework's hot spots. This is confirmed by several other authors who showed that design patterns are indeed perfectly well suited to document the hot spots of a framework in an adequate way [2,7,13].

### *2.2 An Example Design Pattern Instance*

As a concrete example, consider an instance of the *Composite* design pattern depicted in the upper left part of Figure 1. It shows part of a framework for building Scheme interpreters [1,13], more specifically, the `ScExpression` hierarchy that represents expressions in the Scheme programming language. The `printOn:` method defined in this hierarchy forms part of the implementation of the *Composite* design pattern and is used to print a textual representation of the objects.

### *2.3 Documenting Design Pattern Instances*

We document design pattern instances in SOUL, a Prolog-like logic programming language [4], that is tightly integrated with the Smalltalk programming environment. SOUL incorporates special provisions for accessing and generating object-oriented source code. We refer the reader to [15] for a more in-depth discussion of these features.

Each design pattern defines a number of roles (the *Participants* section of a design pattern description) that source code entities (such as classes, methods and variables) play in its instances. We use logic facts to explicitly map these roles onto the corresponding source code entities. For example, the above instance of the *Composite* design pattern is documented as follows by logic facts:

```
dpInstance(CompositeExpr,compositeDP).
dpRole(CompositeExpr,component,ScExpression).
dpRole(CompositeExpr,leaf,ScConsExpression).
...
dpRole(CompositeExpr,composite,ScSequenceExpression).
dpRole(CompositeExpr,compositeMethod,printOn:).
```

The *dpInstance* predicate is used to register that *CompositeExpr* is an

instance of the *Composite* design pattern. The *dpRole* predicate maps source code entities onto the roles of the design pattern. Its first argument is used to identify the particular design pattern instance that is being documented. Its second argument represents the design pattern role, while its third argument refers to the source code entity that plays that particular role.

The main motivation for documenting design pattern instances in this way in SOUL is that it enables us to guarantee the consistency between the information represented and the actual source code. This is possible because SOUL is tightly integrated in a standard development environment. Traditional documentation techniques often lack this feature, which is one of the major causes of documentation being outdated. Moreover, based on this documentation, elaborate support for framework-based software development can be provided. This support includes verifying that the appropriate design constraints are satisfied, guiding a developer in instantiating an application from the framework and detecting possible upgrade conflicts. In what follows, we will discuss some of these issues in more detail.

## 3 Design Pattern Constraints

### *3.1 General Remarks*

With each design pattern we can associate a number of *constraints* that should hold between its roles in order to guarantee a correct design pattern instance, and thus appropriate behavior of its implementation. Such constraints can be used to alleviate the problem of design drift [14], especially when a framework is evolved manually. Such problem occurs because a developer is not aware that the code he is changing forms part of the implementation of a design pattern. He may thus involuntarily break the intended design, which can result in the framework or its instances behaving in strange and unexpected ways. By explicitly defining the constraints associated with a design patterns, we can alleviate this problem, because we can verify whether the source code actually adheres to the appropriate constraints after changes have been made [9].

### *3.2 An Example Design Pattern Constraint*

Several constraints are associated with the *Composite* design pattern. It requires that the classes that correspond to the *composite* and *leaf* roles are (possibly indirect) subclasses of the class that plays the *component* role, for example. Additionally, the design pattern also imposes a particular implementation for the `printOn:` method in the `ScSequenceExpression` class. This method should iterate over all components and delegate the message to each individual component. Several other constraints can be defined for the *Composite* design pattern.

# 4 High-level Design Pattern Transformations

## *4.1 General Remarks*

Because design patterns are used to implement hot spots, they implicitly contain knowledge about how these hot spots need to be filled in. Such filling in consists of adding the appropriate participants to the various design pattern instances. This is often not a matter of one single change (e.g. adding a class), but may require many succesive changes in order to guarantee that the appropriate design constraints are preserved (e.g. adding a class, providing the appropriate method implementations, adding helper classes, etc.).

We make this knowledge and the corresponding changes explicit, by providing *design pattern transformations* that can be used to support a developer when instantiating the framework. Not only does this improve the quality of the resulting applications, since all required hot spots are filled in in the appropriate way, it also enables us to reason about the instantiation and evolution of the framework at a high level of abstraction.

