ECONET Project

NANTES 2008 - WORKSHOP REPORT

Pascal ANDRE[1] Dan CHIOREAN[2] Frantisek PLASIL[3] Jean-Claude ROYER[4]

2008, 12-16 May

supported by

ÉGIDE

[1]LINA CNRS UMR 6241 - 2, rue de la Houssinière, B.P.92208, F-44322 Nantes Cedex 3, France
[2]Computer Science Research Laboratory, Universitatea BABES-BOLYAI Mihail Kogalniceanu nr. 1 RO- 400084 Cluj-Napoca, Romania
[3]Distributed Systems Research Group, Charles University, Malostranske nam.25, 11800 Nantes 1, Czech Republic
[4]OBASCO - EMN/INRIA LINA FRE CNRS 2729, 4, rue Alfred Kastler F - 44307 Nantes cedex 3 France

Executive Summary

An Egide-sponsored workshop was held at the Nantes Laboratory for Computer Science -in french Laboratoire dŠInformatique de Nantes-Atlantique (LINA CNRS UMR 6241)- in Nantes. This workshop was the second one in a series of the ECONET Project Nr 16293RG entitled, "Behaviour Abstraction from Code, Filling the Gap between Component Specification and Implementation".

The LINA laboratory in "Sciences and technologies of the software" is specialized on two axes : distributed software architectures and computerized decision-making systems. Associated to the CNRS, the University of Nantes and the Mines School (EMN - Ecole des Mines de Nantes), the LINA also includes two INRIA projects.

The first workshop provided a detailed outline for the project defining the objective and means, and structuring it in three subprojects. This second workshop is a milestone in the second year project. It should observe the project state and refine objectives and cooperation, according to the objectives of the two years of the project. We remind here a list of the main tracks we had to follow

- Present the current situation for each subproject (including products and problems, future work),
- Tools normalisation (compare tools and techniques of each subproject, final decisions on the tools panel, perspectives),
- Study the interface between the parts (languages, format, filters, API...),
- Get a first prototype (source and documentations for each subproject, final decisions on the metamodel part, extract the main open issues, applications on CoCoME)
- Draw the roadmap to the end of the year (development, documentation, workshop preparation, publication of reports and papers)

More precisely, the aims of the workshop were (1) to get some feedbak of the current developments (2) to share the experiences and (3) to settle interfaces and common tools. Additionally we would to take concerted decisions on the project issues (concrete objectives, tasks, organisation, responsabilities, deliveries, planning...).

On these points the workshop put forward new advances but also some delay of subproject tasks and discussions led to some decisions on both the interaction points and project organisation. The following issues have been discussed: tools and approaches, interaction points (metamodel, annotations), shared techniques and tools, common benchmark, etc. The working sessions enabled (1) to validate the common component metamodel (its specification is on the way), (2) to refine the subproject objectives and context, (3) to plan the work (subproject objectives and responsabilities) until the next milestone (Cluj's workshop in september), (4) to draw some project continuation (publications, projects).

The main concrete results are A project architecture was drawn after fruitful exchanges accompanied with the definition of tasks, with balanced responsabilities and partnerships. This project includes three distinct but complementary parts:

- A definition of the common component metamodel.
- A new definition of the annotation language.
- A gained experience on model driven tools and code processing.
- A finer architecture understanding.

The workshop concluded with some guidelines to the next workshop that should take place in Cluj 2008.

This report relates what happened in the Nantes's workshop (2008).

Acknowledgements The participants would like to thank Egide for its financial support of this workshop.

Contents

Chapter 1

Introduction

In this part we remind the context of the workshop, its preparation, organization and the program. This workshop was the second one in a series of the ECONET Project Nr `16293RG`.

1.1 The `16293RG` ECONET Project

The activity described in this report is supported by Egide in the context of ECONET Projects[1]. This section gathers the main features of the `16293RG` ECONET project.

- Title: Behaviour Abstraction from Code
- Subtitle: Filling the Gap between Component Specification and Implementation
- Type: Research and Technology Development Project
- Duration: 2 years
- Domain: Sciences and Information Technology
- Partners: COLOSS (French) - DSRG (Czech) - LCI (Romanian) - OBASCO (French)

1.1.1 Motivations

The project takes place in a specific domain of Information Technology, called Component Based Software Engineering whose goal is to provide languages, methods, techniques and tools for software developpers. The field of component-based software engineering (CBSE) became increasingly important in software construction approaches because it promotes the (re)use of components, also called Components Off The Shelf (COTS), coming from third party developers to build new large systems. Components are scalable software modules (bigger units than objects in object-oriented programming) that can be used at the high levels of abstraction (software architectures, design) and the low levels (programs, frameworks).

Component-based software engineering is still challenging in both industrial and academic research. Most of the academic approaches focus on abstract models (sometimes close to architectural description languages) with checkable properties such as safety and liveness; some of them deal with refinement and code generation. As a counterpart, the industrial proposals such as CORBA, EJB, OSGI or .NET are merely implementation-oriented and also object-oriented. They define flat components (without hierarchical structures) and the model is based on an underlying infrastructure for component repositories and communication management. They often lack of abstraction means to promote the reuse of components. Moreover, at the implementation level of a component based development, some implementations have nothing to do with the above industrial standards in the sense that there are no components at all. The main reason is that there are no true component programming languages yet (a language such as ComponentJ is a layer on Java). In other words, there are various component models that cover the whole software development process but there is a gap between component specifications (the academic models) and component implementations (industrial infrastructure or object-oriented implementations). The above

[1] http://www.egide.asso.fr/fr/programmes/econet/

mentioned problem is due to the fact that, usually, component implementation is not based on a rigorous specification. In cases when the specification precedes the implementation, the conformance between implementation and specification is seldom realized.

A major problem is then to fill this gap. One way is to define model transformation techniques in order to generate a code for the component with respect to the component specifications. This way can be qualified as the *engineering* way and it is similar as MDA and MDE approaches. It is quite complex since we should, in theory, prove the correctness of the translation and also because there are various target frameworks and languages. There are ongoing works on that direction [PNPR05, PP99].
Another way is to focus on program code analysis in order to compare component's actual code with its high-level (abstract) description. This way can be qualified as the *reverse engineering* way. It is quite an open issue in the current research on CBSE [BHM06, PP07]. This problem is even more complex than the one above, due to the following reasons :

- Often the source code of a component is not available after its deployment or even not physically available in a remote service invocation or Web Service. However, for a component industry the unavailability of source code is essential – services may even be offered on a pay-per-use basis.
- In case of OO implementations, the absence of component structures implies to find convenient and adequate criteria to structure components.
- Many statements and message send are to be omitted for a relevant service identification.
- There are no common component model (or standard) for the component (abstract) specification – many targets for reverse engineering.

Service clients have to properly intercat with the services and need to know at least the interface but in most cases the dynamic behaviour or protocol attached to the services. From that some compatibility checking and consistency controls may be performed to ensure a good interaction or to avoid wrong or illegal use of the services. Both the engineering and reverse engineering approaches remain research open issues.

The goal of the project is to contribute to the reverse engineering way by developing techniques for extraction of abstractions from code (including some component interface description) and for the verification of abstractions against the code, *e.g.* to check an in-line bank service with no available code, to check that a client component is compatible with an implemented component.

The core project is to establish a link between component codes and component specifications. The advantages of abstraction are to check the conformance of component codes and component specifications, to statically check various properties of the components such as safety and liveness. To be pragmatic we have to restrict this huge mapping according to the partner's experience.

