A framework for Model Driven Production of Graphic Modeling Tools

Bassem KOSAYBA
Laboratoire d'Informatique Fondamentale de Lille (LIFL)
UMR CNRS 8022
Bâtiment M3
Université des Sciences et Technologies de Lille (USTL)
59655 Villeneuve d'Ascq Cedex
France
E-mail: kosayba@lifl.fr
Tel: 33 (0)3 20 43 69 40, Fax: 33 (0)3 28 77 85 37

Abstract

A new tendency in software engineering is to use models in order to better structure software development processes. In order to model systems, we need modeling environments. These environments should not impose to the software process their own concepts, but they must support the concepts related to the application domain targeted by these processes. Thus, the modeling environment must be adapted to user's domain instead of the user having to adapt himself to the environment.

We propose in this paper an approach for designing graphic modeling environments tailored at the same time to the domain and to the wishes of the user. We also present a framework that implement this approach and its associated tools. These tools allow the automatic generation of the modeling environment on the basis of the models that define this environment. We use the separation of concerns and model driven engineering approach to implement this framework.

1. Introduction

Models have become a necessity in the world of computer science because the number of applications explodes and engineers seek to preserve knowledge related to these applications either to reproduce them using new technologies or to integrate them in more complex systems.

Models are abstractions of systems. High levels of abstraction allow easier understanding and handling of systems. Moreover, new approaches like the *MDE* (*M*odel *D*riven *E*ngineering) [2] and the *MDA* (*M*odel *D*riven *A*rchitecture) [3] aim to automate the use of models. For example, one uses models to generate the whole or part of a system code.

Then, we need to provide modeling means which offer and allow handling of the concepts necessary to express the system descriptions. These concepts should be those of the system domain. Thus, they have precise meaning for the people working in this domain [1].

There are many domains of applications such a *e-business*, *component-based* applications, *telecommunication* applications, etc. The manual development of modeling environments for all of these domains is not feasible. Consequently, it becomes necessary to automate the building of these environments.

Several approaches to adapt a modeling environment to the user domain have been proposed. Most of these approaches allow to generate an unique graphic interface like the *EMF* (*E*clipse *M*odeling *F*ramework) [10] and the Eclipse platform [11] or like the approach discussed in [8]. Other approaches allow to customize the graphic interface. For each new domain, we have to develop manually a part of the graphic interface as in *GEF* (*G*raphical *E*diting *F*ramework) [9].

We propose in this paper an approach to systematically generate a graphic tool adapted to the user's need, starting from a model expressing the concepts of his/her specific-domain and a model describing the graphic view. Furthermore, we present a framework that offer a MDE implementation to this approach.

This paper presents in Section 2 our objectives and our approach. Section 3 describes the framework implementing our approach. Section 4 gives some examples to illustrate our approach and the use of our framework. Finally, Section 5 presents some conclusions and perspectives.

2. Objectives and approach

We define an approach for the definition and the production of graphic tools for domain-specific modeling. The model defining a modeling environment includes two concerns: the first relates to the application domain for which the tool is defined, the second relates to the graphical representation of the tool.

We insist on the separation between the definitions of these two concerns. This enables to capitalize them and to re-use them independently. Thus, one can change only the definition of the graphical representation in order to change its graphic interface without redefining the domain for which this tool was produced. Vise versa, one can re-use the definition of graphical representation in order to produce several tools for different domains. The overview of our approach is shown in Figure 1.

To address the definition of dedicated modeling environments, we need means that allow to describe the domains of applications and to define the tool graphic interface. The *MOF* (*M*eta *O*bject *F*acility) of the

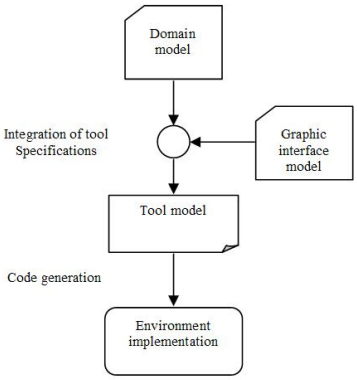

Figure 1. Approach overview

OMG (*O*bject *M*anagement *G*roup) is defined like an abstract language to define meta-models [4]. We can use the meta-models to express the concepts for particular domains. Furthermore, the MOF defines mapping rules that allow, starting from a meta-model, to obtain a repository interface. This repository enables to handle and to store models that are conforms to the meta-model for which this repository is produced. There are many projects like ModFact [5] that make it possible to produce implementations for the repository interface starting from a meta-model. So, the graphic tools which we want to produce, can benefit from these repositories to store the models.