Design pattern transformations are implemented by logic rules that use advanced code generation techniques to provide appropriate code automatically, consult the developer whenever necessary and automatically update the documentation [9]. We will elaborate upon this issue in Section 6

## *4.2 An Example Design Pattern Transformation*

We define two design pattern transformations for the *Composite* design pattern: an *addLeaf* and an *addCompositeMethod* transformation. Those transformations add new *leaf* and *compositeMethod* roles to a *Composite* instance, respectively.

As a concrete example, consider the left part of Figure 1, where a new expression type is introduced in the Scheme framework by means of an *addLeaf* transformation as follows:

```
dpTransformation(CompositeExpr,addLeaf,ScQuoteExpression).
```

The specific actions that this transformation carries out are the following:

- check some preconditions to ensure that the transformation can be applied. In this case, it verifies that *CompositeExpr* is indeed a documented instance of a design pattern, to which the *addLeaf* transformation can be applied, and that no `ScQuoteExpression` class exists in the framework.
- add the `ScQuoteExpression` class as a subclass of the appropriate superclass, `ScExpression` in this case.
- add an implementation for the `printOn:` method to the new `ScQuoteExpression` class, since this method plays the role of a *compositeMethod* in the design pattern instance.

Besides performing all these changes, the transformation also updates the documentation of the *CompositeExpr* design pattern instance as appropriate.

In this particular example, the specification is updated with the following information:

```
dpRole(CompositeExpr,leaf,ScQuoteExpression).
```

Note that it is essential that the documentation remains synchronized with the current implementation, since the constraints and the transformations heavily rely on this documentation to perform their task.

# 5 Support for Software Upgrading

## *5.1 General Remarks*

Clearly, the high-level transformations associated with design patterns are of great value for a developer who needs to fill in the hot spots of a framework. The fact that we have such explicit transformations at our disposal allows us to provide even more elaborate support for framework-based development. Large and complex frameworks and their applications are often developed by a team of developers. This often leads to situations in which one developer evolves the framework while another one instantiates it at the same time. We would of course like to upgrade the resulting application to use the new version of the framework. Such upgrading is far from trivial, however, and can lead to a number of *upgrade conflicts*, which need to be resolved in order to ensure the correct behavior of the resulting application [8].

## *5.2 An Example Upgrade Conflict*

As an example, consider the situation depicted in Figure 1. One developer evolves the framework by invoking a *pullUpMethod* refactoring to pull up the `printOn:` method to the `ScExpression` class (upper part of the figure). This refactoring forms parts of a larger evolution, where a *Visitor* design pattern is introduced to implement operations on `ScExpression` objects. At the same time, another developer instantiates the framework, and decides to add a `ScQuoteExpression` class to his application by means of an *addLeaf* transformation. This requires him to provide an implementation for the `printOn:` method as well (left part of the figure).

Clearly, an inconsistency is created by applying these two changes in parallel: in the new version of the framework, the `printOn:` method should no longer be defined in the leaf classes of the `ScExpression` hierarchy, whereas the application-specific `ScQuoteExpression` class still provides an implementation (lower right part of the figure). Such inconsistencies are called *upgrade conflicts*, and can easily lead to erroneous behaviour. In the above example, all classes in the `ScExpression` hierarchy use the *Visitor* design pattern to print a textual representation of themselves, except the `ScQuoteExpression` class.