1. The source model (implementation level) is limited to Java code. The problem of obtaining an abstract specification of a component from its code, cannot be solved in a satisfactory manner if the code does not contain appropriate comments, rather in well defined patterns, or if the code is not limited to a consistent subset of concepts.
2. The target models (specification level) are abstract component models inspired from the ones of the partners. Instead of studying only the structural features of the system, we plan to work on *behavioural* abstraction from Java code. Behaviour [PV02, AAA06a, PNPR05] is related to the dynamic and functional features of the components and services. In particular, dynamic behaviours describe the dynamic evolution of components, connectors or services (interactions). The mechanisms used for component specifications are grounded on different formalisms: design by contract (implemented by assertions), algebraic specifications, state machines, regular expressions and so on. Each above mentioned formalism offers a set of advantages and has some drawbacks. Design by contract, a declarative specification only, supports an "incomplete" behaviour specification. Algebraic specifications generally have sound semantics but are, in most cases, difficult to understand by people working in the industry and not all kind of components can be specified. The state machines and regular expressions formalisms are suited for dynamic descriptions and have formal semantics.

1.1.2 Partners

The partners are four research teams which have competences on the project topics.

- **COLOSS**: COmposants et LOgiciels SûrS
  Reliable Component and Software ⇝ Component System Specification and Verification
  `http://www.lina.sciences.univ-nantes.fr/coloss/`

- **DSRG**: Distributed Systems Research Group
  SOFA model ⇝ previous work = basis for the project
  `http://dsrg.mff.cuni.cz/`

- **LCI**: Laboratorul de Cercetare in Informatica
  Computer Science Research Laboratory ⇝ OCL, MDD, Tools
  `http://lci.cs.ubbcluj.ro/`

- **OBASCO**: OBjects, ASpects and COmponents
  Previous work on Java and Components
  `http://www.emn.fr/x-info/obasco/`

The four teams have complementary knowledge and background on the project domain. The goal is therefore to compare and exchange the point of view, and to integrate the new ideas and techniques in the current proposal.

1.1.3 Initial Plan

The project is established for two years. The initial planning was organised as follow:
First year:

- Determination of the field of application (boundaries of Java concepts and idioms).
- Settings of the major principles to abstract behaviours for software components (into Kmelia, SOFA and STS) from Java code.
- Experimentations on existing code.
- Studying and proposing a pattern for annotating EJB components in order to better support RE (behavior abstraction from code).
- Integration of the verification of guards using OCL (and OCLE).
- Documentation, research report and workshop preparation.

Second year:

- Refinement and classification of the principle and techniques.
- Study of the verification of assertions with OCL.
- Reverse engineering from EJB code to EJB specification realized in JML or OCL.
- Experimentation with larger case studies.
- Documentation, research report and workshop preparation.

Once the context has been introduced, we present now the workshop itself.

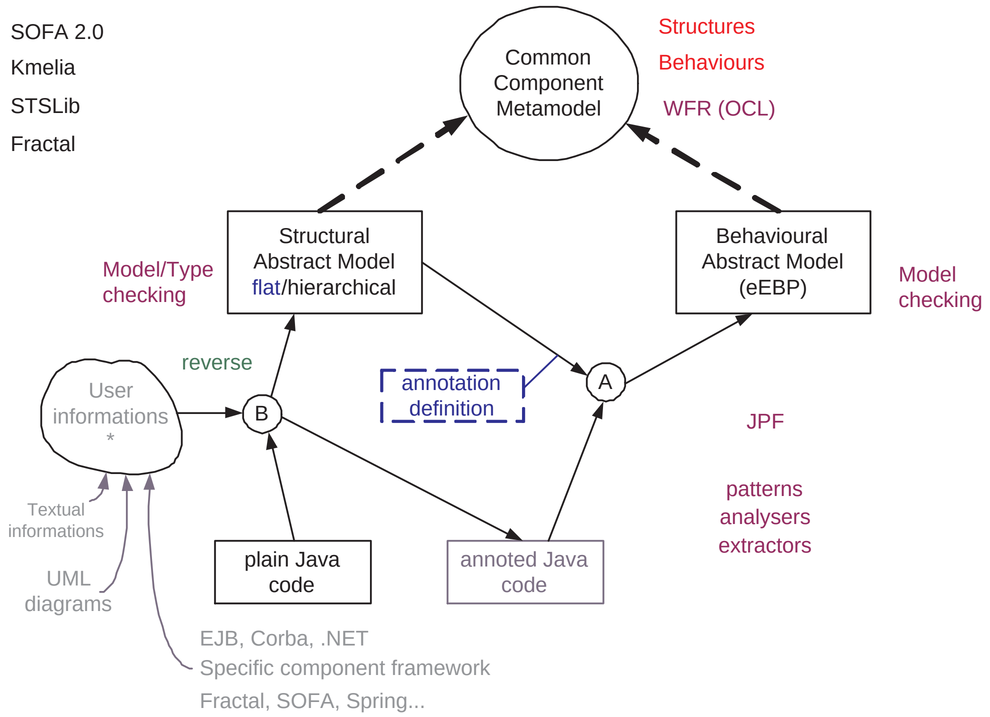


Figure 1.1: Econet Architecture: final version

1.1.4 Current State

The general project organisation has been drawn during the first project workshop in prague in september 2007. Figure 1.1 shows the project architecture.

The executive roadmap for reengineering program is built on a three part architecture:

- Process B: Structural abstraction from Java code.
- Process A: Behavioural abstraction from Java code.
- Metamodel definition and consistency verification.

The objective of the process B is to build a structural component model and a corresponding annotated Java code. These two elements are inputs of the process A. The model is also an instance of the metamodel that will control its consistency. From plain Java code and user interaction, process B should produce an annotated Java code and a corresponding component model (both results must be consistent).

Process A extract a dynamic behaviour specification of the components identified during the process A from the annotated Java code. Therefore, the idea is to make the reverse engineering as general as possible in order to allow extraction of behaviour in any formalism. To be more specific, the formalisms considered are: *Enhanced behaviour protocols* (EBP) developed by DSRG, *eLTS* developed by COLOSS and *STS* developed by OBASCO.

The metamodel part is shared by the two processes and constitues the foundation API (Application Programming Interface) for component model processing. A main issue of a component metamodel is to answer to the problem of handling several component models to get a generic reengineering process. Moreover, in the context of reengineering the metamodel must handle tightened connections to the code that implements component applications. These connection points are represented by annotations in the Java code. In order to provide a convenient component model API, a metamodel specification is necessary to serve as reference guide.

The Prague workshop report [ACPR07] provides detailed informations on these subprojects.

The current state of the project is online the wiki pages (figure 1.2).