3. The Framework

We provide a framework to produce a graphic modeling environment for a given domain. Figure 2 present our framework that is a MDE implementation of the Figure 1 approach. In this framework, we provide means that allow the tool designers to describe the application domains and to define the tool graphic interface. So, we define a GUI meta-model that enables to describe the desired graphic interface. Also, we define a model transformation that produces, taking as inputs a domain meta-model and a graphic interface model, a model of the intended tool. The tool code is generated on the basis of this model and through others transformation processes.

We support the framework meta-models by repositories. Thus, users can use these repositories to define the models specifying their tools. Also, the transformation processes can handle the models defined inside. We use the *Picore* tool [6] to produce prototypes for the framework meta-models repositories.

We have presented the framework schema in Figure 2 as it is divided into three principal parts. First part "GUI IM" includes the models which describe the domain of applications and the graphic model of the tool. These models specify the two concerns of the tool and are defined independently. Second part "GUI SM" includes the models which describe the tool. These models are specific to the graphic model of the tool. Third part "CODE" forms the different parts of the graphic tool code. Moreover, the program

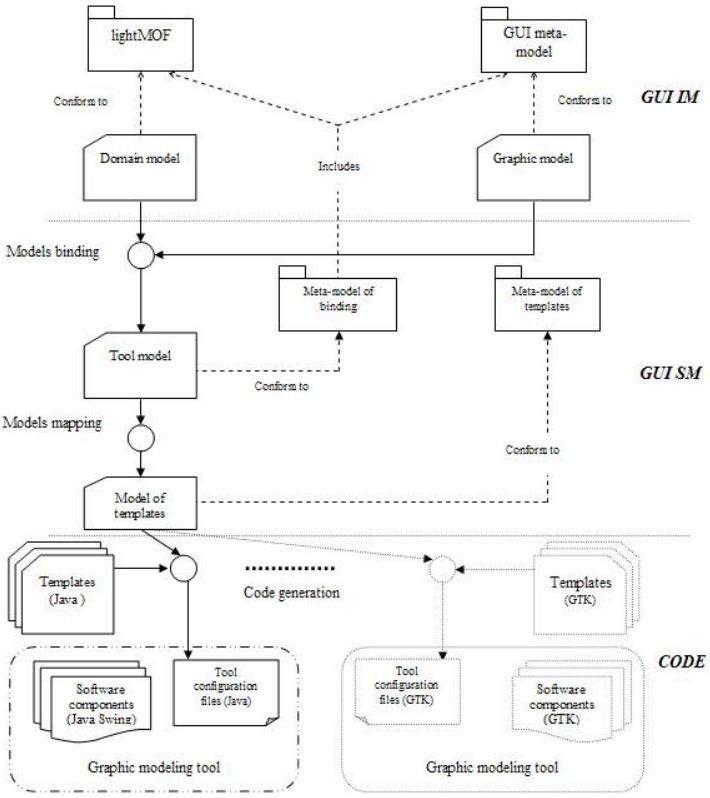

Figure 2. Framework schema

which gathers these various parts.

3.1. GUI IM Part

In this part, the separation between the domain definition and that of the graphic interface is always maintained.

3.1. 1.The definition of domains

In our framework, we offer a repository which makes it possible to handle the concepts of MOF. These concepts are necessary to define the domain models. The framework that we have defined does not take into account the whole concepts of the meta-meta-model MOF.

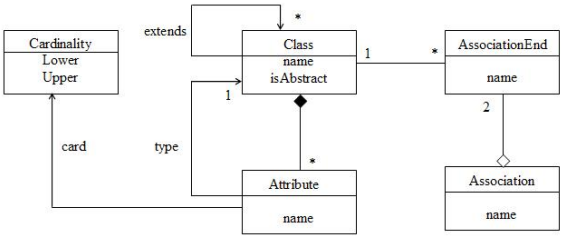

Figure 3. LightMOF

We support for the moment the following MOF concepts: *CLASS*, *ATTRIBUTE*, *ASSOCIATION*, *CARDINALITY* and the *INHERITANCE*. In most cases, these concepts are enough to specify the majority of domains. The framework can be extended

to support the other MOF concepts. Figure 3 shows the meta-model lightMOF which enables the domain models to be defined.