Upgrade conflicts occur because one transformation relies on assumptions

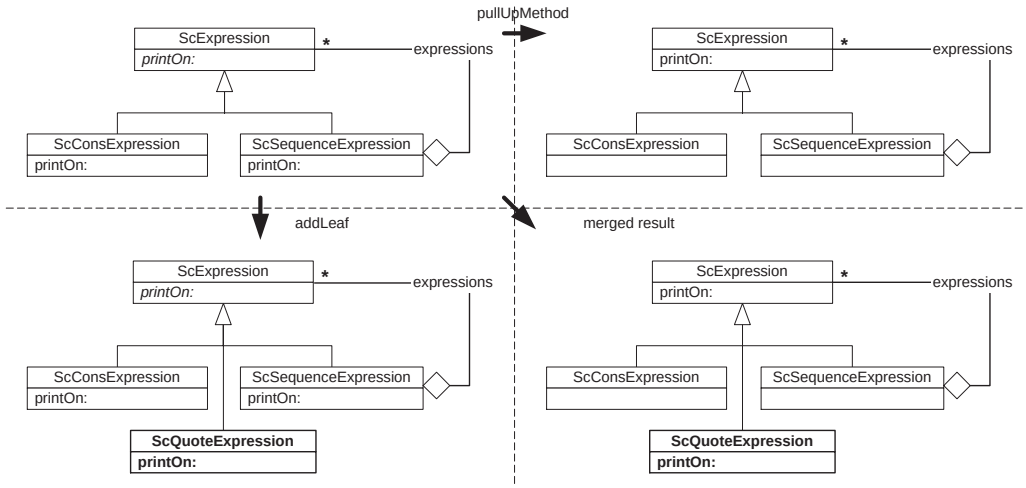


Fig. 1. Applying an *addLeaf* transformation and a *pullUpMethod* refactoring in parallel

that are broken by another transformation that is applied in parallel. In the concrete example presented above, the *pullUpMethod* refactoring pulls up the `printOn:` method, only considering the implementations currently present in the subclasses of the `ScExpression` class. By applying the *addLeaf* transformation, we introduce an extra subclass into this hierarchy, and thereby break the assumption made by the *pullUpMethod* refactoring. We will explain how such conflicts can be detected in Section 6.5, but first, we have to elaborate upon the implementation of our ideas.

# 6 Implementation

In this section, we will show how we implemented an environment that incorporates the support mentioned above. The underlying model of this environment is an advanced abstraction of design patterns, called metapatterns [11]. We will define what a metapattern is, show how a design pattern instance specification can be translated automatically into a metapattern instance specification, how metapattern transformations can be implemented and how an upgrade conflict detection algorithm can be defined based on comparing such transformations.

## *6.1 Metapattern Definition*

Metapatterns were conceived out of the observation that the implementation of many design patterns share a similar structure, and only differ in some specific details [11]. Therefore, an abstraction can be constructed, which proves useful for providing all kinds of tool support based on (the structure and implementation of) design patterns.

The definition of metapatterns is based on a distinction between *template*

and *hook* methods, the corresponding *template* and *hook* classes and the specific ways in which these classes are related:

- A *template* method is a concrete method that calls some other methods, which are the *hook* methods. Hook methods can be abstract methods, regular methods with a default implementation intended to be overridden, or template methods in their turn.
- A *template* class is a class that implements a template method, and similarly, a *hook* class is a class that implements a hook method.

Template classes need to be combined with hook classes, in order for template methods to be able to call hook methods. A metapattern is a particular combination of a template class and a hook class. Two aspects influence this combination:

(i) The cardinality of the association relationship between the template class and the hook class. An object of the template class may refer to exactly one object of the hook class, or it may refer to multiple objects of this class.

(ii) The hierarchical relationship between the template class and the hook class. The template class and the hook class may be unified into one class, or they may or may not be related via an inheritance relation.

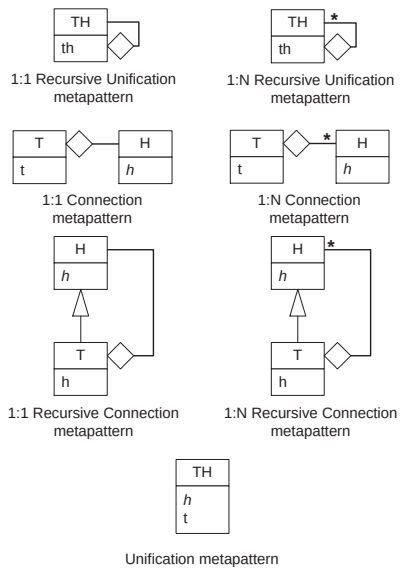


Fig. 2. The Existing Metapatterns

Figure 2 shows the metapatterns as defined by [11].