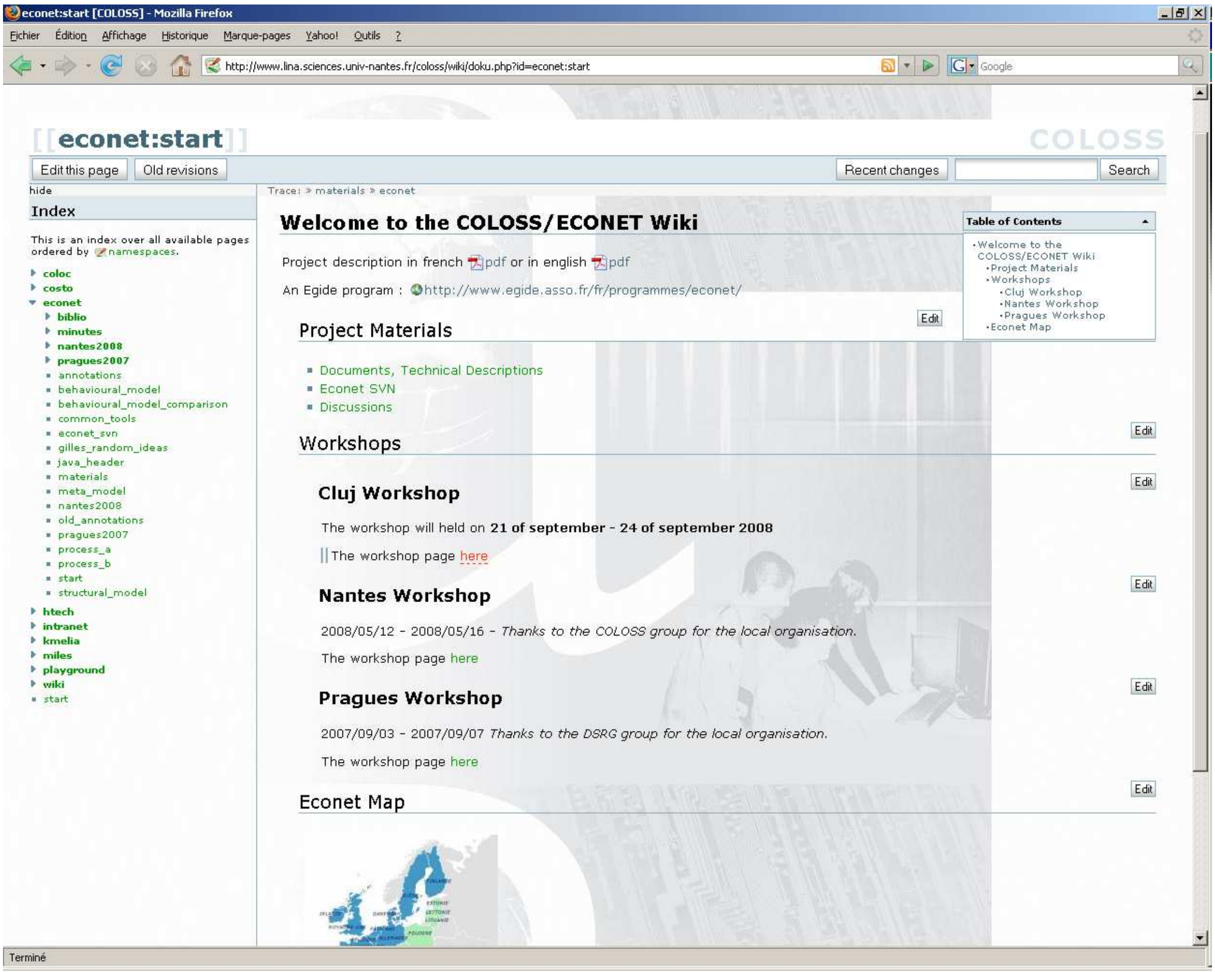


Figure 1.2: Project Wiki

http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:start

Project material and documents are downloadable from the collaborative tools (further information is given in appendix A).

1.2 Report Contents

In the remaining of the report, we provide general informations on the workshop contents in chapter 2. The detailed information of the presentation sessions are described per subproject in chapter 3. Chapter 4 relates the working sessions and results and especially the common component metamodel validation which is the main result of the workshop.

Warning
This report has been mainly written by Pascal from his personal notes and memory of events. There may remain english errors, misunderstanding, transcription errors, and so on. He apologise for these errors.

Chapter 2

The Workshop at the University of Nantes

The workshop is an intermediate milestone for the second year of the project.

2.1 Preparation

The preparation was twofold: material and organisation. The collaborative support is based on a wiki and a SVN repository (see appendix A. In particular there are chapters for each workshop (see figure 1.2).

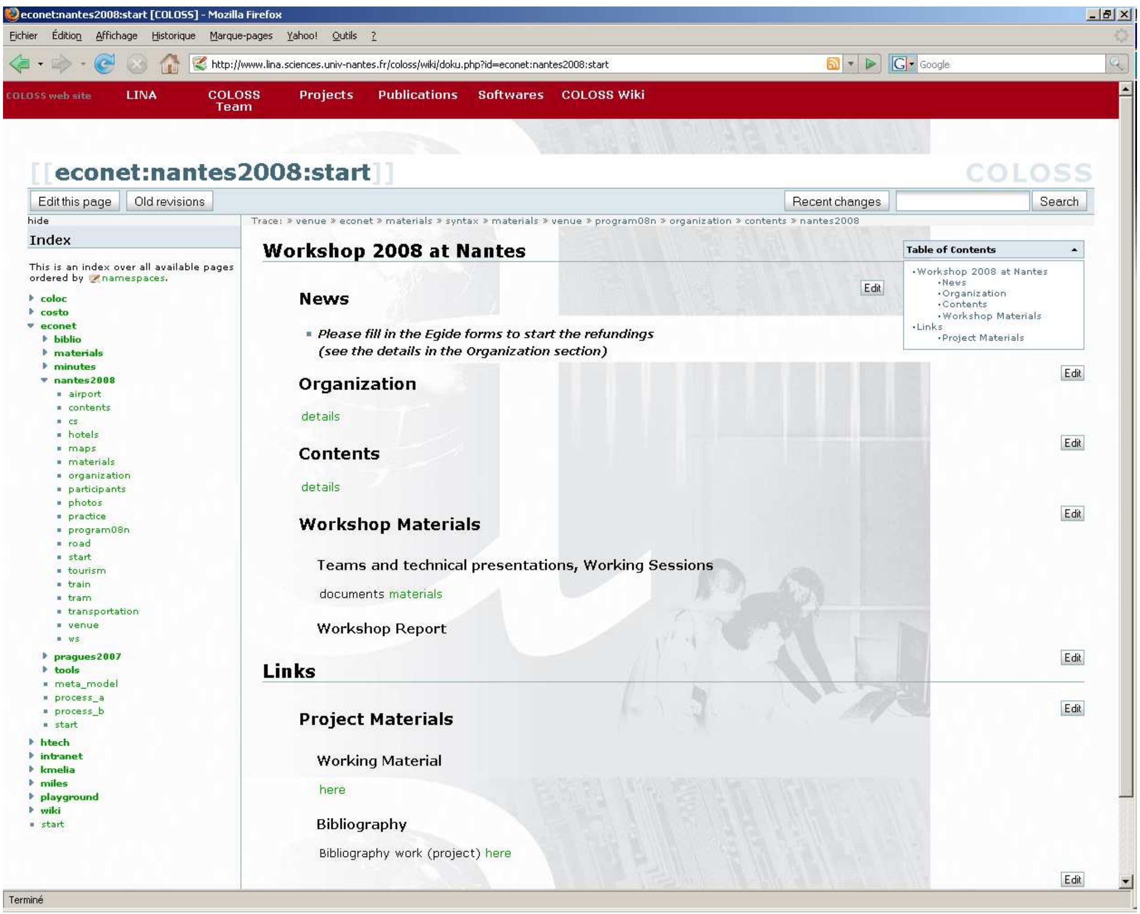


Figure 2.1: Workshop pages on the Wiki

The URL address for the one of Nantes (see figure 2.1) is:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:nantes2008:start

2.1.1 Material

Since the last workshop the contributions mainly focused on the reports (Prague07 Workshop report, Econet first year evaluation) and the metamodel description (Rational Rose metamodels, notes). Minutes have not been summarised on the wiki but the results and documents are put on both the wiki (figure A.3) and the SVN repository (figure A.1).