3.1. 2.The definition of graphical views

The graphic tools that we produce are for domain-specific modeling. Consequently, we identify two principal elements of the graphic interface of these tools. The first element must present the different concepts of the domain for which the tool was produced. The second element must present information around the model elements which the tool user defines.

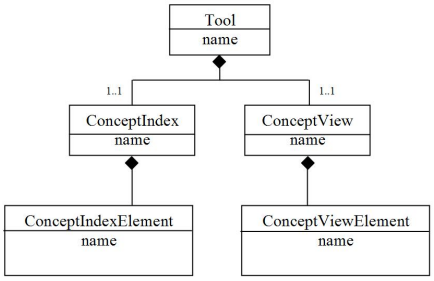

Figure 4. GUI meta-model

Figure 4 shows the GUI meta-model that we support in our framework. The concept "*Tool*" represents the principal frame of the tool, which contains the other elements. The concept "*ConceptIndex*" represents the element that enables to introduce the concepts of the application domain supported by this tool. The concept "*ConceptView*" represents the element making it possible to show information around the model elements which the tool user defines. The concept "*ConceptIndexElement*" represents the graphic element used to represent the domain concepts within the element implementing "*ConceptIndex*". The graphic element that implements this concept must offer the actions necessary to create instances of the domain concepts in the model defined within the tool. The concept "*ConceptViewElement*" represents the graphic element used to handle model elements within the element implementing "*ConceptView*". The graphic element that implements this concept must offer to the user actions that enable him/her to handle information around the model elements.

These concepts can be implemented in several ways. The choice between their implementations is done during the definition of a tool graphic model.

3.2. GUI SM Part

The models of this part are specific to the graphic model. In this part, the transformation process "models binding" integrates the domain model and the graphic model in the tool model. After that, the transformation process "models mapping" transforms this tool model in a model of templates in order to produce the tool using templates.

3.2. 1.Models binding

Each concept of the application domain must be associated to a graphical representation and an index. The graphical representation allows the tool user to display and to modify the attributes of an instance of this concept. The index allows the tool user to create an instance of this concept within the model defined using the produced tool.

We define in the "meta-model of binding" concepts to bind the concepts of the domain model and those of the graphic model. The process "Models binding" is driven by the meta-model of binding. It uses instances of this meta-model concepts in order to bind the elements of domain model with those of graphic model. Thus, the tool model is conform (compliant) to this meta-model. In this model, relationships are created between the domain concepts and the graphic model items. These relationships define how the domain concepts will be represented in the graphic interface.

3.2. 2.Models mapping

We define a meta-model of templates to organize the tool code as templates. This meta-model guides a model mapping process, which transforms the tool model into a model of templates in order to use the templates. This model of templates is an intermediate stage which goal is to support several technologies for the implementation of tool components.

3.3. Code Part

The tool architecture is divided into two principal units: the model repository and the graphic interface. The architecture of the tool follows the pattern *MVC* (*M*odel-*V*iew-*C*ontroler).

The model repository corresponds to the *Model* part of the pattern *MVC*. It store the model data which the tool user defines. Moreover, it is specific to the application domain. The graphic interface is related to the two parts *View* and *Controler* of the pattern *MVC*. It is customized with the domain model, the chosen graphic model and their mapping.

The model repository can be generated from the domain model which is a MOF meta-model. We use ModFact [5] to generate the model repository. ModFact can generate CORBA repositories as well as JMI (Java Metadata Interface) repositories [7].

There are two ways to support graphically the repositories generated by ModFact. In the first way, the graphic interface classes directly use the repository classes to store the model. Thus, the graphic interface classes must be specific to the domain model and the graphic model. Consequently, we generate all graphic interface classes. In the second way, we use an intermediate repository. This intermediate repository interacts with the repositories generated by ModFact using their reflective interfaces. The intermediate repository is composed of generic classes. Thus, this intermediate repository can be handled by the elements of the graphic interface independently of the domain which it describes. Consequently, we

generate the configuration of the intermediate repository elements in order to describe the domain for which a graphic tool will be produced. This section describes in details the second way to generate the graphic tools.