### *6.2 Documenting Metapattern Instances*

To enable the envisaged support for framework instantiation and evolution, we require metapattern instances to be documented explicitly. Instead of

specifying metapattern instances manually, their specification is derived automatically from the design pattern specifications. Mapping design pattern instance specifications onto the appropriate metapattern instance specifications is achieved by mapping the design pattern roles on the metapattern roles. Such mapping is implemented by logic rules, that translate the *dpRole* facts used in a design pattern instance specification into *mpRole* facts, that describe a metapattern instance specification.

```
mpPatternInstance(?dpPatternInstance,?mpInstance,1:NRecursiveConnectionMP) if
     dpPatternInstance(?dpInstance,compositeDP),
     generateMPInstanceName(?dpInstance,?mpInstance).

mpRole(?mpInstance,hookRoot,?className) if
     dpPatternInstance(?dpInstance,compositeDP),
     mpPatternInstance(?dpInstance,?mpInstance,1:NRecursiveConnectionMP),
     dpRole(?dpInstance,component,?className).
```

The above rules show how the *component* role of the *Composite* design pattern should be mapped onto the *hookRoot* role of the *1:N Recursive Connection* metapattern. Similar rules are defined that implement the mapping for the other roles.

Note that in this particular example, the mapping is quite straightforward. This is not necessarily the case in general, however. One design pattern instance sometimes needs to be mapped onto multiple metapattern instances. Our approach is general enough to handle such cases as well [13].

### *6.3 Implementing Metapattern Constraints*

For each metapattern identified, we defined a number of corresponding constraints. For example, the constraint that states that the implementation of the method fulfilling the *compositeMethod* role in the class corresponding to the *composite* role should iterate over all contained components and delegate the message can be implemented as follows:

```
mpConstraint(1:NRecusiveConnection,?mpInstance,<?class,?selector>) if
     mpRole(?mpInstance,hookMethod,?selector),
     mpRole(?mpInstance,templateRoot,?class),
     not(calls(?class,?selector,?selector))
```

The first two lines consult the metapattern instance documentation to fetch the appropriate class and method roles. The *calls* predicate used on the last line employs this information to verify whether the method actually sends the appropriate messages. It does this by consulting the implementation of the method in the underlying Smalltalk environment. More details on how this is achieved in practice can be found in [15]. If the method's implementation is not conform, a constraint violation conflict is reported.

### *6.4 Implementing Metapattern Transformations*

In [13], we define metapattern transformations for each of the metapatterns we identified. These metapattern transformations are the most primitive building

blocks of our approach, and can be combined to form design pattern transformations.

As an example, consider the implementation of the *addLeaf* design pattern transformation on an instance of the *Composite* design pattern. Such a transformation actually corresponds to an *addHookLeaf* transformation on an instance of the *1:N Recursive Connection* metapattern. This correspondence is reflected by the following logic rule [3]:

```
mpTransformation(?mpInstance,addHookLeaf,?className) if
     dpPatternInstance(?dpInstance,compositeDP),
     mpPatternInstance(?dpInstance,?mpInstance,1:NRecursiveConnectionMP),
     dpTransformation(?dpInstance,addLeaf,?className).
```

As can be seen, the application of a metapattern transformation is represented by the *mpTransformation* predicate, which is derived from the *dpTransformation* predicate, that represents the application of a design pattern transformation. The metapattern transformations are then implemented by translating such *mpTransformation* facts into appropriate *generate* facts. For example, the *addHookLeaf* transformation on the *1:N Recursive Connection* metapattern is implemented by means of the following logic rules:

```
generateClass(?className,?superclassName) if
     mpTransformation(?instance,addHookLeaf,?className),
     mpRole(?instance,hookRoot,?superclassName)

generateMethod(?className,?selector,?code) if
     mpTransformation(?instance,addHookLeaf,?className),
     mpRole(?instance,hookMethod,?selector),
     askUser('Implementation for ?className>>?selector? ', ?code)
```

The *generateClass* and *generateMethod* predicates form part of our logic code generator. They contain information that is necessary for generating classes and methods. The code generator merely assembles all this information and effectively adds the appropriate code to the implementation.