A special group of pages have been written for the Workshop material (figure 2.2). The URL address is:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:nantes2008:materials

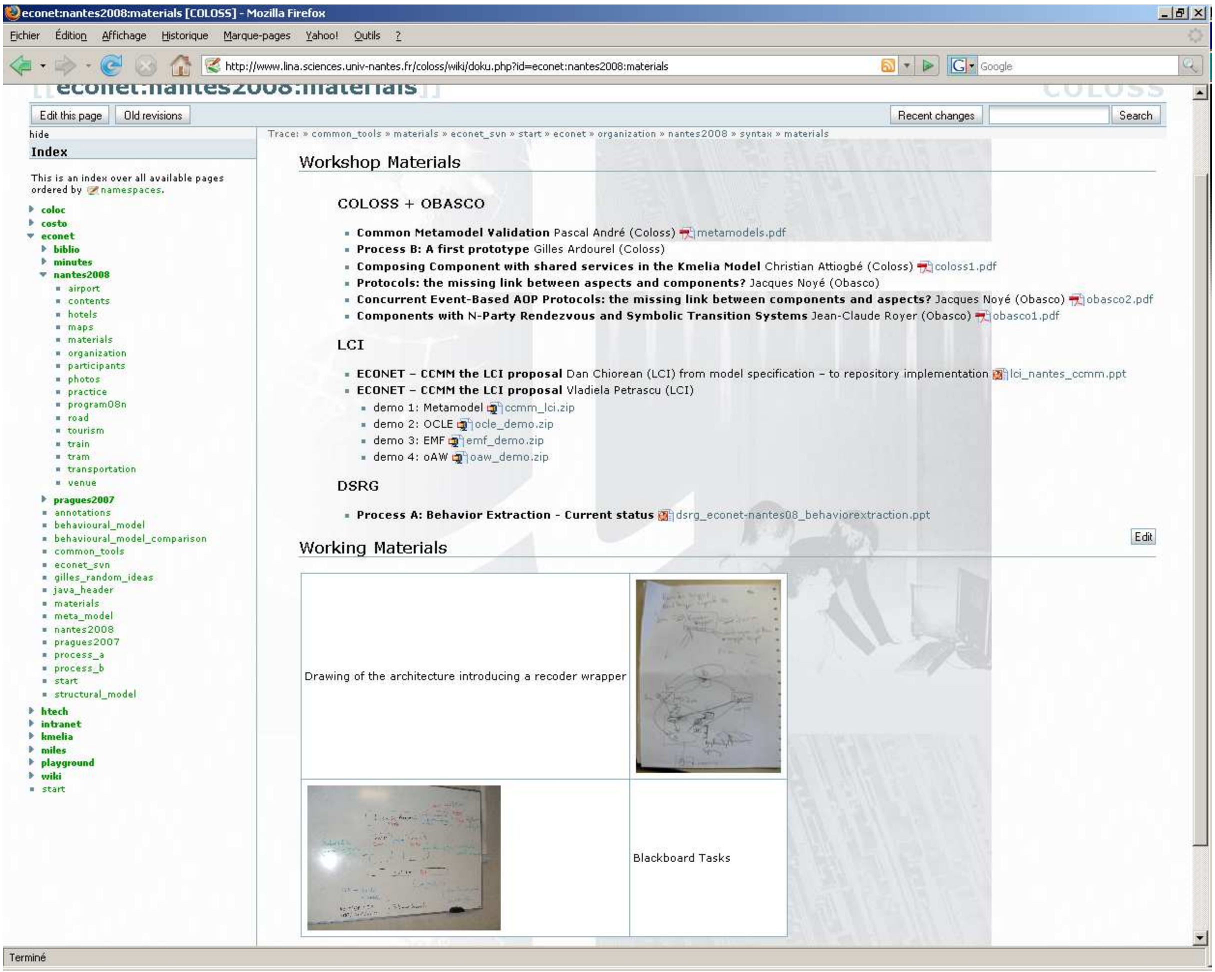


Figure 2.2: Workshop Materials on the Wiki

2.1.2 Organisation

The workshop was initially planned on the end of March. Since we had not the confirmation of the project continuation we should delay to the second week of may after the Egide decision fall and reasonable time to get transportation means.

The local organization committee included Pascal André, Gilles Ardourel, Christian Attiogbé, Isabelle Condette and Anne-Françoise Quin.

Detailed information is given on the wiki site (figure 2.3): venue, program, transportation, city and tourist information, photos, maps and so on.

http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:nantes2008:organization

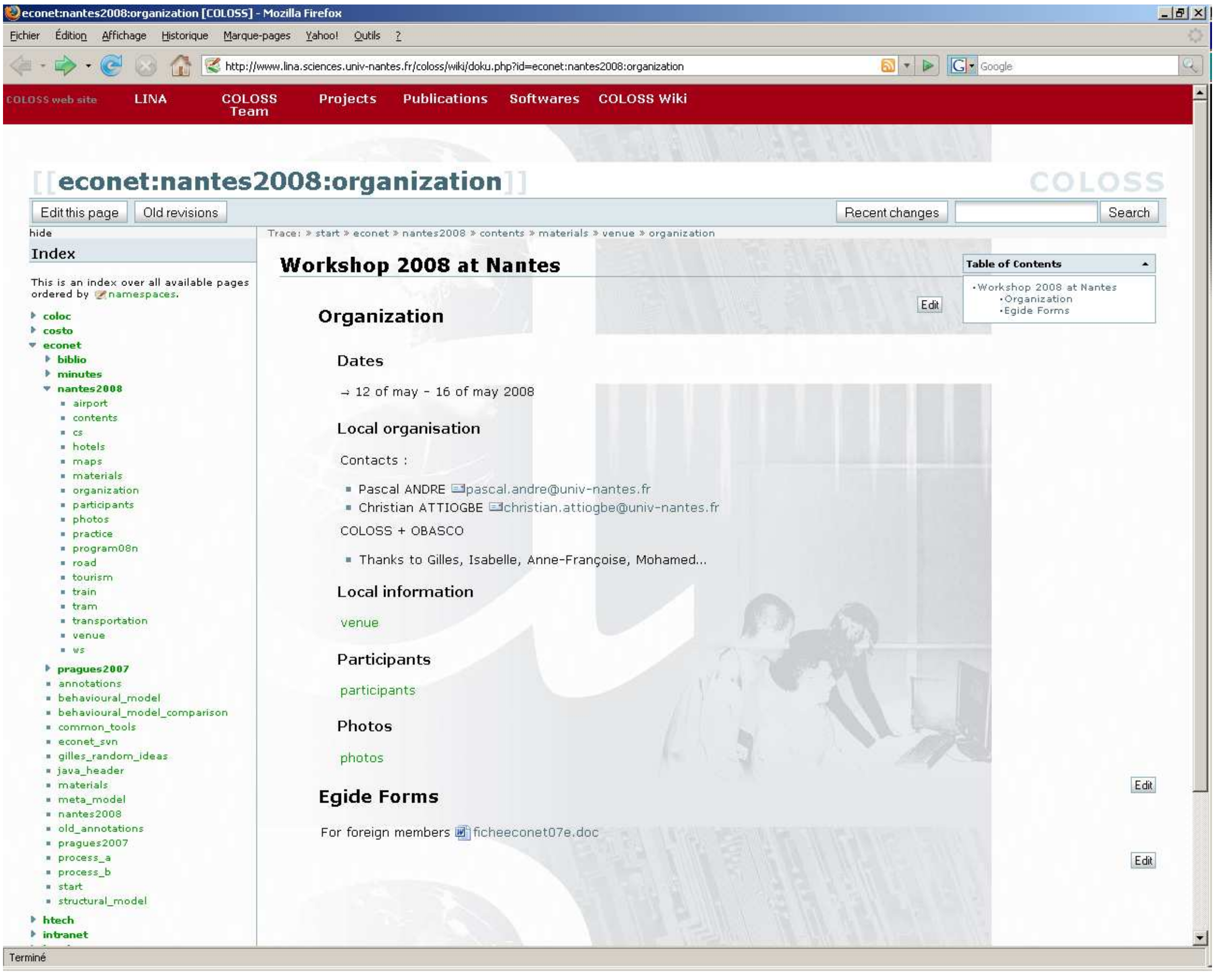


Figure 2.3: Workshop Organisation on the Wiki

2.2 Objectives

The following 'Workshop Objectives and Delivery' statement was a first throw and kept many issues open. We remind here a list of the main tracks we had to follow

1. Present the current situation
   - for each subproject
   - products and problems
   - future work
2. Tools normalisation
   - compare tools and techniques of each subproject
   - final decisions on the tools panel
   - perspectives
3. Study the interface between the parts
   - format, filters, API...
   - languages
4. Get a first prototype

- source and documentations for each subproject
- final decisions on the metamodel part
- extract the main open issues
- applications on CoCoME

5. Draw the roadmap to the end of the year

- development
- documentation
- workshop preparation
- publication (reports, papers)

2.3 Participants

The detailed list is arranged according to the alphabetical order of first names.