3.3. 1.Intermediate repository components

We have developed the components of this repository as generic classes. These classes represent the concepts of the MOF as *Class*, *Attribut*, *Association*, etc. They offer operations that enable to create relations between their objects. These relations look like those that can exist between the instances of MOF concepts. Thus, we can produce a repository specific to an application domain by the configuration of these classes instances.

3.3. 2.Graphical components

The graphical components are implementations for the concepts of GUI meta-model. They respect some interfaces which define their interactions with the intermediate repository.

We have provided two implementations for the concept *ConceptIndex* of the GUI meta-model. The first implementation offers a graphic view which introduce the domain concepts, such as a button list. The second implementation is a graphic view which enables to expose the domain concepts, such as tree leaves, and to show the elements of the model which the user tool defines.

We have also provided two implementations for the concept *ConceptView* of the GUI meta-model. The first implementation is a graphic view based on the notations as boxes and links. For a given domain concept which is defined using the concept *Class* of MOF, this graphic view draws the instances of this domain concept (inside the model defined using the produced tool) as boxes. For a domain concept which is defined using the concept *Association* of MOF, this graphic view draws the instances of this domain concept (inside the model defined by the tool user) as links. The second implementation offers a graphic view as a form set. These forms list the attributes of an element of the model defined using the produced tool. Moreover, these forms allow the tool user to handle the values of these attributes.

3.3. 3.Code generation

The definition of a meta-model of templates enables to implement a code generator. This generator uses information of the model of templates to generate the tool code. The result of the code generation process is a program for starting the tool. This program creates intermediate repository objects, the graphic interface elements and links between those two tool parts.

4. Experiments

In order to experiment our framework, we choose to produce graphic tools for the modeling of component-based applications. Moreover, we have produced a graphic tool for the modeling of application domains.

4.1. Modeling Environments for Component-Based Applications

In order to use our framework to produce modeling tools for component-based applications, we have defined a domain model for the component-based applications.

4.1. 1.The domain model for component-based applications

To define a component-based model for an application, we define its components and their connections.

A component can own attributes that describe the state of its instances. It can also have ports. The ports represent the connection points for a component. There are two types of ports. The provided ports specify a set of operations provided by a component. The required ports specify a set of operations required by a component. An operation is specified by a name and a return type. It can also need several parameters. A connection between two components is done through a connection between a port provided by a component and a port required by another.

Figure 5 presents our component model. We express this model using the lightMOF concepts in order to use our framework.

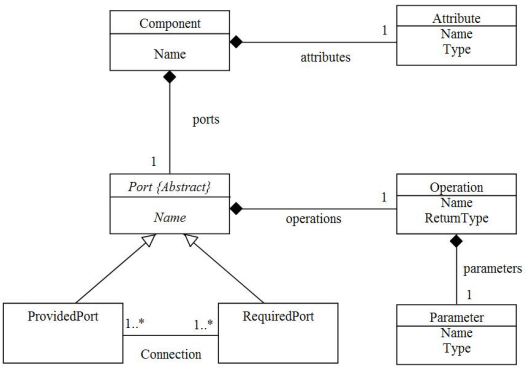

Figure 5. Component model

For example, we use the tools that we have produced for this domain in order to model an application of automatic distributor. The components of this application are: *Distributor*, *Client* and *Owner*.

The component *Distributor* has two ports: *Display* and *Provision*. The port *Display* provides operations to the component *Client*. The port *Provision* provides operations to the component *Owner*. The component *Client* has a port *Choice*. This port is connected to the port *Display* of the component *Distributor* by a connection *Buy*. This port has an operation *chooseArticle*. The component *Owner* has a port *Visit*. This port is connected to the port *Provision* of the component *Distributor* by a connection *Passage*. The port *Visit* contains the operations *loadArticles* and *emptyCase*.

4.1. 2.Graphics models and modeling environments

We define a graphic model in which domain concepts are represented by a button list and information related to model elements are handled through a drawing board. Figure 6 presents the modeling tool produced from the component model and the graphic model. It shows the model of automatic distributor defined by the tool user.