The first rule denotes the fact that a new class should be added when an *addHookLeaf* transformation is applied. The rule consults the metapattern instance specification to automatically derive the appropriate superclass for this new class. The second rule expresses that when a new hook leaf participant is added to the metapattern instance, it should provide an implementation for all registered hook method participants. To this extent, the rule consults the metapattern instance specification to fetch these hook method participants, and prompts the user to provide an appropriate implementation for them (by means of the *askUser* predicate).

All that remains is updating the documentation. This is achieved by using the following logic rule:

```
mpRole(?instance,hookLeaf,?className) if
     patternInstance(?instance,compositeDP),
     mpTransformation(?instance,addHookLeaf,?className)
```

[3] Note that once again this is a rather simple example. One design pattern transformation sometimes consists of many metapattern transformations, but this poses no problems for our approach

### *6.5 Detecting Upgrade Conflicts*

Upgrade conflicts are detected by mutually comparing transformations and defining the conditions that give rise to a problem when the transformations are applied in parallel. As such, a conflict table is compiled that relates all transformations and whose entries contain the predicates that detect the aforementioned conditions. For example, the confict table will relate the *pullUpMethod* refactoring and the *addHookLeaf* metapattern transformation, and will contain the following logic rule as an entry:

```
upgradeConflict(1:NRecursiveConnectionMP,?mpInstance,obsoleteMethod,<?leaf,?selector>) if
     mpTransformation(?mpInstance,addHookLeaf,?leaf),
     refactoring(pullUpMethod,?class,?selector),
     mpRole(?mpInstance,hookRoot,?class),
     mpRole(?mpInstance,hookMethod,?selector).
```

This rule states that applying an *addHookLeaf* transformation and a *pullUpMethod* refactoring in parallel leads to an *obsoleteMethod* upgrade conflict in the *leaf* role that is added.

## 7 Conclusion

In this paper, we have shown that explicitly and precisely documenting the hot spots of a framework allows us to provide elaborate support for framework-based software development. Such support includes automatically verifying whether design pattern constraints are preserved, guiding developers when implementing concrete applications by means of high-level transformations and providing support for framework upgrading. In practice, we used a declarative meta-programming environment to support our approach: hot spots are documented by declaring logic facts that express design pattern information, while the design constraints, transformations and upgrade conflict detection algorithms are implemented by means of logic rules. The scalability of the approach is ensured by using an advanced abstraction of design patterns, called metapatterns.

The approach we have discussed in this paper is actually part of a more general approach to support framework-based development, that is described in [13]. This approach supports instantiation as well as evolution of frameworks and has been tested on two medium-sized frameworks: a framework for implementing Scheme interpreters, and the well-known HotDraw framework. The results obtained are reported on in [13], and confirm our believe that the approach is feasible and can be used in practice.

## References

[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Kent Beck and Ralph Johnson. Patterns Generate Architectures. In *Proceedings of the European Conference on Object-Oriented Programming*, 1994.

[3] Greg Butler and Pierre Dénommée. *Documenting Frameworks to Assist Application Developers*, chapter 7. John Wiley and Sons, 1999.

[4] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard Reference Manual*. Springer-Verlag, 1996.

[5] Mohamed Fayad and Douglas C. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10), 1997.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[7] Ralph Johnson. Documenting Frameworks Using Patterns. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1992.

[8] Tom Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 1999.

[9] Tom Mens and Tom Tourwé. A Declarative Evolution Framework for Object-Oriented Design Patterns. In *Proc. Int. Conf. Software Maintenance*, 2001.

[10] Simon Moser and Oscar Nierstrasz. The Effect of Object-Oriented Frameworks on Developer Productivity. *IEEE Computer*, 29(9):45–51, 1996.

[11] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley/ACM Press, 1995.

[12] Mark Rettig. Nobody Reads Documentation. *Communications of the ACM*, 34(7):19–24, 1991.

[13] Tom Tourwé. *Automated Support For Framework-Based Software Evolution*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2002.

[14] Jilles van Gurp and Jan Bosch. Design Erosion: Problems & Causes. *Journal of Systems & Software*, 61(2):105–119, 2001.

[15] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2001.