- *Christian ATTIOGBE* - COLOSS
- *Dan CHIOREAN* - LCI
- *Dragos PETRASCU* - LCI
- *František PLÁŠIL* - DSRG
- *Gilles ARDOUREL* - COLOSS
- *Jacques NOYE* - OBASCO
- *Jean-Claude ROYER* - OBASCO
- *Mohammed MESSABIHI* - COLOSS
- *Ondřej ŠERÝ* - DSRG
- *Pascal ANDRE* - COLOSS
- *Petr HNĚTYNKA* - DSRG
- *Tomáš POCH* - DSRG
- *Vladiela PETRASCU* - LCI

2.4 Program and Schedule

We present here an overview of the workshop program. It was organised in two parts

- Day 1 and 2 are dedicated to workshop presentations. The durations and schedules leave time for numerous discussions...
  - Presentation of the subprojects (recent work, tools, ...)
  - Technical presentations and demonstrations
- Day 3 is dedicated to the coordination issues for the project, the Cluj workshop organisation and social events.
- Day 4 and 5 are dedicated to the project work (metamodel, interfaces, tools, sharing experience, practical organisation and responsabilities)

Actually the schedule evolved due to some people own constraints (flighs...).

The detailed program is given on the wiki at:
`http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:nantes2008:program08n`

2.5 The Workshop Sessions

This section is a quick overview of the executed program of the workshop. The detail features will be presented in the following chapters. The workshop material is available on the wiki at:
`http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:nantes2008:materials`

We first begin by the presentation sessions where the participants presented the their technical contributions (chapter 3). Then we summarise in chapter 4 the contributions of the working sessions where the participants discussed on the project (issues, structure, tasks, technical aspects, tools...).

2.5.1 The Presentation Sessions

Monday, May 12, 2008

Time	Title	Speaker
14:00	Welcome	Christian Attiogbé
|	Workshop Introduction	Pascal André
|	Local Organisation	COLOSS
14:30	Technical presentations about the Metamodel subproject	
|	CCMM the LCI proposal: from model specification - to repository implementation	Dan Chiorean
|	demo1: Metamodel	Vladiela Petrascu
17:30	demo2: OCLE	Vladiela Petrascu

Welcome Christian welcomed the participants in the name of the Laboratory and the COLOSS team.

Workshop Introduction Pascal introduced the workshop recalling the ECONET project context for the "new" participants. He quickly summarised what happended during the first year.

Summary
Events

- March: starting the project
- September: workshop at Prague (initially planned for june)
- October: workshop report, project evaluation
- November: First Common Component Metamodel published

Results

- Workshop Report
- Project Continuation
- First Draft Common Component Meta-Model

Quick Analysis

\+ Workshop organisation and result

\+ Complementary background of the teams

\+ Methods and collaborative tools (Wiki, SVN, email)

\- Time Allocation (late start, deadlines, asynchronous working period and exchanges)

\- Too few (despite fruitful) technical echanges (bibliography, metamodel, tools)

\- Some Misunderstandings (due to informal definitions or varying contexts?)

First Year Results

Advance in

- Clear project definition (workshop results)
  - Convergence on the objectives
  - Convergence on the means
  - Definition of the tasks
- Project Collaborative tools (Wiki, SVN)
- Toward a common component meta-model

Standby/delay for

- Collaborative field exploration: Annoted bibliography and Synthesis (components, RE, code engineering, tools)
- A validation of the common component meta-model
- Delayed or lost activities
  - Studying and proposing a pattern for annotating EJB components in order to better support RE (behavior abstraction from code).
  - Integration of the verification of guards using OCL (and OCLE).

First Year Workshop Results

Convergence on the objectives (summary)

- Clear agreement on the "abstract" context
  Abstract component models
  + Java Code
  + Reverse = from code to abstract models
- Some vision of the "concrete" context
  - Java code nature
    Bytecode or Plain source or Annoted Source
  - Java code structure
    plain Java + informations
  - reengineering issues
    abstraction rather than full reverse engineering
    compare code and specifications (conformance)
- Benchmark = CoCoME
- Two other tracks: cross LTS extensions, WFR definitions

Convergence on the means (summary)

- Project Architecture with Three parts
  1. Component Metamodel cross LTS extensions, WFR
  2. Structure Abstraction user interacted tool
  3. Behavior Abstraction A-interface definition, annotations generation
- Problem Domain Restriction
  - metamodel $\Longrightarrow$ components and behaviours

  - A $\Longrightarrow$ no connections, no composition, no statement abstraction
  - B $\Longrightarrow$ no composition, no statement abstraction, user-interactions

- Benchmark = CoCoME

Definition of the tasks (summary)

- Prototype on the project architecture
  - Metamodel
  - Process A
  - Process B
- Cross Contributions a subset of
  - Common Metamodel Definition
  - Annotation language definition (input of process A)
  - Tools Prototypes for Metamodel verification, Process A, Process B
- Synchronisation points =
  A-interface, Metamodel def, B-Information def
- Planning deadlines
  - Workshop Nantes (begin of March 2008)
  - Workshop Cluj (end of august 2008)
- Publications

Workshop Program The contents includes

- Participants
- Objectives (open issue !) $\rightsquigarrow$ Detail Design of the Project Architecture + Technical Issues
  - Metamodel: contents and design
    concepts, relations, mains issues, approaches, plateforms and tools
  - Processes: interfaces and design
    structure, libraries, techniques, tools
  - Integration and examples
    CoCoME
- Delivery $\rightsquigarrow$ workshop report + roadmap until next workshop
  - Prototype
  - Refine with concrete models
  - Documentation, research report and workshop preparation.
  - Perspectives and Publication
- Detailed Program and Schedule

Presentation Session We started by LCI because the Metamodel supports the interface between subprojects A and B. Dan recalled the LCi tasks, mainly

- CCMM definition: Metamodel specification, constraints specification, metamodel testing, repository code generation
- Studying and testing different tools supporting the above mentioned activities (OCLE, EMF, oAW).

Then he argued the LCO position and proposals.

There after Vladiela presented a part of the demonstration. She started with a metamodel proposal and discussion and continued with the OCLE implementation.

The `Metamodel` subproject if further developped in sectionmetamodel of chapter 3.

Tuesday, May 13, 2008

The initial schedule was modified in order to continue the LCI demonstrations.

Time	Title	Speaker
09:00	Technical presentations about the Metamodel subproject (contd.)	
|	demo2: OCLE (contd.)	Vladiela Petrascu
|	demo3: EMF	Vladiela Petrascu
|	demo4: oAW	Vladiela Petrascu
12:15	Common Metamodel Validation	Pascal André
13:30	Technical presentations about the Process B (structure extraction) subproject	
|	Process B: A first prototype	Gilles Ardourel
|	Composing Component with shared services in the Kmelia Model	Christian Attiogbé
|	Concurrent Event-Based AOP Protocols:	Jacques Noyé
17:30	the missing link between components and aspects?	
delayed	Components with N-Party Rendezvous and Symbolic Transition Systems	Jean-Claude Royer

At the beginning Vladiela continued with the second part of the demonstration using the OCLE, EMF and oAW implementations. The `Metamodel` subproject if further developped in sectionmetamodel of chapter 3.