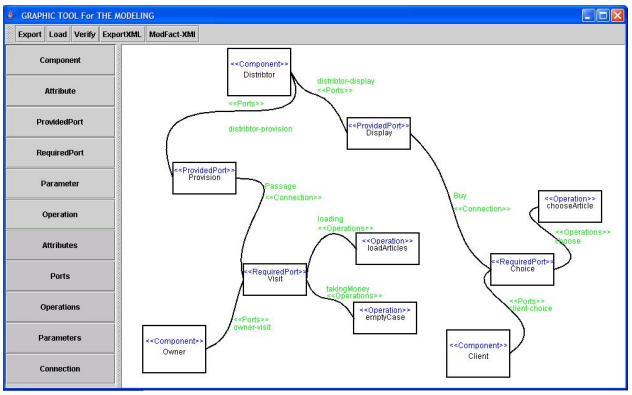

Figure 6. A tool for modeling component-based aplications

We define another graphic model in which domain concepts are represented by a tree leaves and information related to model elements are handled through a drawing board. Figure 7 presents the modeling tool produced from the seam component model and this another graphic model.

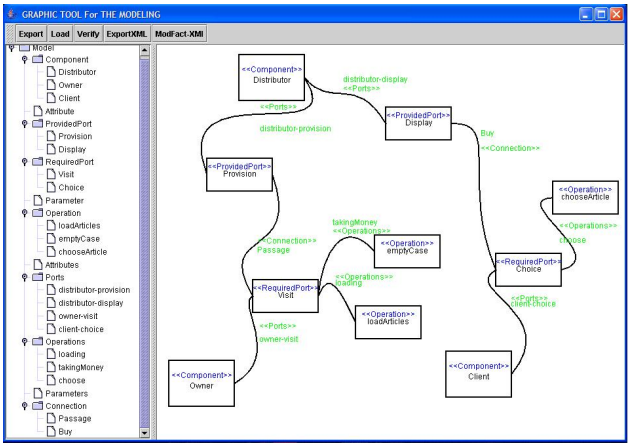

Figure 7. Another tool for modeling component-based aplications

4.2. Modeling Environment for LightMOF

In order to use the framework generators to produce a graphic tool, we need the domain model for which this tool will be produced. The repository generated by *Picore* for lightMOF meta-model can be used to define the domain model. Another alternative is to define the domain model in *XMI* directly.

We do not want to impose to our framework users to use the repository technology or *XMI*. They need only to know their domain concepts and how can be expressed with lightMOF. So, we provide a graphic environment for modeling the application domains. This environment allows our framework users to define domain models. After that, they can use the models defined by this environment as entries for the framework generators.

4.2. 1.Production approach

As a proofs concept, we have produced this environment by our framework itself. We have defined the lightMOF meta-model as a model conform (compliant) to lightMOF itself. After that, we have produced the graphic environment using this model as entry for the framework generators. This solution allows the graphic tool designers to benefit from services that the produced tools offer. For example, the service that checks the models validity according to the meta-model for which the tool was produced. Then, this environment asserts that the domain models are valid and conform (compliant) to lightMOF.

4.2. 2.A graphic model and the modeling environment for our framework

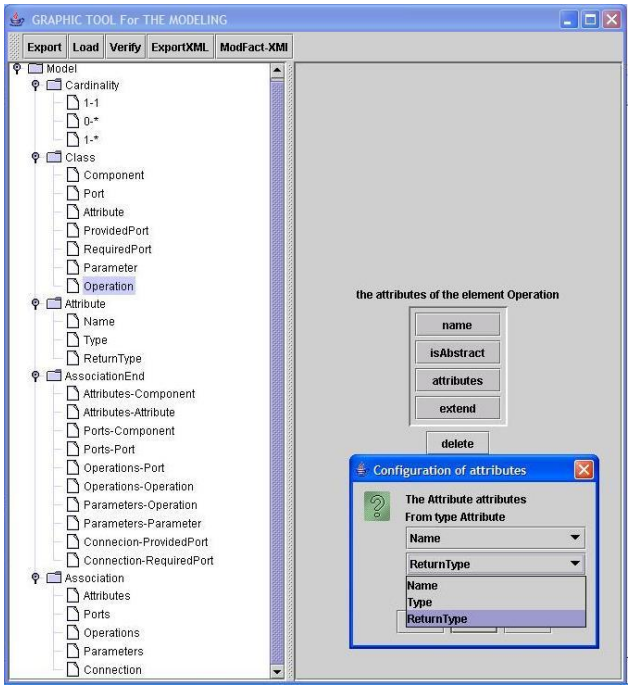

Figure 8. A graphic tool for modeling domain models

We use, to generate the modeling tool dedicated to lightMOF, a graphic model in which domain concepts are represented by a tree leaves and information related to model elements are handled through a form set. Figure 8 shows the modeling tool dedicated to lightMOF while one uses it to define the domain model for component-based applications.

5. Conclusion and Perspectives

The proposed framework allows one to obtain a graphic modeling tool more easily, starting from the definitions of the domain and the graphic interface. Moreover, we provide a graphic tool in order to simplify the definition of particular domains. This graphic tool was produced by our framework itself.

In this framework, we use meta-models to specify the domains for which one wants to produce graphic environments. Furthermore, we provide a GUI meta-model which defines the concepts of these graphic interfaces and we use its models to specify the tool graphic interfaces.

The MDA/MDE approaches are intended initially for the development of an application. They are aimed at organizing this development around several models and transformation processes. Then, one can reuse these models and transformations to easily make evolve the development of an application. We follow these approaches in our framework for the definition of the modeling environments.

Moreover, the separation between the domain description and the view description makes these definitions reusable. Consequently, it is possible to easily produce various graphic interfaces for the same domain and to use the same graphic interface definition for various domains.

There are perspectives for our work. The GUI meta-model defines only the abstractions of the graphic interface front. We can extend this meta-model in order to add other abstractions as the dialog boxes shape, the elements layout inside the graphic interface, the form of model elements, etc.

Moreover, we can identify new concerns of the produced tools. For example, the specification of actions which can be applied on model elements. This specification enables to apply simulations and checks on models. For each new concern, we must define a meta-model to capture the abstractions of this concern and we must extend the meta-model of binding in order to integrate a definition of the new concern into the tool model.

The collaboration of several persons for the definition of a model is sometimes necessary. Moreover, these persons may not be in the same place. The graphic tool, that can support such collaboration, must be divided in several graphic interfaces and a central repository. Each graphic interface allow the user to define a part of the model. The central repository can gather the various definitions made in the different graphic interfaces. The division of the domain model into several parts is necessary to define such tools. Each part of the domain model would describe the concepts supported by a graphic interface. As well as, the relations used to link the different parts of a domain model, can be exploited by the produced tool in order to integrate the various definitions into the central repository.

6. Acknowledgements

I would like to thank the professor Jean-Marc GEIB for his direction and all his advices enriching this research. I would like to thank the associate professor Raphaël MARVIE for his help in building the framework, for his advices, for our discussions, and for his reading of this paper. I also thank Anne-Françoise LE MEUR and Richard OLEJNIK for their reading of this paper.

7. References

[1] Janne Luoma, Steven Kelly and Juha-Pekka Tolvanen, Defining Domain-Specific Modeling Languages : Collected Experiences, *4th OOPSLA Workshop on Domain-Specfic Modeling (DSM'04) - October 2004*

[2] S. Kent, "Model Driven Engineering", *Third International Conference on Integrated Formal Methods - 2002*

[3] "Model Driven Architechure", *http://www.omg.org/mda*

[4] "Meta Object Facility(MOF) Specification", *http://www.omg.org/docs/formal/02-04-03.pdf*

[5] "ModFact Project", *http://modfact.lip6.fr/indexStatic.html*

[6] Raphaël Marvie, "Python pICO REpository generator", *http://www.lifl.fr/ marvie/Software.html#picore*

[7] Ravi Dirckze, Unisys Corporation, "Java Metadata Interface (JMI) Specification - 2002", *JSR 040 Java Community Process, http://www.jcp.org/*

[8] Mikaël Peltier, "Transformation entre un profil UML et un meta-modele MOF, Application du langage MTRANS - 2002", *Langages et Modles Objets (LMO'2002) Montpellier, France*

[9] Bill Moor, David Dean, Anna Gerber, Gunnar Wagenknecht, philippe Vanderheyden, "Eclipse development using the Graphical Editing Framework and the Eclipse modeling Framework - 2004", *IBM Redbook*

[10] "EMF Eclipse Modeling Framework", *http://www.eclipse.org/emf/*

[11] "Eclipse Platform Technical Overview", *http://www.eclipse.org/whitepapers/eclipse-overview.pdf*