The Common Metamodel Validation is part of the working sessions closely related to the metamodel subproject (see section 2.5.2).

Technical presentations about the Process B subproject started with a short presentation of the experimentations leaded in the COLOSS team. The project was realised by a group of students and included bith the annotation processing and the metamodel managment (for a limit subset of the metamodel). The idea was to install a bootstrap for the Process B machinary which is an iterative process. The goal is to link Java programs (with or without annotations) and component models (which is assumed to be an abstraction of the Java program). The prototype read and write annotations and instantiate models from a metamodel implementation in ATL (see section B.1.3). The `Process B` subproject if further developped in section 3.2 of chapter 3.

The other presentations are related work. The last presentation occurred on thursday due to timing constraints. Here is a short summary of the presentations.

Composing Component with shared services in the Kmelia Model The Kmelia component model [AAA06b] was introduced as an abstract formal component model dedicated to the specification and development of correct components. The model is equipped with a language which is evolving together with the expressive power of the model. In [AAA06b] we have distinguished two semantics for the link between component services. Only one, *monadic semantics*, was treated in this previous article. The second one, *polyadic semantics*, was not treated. The hypothesis for the *monadic semantics* is: only one provided service may be associated to a required service; a component is both a component type and the unique instance of it; a required service may be linked to at most one provided service; only one instantiation of a service exists at any time.

In the current article we consider the *polyadic semantics*: a provided service may be linked with various required services (allowing broadcast communications); as an example, a chat system provides an interaction service for multiple clients. In the same way a required service may be linked to various provided services. We present the new features of our Kmelia model, the language aspects that support these features and how these improvements are integrated with the previous works on Kmelia.

The modelling of various real life systems such as auction systems, chat systems, distributed brokers, etc requires the use of several components of the same type or several services with identical functionalities but coming from different components. This leads to the need of interaction means to support the assembly and the composition w.r.t to the multiplicity of services that may be connected. The current Kmelia model and language provide a one to one service/component interaction even if several components participate in the assembly. This does not cover the kind of systems listed above.

The contribution of this article is the improvement of the expressivity of the Kmelia component model with shared services, multipart interaction based on synchronous n-ary communications. We extend Kmelia to support multiple connections between services. Also, we explicitly distinguish between *component types* and *components (as elements)*, hence we may use several components of the same type in an assembly. Accordingly, the interaction between Kmelia services is updated.

Concurrent Event-Based AOP Protocols Concurrent Event-based AOP (CEAOP) [DLBNS06] is based on the seminal work by Douence, Fradet, and Südholt [DFS02] on Event-based AOP. Event-based AOP extends "standard AOP" (à la AspectJ) with *stateful* or *event-based* aspects, which, instead of associating additional behaviour (an *advice*) to an atomic execution point (a *join point*), associate behaviour to a sequence of execution points, seen as *events* monitored by the aspect. Whereas the initial semantics of EAOP was sequential, CEAOP defines a concurrent semantics of stateful aspects. It does so by considering abstract aspects defined by regular sequences of events to which advices can be associated. These aspects are abstract as events are plain labels and advices are simply sequences of actions, including the predefined actions `skip` and `proceed`, to specify whether an event should be skipped or not. The semantics of such an aspect is then defined by two transformations, an aspect transformation turning the aspect into a Finite State Process (FSP) [MK06], and a base transformation "instrumenting" the FSP representing the *base* program with which the aspect should be composed, such that the parallel composition of both the aspect FSP and the instrumented FSP behaves as expected.

For instance, if we compose the base application `Server` and the aspect `Consistency` (where the operator `>` and the keyword `skip` are constructs specific to CEAOP), we expect the event `update` not to happen during server sessions.

```
Server =                    Session =
  ( login -> Session          ( checkout -> Server
  | update -> Server          | update -> Session
  ),                          | browse -> Session
                              ).
 Consistency =               Session =
   ( login -> Session          ( update > skip  -> log -> Session
   ),                          | checkout -> Consistency
                               ).
||S = (Server || Consistency).
```

The instrumentation scheme makes it possible to control synchronization between the aspect and the base program whereas additional composition operators (which can also be translated into plain FSP) make it possible to deal with the synchronization of several aspects.

We have used this model as the execution model of a concrete extension of Java, Baton [NN07a], which combines concurrent and aspect-oriented programming. In Baton, base programs are composition of active objects. These objects are instrumented with *pointcuts* describing the events of interest whereas the aspect transformation of CEAOP is used to synthesized aspects described in a syntax combining FSP and Java traits. As part of instrumenting the base program and synthesizing the aspects, the compiler also generates calls to a global monitor, which is responsible for performing synchronization as specified by the model.

This has been extended in order to support a simple component model [NN07b], whereby the base is structured as components with static interfaces describing the *required* and *provided* services, as well as the *published* events (this is related to the notion of *open modules* [Ald05]) and dynamic interfaces describing the corresponding behaviour. On the aspect side, the static *aspect interfaces* describe the events of interest, which may be *skippable*, as well as *required* and *provided* services. In the same way as a composition of aspects and FSPs can be turned into a mere composition of FSPs, a composition of aspects and components can be turned into a composition of mere components.

Finally, we have considered, on top of CEAOP, abstractions that facilitate the modelling of context-aware applications [NN07b].

We think that this work give an interesting perspective on the links between processes, components, and aspects and paves the way to concrete languages that support these notions, including support at the architectural level, in a more integrated way.

Components with N-Party Rendezvous Component software engineering has been used to improve system modularisation and artefact reuse. However, most of the current proposals are restricted to binary communications. They are often suitable, but there exist some applications domains, like controller synthesis, where they are not sufficient enough. We argue that more complex interactions are needed, and we designed a component language with explicit symbolic protocols and N-party rendezvous. In this context, we introduce sophisticated bindings to control component behaviour in a black box way, and we address the computation of a global protocol associated to component assemblies. We define an extension of the synchronous product adapted to our protocols which keeps inside states and transitions, the structure of the composite and enables four kinds of bindings In a second step, we formalise our model and define behavioural compatibility. We further introduce a new property called event strictness, and we prove some preliminary results about the checking of these properties.

Wednesday, May 14, 2008

The initial schedule was modified in order to discuss about the project itself and the workshop of Cluj.

Time	Title	Speaker
09:00	Technical presentations about the Process A (behaviour extraction) subproject	
|	Econet process A: Reengineering behaviour specification	Tomas Poch
11:30	ECONET Project discussions	
13:00	Social	
18:30	Events	

Tomas presented the work leaded by DSRG about the Process A (behaviour extraction) subproject. The goal is to extract the behaviour specification of a primitive component implemented by a set of Java classes. Only **primitive components** behaviour will be abstracted. Composite components are outside the scope of the subproject. Additional information is still needed which are provided by the process B in form of annotations (e.g. which classes implement the component, which are the provisions and requirements, which are the data abstraction...). The strategy is to stick with Java as long as possible, make transformations over the Java AST and perform the transformation to the target behavioural model is the last step. The transformation chain should be configurable. An experimentation is shown on a toy example.
The `Process A` subproject if further developped in section 3.3 of chapter 3.

We also discussed about the organisation of the next workshop in three months (budget, dates, people). A two-week period is fixed that takes into account various unavailable constraints. It has been precised after the workshop. It will held on **21 of september - 24 of september 2008**.

2.5.2 The Working Sessions

This section summarises the discussions and contributions of the working sessions.

Working Session Roadmap

The initial Working Session program was proposed as follow:

1. Common Component Metamodel
   - Materials
   - Discussions and Decisions
     - Concepts and relations
     - Architectural choices (core, concepts, specialisations, annotations, management, instances)
     - Tools
     - API and tools
   - Others: Roundtrip
   - Specification document

   Goal of days 2,4 = Clear agreement on the "common" metamodel

2. Tools and techniques
   - Discussions on Tools and techniques
     - Experience feedback
     - Tools coordination
   - Model Management

  - EMF, OCLE, oAW...
  - Rule based systems, checking
  - Compatibility
  - ...
- Re-engineering techniques
  - Java Compilers and Analysers
  - Patterns, rule based systems
  - Used notations and Intermediate layers (models)
  - ...

(optimistic) Goal of day 4 = organize the implementation means

3. Definition of the tasks

- What to do ? on the project architecture
  - Metamodel
  - Process A
  - Process B
- Contributions ? a subset of
  - Common Metamodel definition
  - Annotation language definition (input of process A)
  - Tools Prototypes for Metamodel verification, Process A, Process B
- Synchronisation points =
  A-interface, Metamodel def, B-Information def
- Planning deadlines
  - Workshop Nantes report
  - Workshop Cluj (end of august 2008)
  - Project Evaluation (november 2007)
  - Publications

(optimistic) Goal of day 5 = each participant has a somewhat clear idea of what he will do

4. Production

- Workshop Report
  - Collect paper and slides Please send them to me
  - Summary of the discussions

\+ Bibliographical Notes

$\Longrightarrow$ project plan for year 2 and *Evaluation*

- Fix the participants objectives
- Documentation, research reports
- Intermediate results $\Longrightarrow$ Thirsd Workshop
- Publications (?)

see also the initial 'Second year objectives'

Thursday, May 15, 2008

The initial schedule was modified in order to include the technical presentation of Jean-Claude and also a discussion on tasks, responsabilities and delivery schedule.

Time	Title	Speaker
09:00	Technical presentations about the Process B (Structure extraction) subproject	
|	Components with N-Party Rendezvous and Symbolic Transition Systems	Jean-Claude Royer
|	ECONET Project discussions	
12:00	Task, schedules	
14:00	Working session II	
17:00	Metamodel, annotations	

Tasks and Scheduled The discussions started with some interrogations of Dan about the metamodel specification and some doubts LCI had about CCMM v1.0 (big model, not enough constraints and informations...). LCI also worried about including the behavioural aspects and annotations management in the metamodel. The answer is twofold :

- Distinction between a specification metamodel and an implementation metamodel which is a subset of the primer metamodel. Behaviours (too specific concepts), implementation language (java concepts), strong model management, additional concepts (specific to one or another concrete component metamodel) are not in the scope of the implementation.

- Validation of the metamodel (selection and definition of concepts and their relations, constraints and examples) is one goal of this workshop.

We also discussed about modelling methodology (to represent variation on concepts in a metamodel *e.g.* using gen/spec relations, attributes, associations) and **model transformations** using ATL, oAW or EMF - for example to get a CCMM instance from *Extended Behavior Protocols (EBP)* or *(Extented) Labelled Transitions Systems (LTS)*.

Thereafter we discussed about tasks, responsabilities and deadlines for the metamodel subproject.

- Tasks
  - CCMM specification + special requirements (input)
  - Metamodel verification
  - API generation and testing
- Deadlines
  - specification: 7 of june 2008
  - version 1 (EMF) : 22 of june 2008
  - version 2 (oAW) : end of june 2008

Discussions on process A and B, prototypes, case study, documentations and publications are delayed. We also discussed again on the dates for the Cluj Workshop.

Working session II One group worked on the metamodel validation (see section 4.2).

The other one on annotation refinement (see section 4.3.1).

Friday, May 16, 2008

The initial schedule was modified in order to discuss about the project itself and the workshop of Cluj.

Time	Title	Speaker
09:00	ECONET Project discussions	
|	Task, schedules	
|	Working session III	
12:00	Metamodel, interfaces, architecture, recoder wrapper, benchmark	

At first we discussed about tasks, responsabilities and deadlines for the processes subproject. Figure 4.4 is a snapshot of the discussions.

Working session III One group worked on the case study selection (see section 4.4).

One group worked on the metamodel validation (see section 4.2).

The other one on annotation refinement and interfaces (see section 4.3.1).

Chapter 3

Project and Technical Presentation Sessions

The contents of this chapter presents a detailed snapshot of the current state of the three subprojects, defined in the workshop of Prague.

3.1 Metamodel Abstraction Subproject

Writer: Vladiela Petrascu

`+++ TODO: This is currently a draft version +++`

3.1.1 ECONET CCMM - From Model Specification to Repository Implementation

`+++ to write +++`

3.1.2 LCI Tool Demos Summarized

Objectives and Goals

The LCI tool demos aimed at analysing and comparing the facilities provided by different CASE tools for metamodels' representation (including Well Formedness Rules - WFRs, and observers - query operations) and generation of the associated repository code. We have considered the following tools: Object Constraint Language Environment (OCLE) [ocl], Eclipse Modeling Framework (EMF) [emf], and openArchitectureWare (oAW) [oaw], and the following criteria for differentiating among them:

(1) support offered for integrating metamodel WFRs and observers, counting the ease of writing and compiling constraints (code completion was taken into account);

(2) ease of evaluating these constraints on concrete models (snapshots) and assistance provided by the tool in locating a possible validation error and correcting it in real time;

(3) completeness of the generated repository code, including the code corresponding to WFRs and observers;

(4) generated code's simplicity and intelligibility (essential in case additions and/or changes are required on it), as well as the amount of dependencies required when running it outside of its generator environment.

The presentation's ultimate goal was for the partners to choose one or several of these tools to be used within the current ECONET project.

The LCI proposal for a starting version of the Common Component MetaModel (CCMM) was the metamodel used throughout the OCLE, EMF, and oAW tool demos. Several WFRs were specified on it, including name uniqueness constraints inside namespaces (name uniqueness of `Type`s, `InterfaceType`s and `ComponentType`s inside a `Repository`; name uniqueness of a `ComponentType`'s `Interfaces`; name uniqueness of an `Architecture`'s `Components`; name uniqueness of an `Operation`'s `Parameters`), valid component bindings constraints (compatible `InterfaceTypes` of `Interfaces` linked through a `Binding`; `Assembly` / `DelegationBinding` semantics encapsulating constraints), or non-cyclic definition of composed component instances. An operation that selects all `ComponentTypes` that provide a certain `InterfaceType`, from within a `Repository`, was taken as an observer example.

The three tool demos are summarized below, following the above mentioned four criteria.

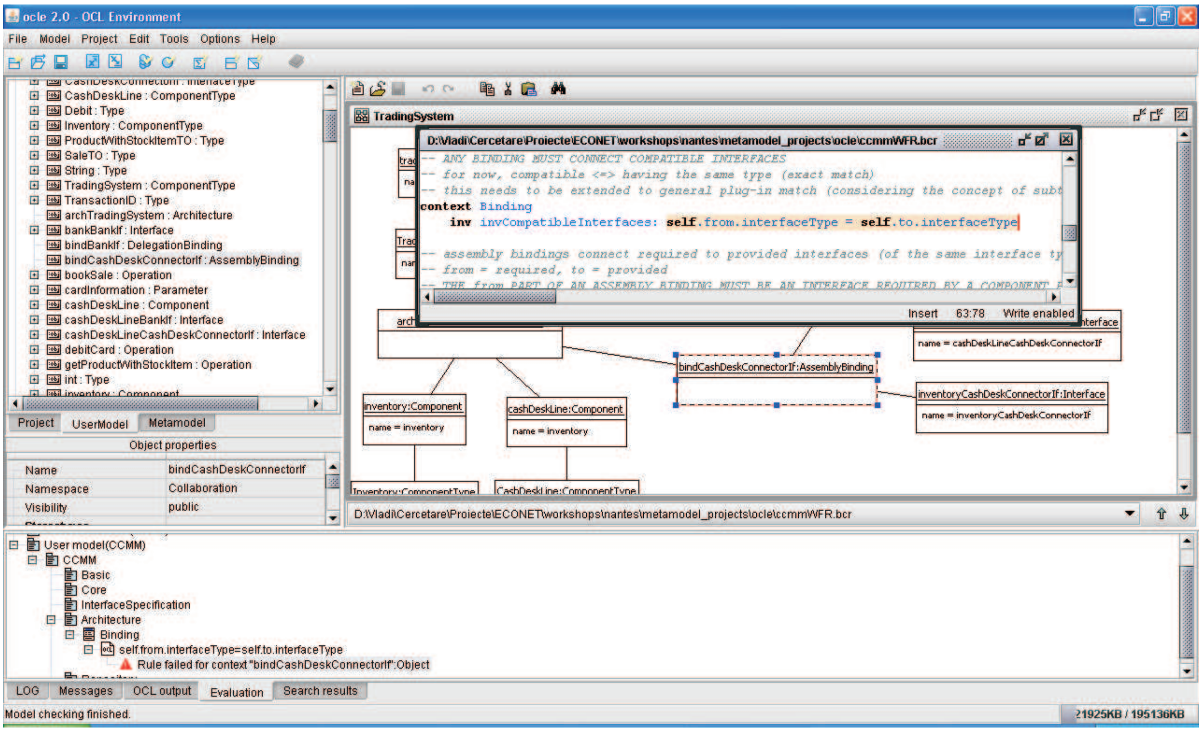


Figure 3.1: Model checking in OCLE

OCLE Demo summarized

(1) In OCLE, the CCMM metamodel is represented as a UML 1.5 model. Both WFRs and observers are included inside .bcr (business constraint rules) files; WFRs are specified as *inv* («invariant» stereotyped) constraints, while metamodel level queries are represented using the OCL *def* mechanism («definition» stereotyped constraints). OCLE .bcr files can be compiled and, if the case, meaningful error messages are displayed inside the Messages tab, including the exact place the error occurred in. Code completion facilities are not yet provided by the tool.

(2) In OCLE, constraints' evaluation is performed on shaphots. These are object diagrams containing (meta)class instances (having slots corresponding to attributes' values) and links among them (instances of associations specified in the (meta)model). The evaluation process can encompass either all specified constraints or a particular one, chosen by the user. Single constraint evaluation involves two steps: (a) selection of a contextual instance among the existing snapshot objects, and (b) evaluation of the different constraint constituents (in particular, the whole contraint), using the Evaluate Selection option. Evaluation results are displayed inside the OCL Output Tab. Observers can be evaluated by following a similar scheme. Evaluation of all specified constraints is triggered by a Check Model menu option. All errors are reported inside the Evaluation tab in a tree-like manner: eack broken constraint is denoted by a node having as a direct ancestor its context (meta)class and as direct descendants rule failure messages pointing at the "responsible" instances. Selecting such a message makes the corresponding object to be automatically set as the constraint's contextual instance (simultaneously with selecting it in the browser and object diagram, respectively), therefore

single constraint evaluation can be done, which significantly helps in identifying the cause of the error. A snapshot of the model checking activity in OCLE is illustrated in Figure 3.1.2.

(3) OCLE code generator uses the Apache Velocity template engine. For each metamodel class, a corresponding Java repository class is created, containing its specified attributes and references, a default constructor, and get/set or get/add/remove methods (depending on the multiplicity) for references' management. In case WFRs were specified in the context of a (meta)class, then its generated code includes a `ConstraintChecker` class with validation methods corresponding to each WFR (the method's code represents the Java translation of the WFR's OCL constraint). Constraint breaking is indicated by a message displayed on the standard output, pointing out the violated invariant's name, as well as the responsible object. Calling the `ConstraintChecker` methods is left on behalf of the user.

(4) The generated CCMM repository code is simple, easy to understand and manage. Using it within a Java project only requires importing the small OCLFramework library.

EMF Demo summarized

(1) In EMF, a metamodel (CCMM, in particular) is represented as an Ecore model. WFRs are specified in OCL (with minor "dialect" differences compared to OCLE, e.g. `oclIsUndefined()` vs. `isUndefined()`) and attached to their context metaclasses in the form of annotations [Dam07]. Metamodel level observers are given as metaclass operations, having their body defined by an OCL expression. The expression is attached to the observer operation in the form of an annotation, having as child a Details Entry of the form (`body`, <bodyOclExpression>) - see Figure 3.1.2. Therefore, EMF constraints and observers directly "pollute" the metamodel as annotations, unlike in OCLE or oAW, where they are specified in separate files. Compilation facilities are not provided at this level. In order to ensure a correct syntax of WFRs and observers, the corresponding OCL expressions should be copy-pasted and evaluated inside the OCL Interpreter tool. The interpreter compiles the OCL before evaluating it, signaling any syntax errors. Code completion facilities are provided. Still, we find this compilation alternative somehow cumbersome.

(2) EMF model checking can be done interactively, by choosing a Validate option from a popup menu on the root element of a model. The model can be constructed using the EMF tree-like editor. Validation results are displayed inside a message box. If validation problems have been identified, then their details may be consulted, each detail line indicating both the violated constraint's name and the model element responsible for breaking it. Theoretically, selecting such a details line should automatically point to the responsible object on the tree, but unfortunately this only works correctly for the first line. We signal this as a bug. Apart from checking the entire model by validating its root, it is also possible to individually check any of its branches (children), in a similar manner. If the validation fails because a constraint is broken by a certain model object, discovering the error's cause is possible through partial evaluations. This resumes to copying different parts of the OCL expression into the OCL Interpreter and evaluating them on the selected object, which is assumed to be the contextual instance. Again, this is not as straightforward as in OCLE, since it involves manually going back to the constraint definition inside the metamodel file and copy-pasting different parts of it inside the interpreter. Thus, the checking facilities implemented in OCLE are indeed quite helpful and time-saving.

(3) EMF code generation uses JET (Java Emitter Templates), the template language having a JSP-like syntax. The code generator uses as input a .genmodel file, which decorates the initial .ecore file containing the metamodel with additional generation related information. For each of the metamodel packages, three corresponding code packages are generated: an interface package, an implementation package and an util one.

For each metaclass, one interface and one implementation java files are generated, inside the interface and implementation packages corresponding to the metamodel package to which the metaclass pertains. The interface contains get/set methods for attributes and multiplicity-one references, and only get methods for multiplicity-many references (returning ELists). Metaclass operations' signature is also included into the generated interface file. Within implementation files, observers' notification is handled appropriately. Moreover, for each metamodel package, corresponding factory (that allows the instantiation of model objects) and package (that allows metadata management) interface and implementation files are created.

The package validator class (from within the generated util package) contains validate methods for all repository classes contained in that package. For eack specified invariant, a corresponding validate method is created. By default (using only the default code generation templates), its body must be filled in by the programmer (only the body skeleton in generated, the code for evaluating the constraint is missing). Generating code for evaluating invariants, observers and derived attributes and references requires using dynamic templates and modifying some .genmodel properties (see the approach proposed in [Dam07]). OCL expressions are not translated directly to the java language, as in OCLE. Instead, their evaluation is delegated to MDT OCL.

Apart from the metamodel repository code, a test project and a textual model editor project can also be generated..

(4) The generated repository code is quite complex, including rich functionality (e.g. notification management, metadata management, factories, several List implementations taylored to specific needs, etc.). However, using it within a new Java project involves several dependencies.

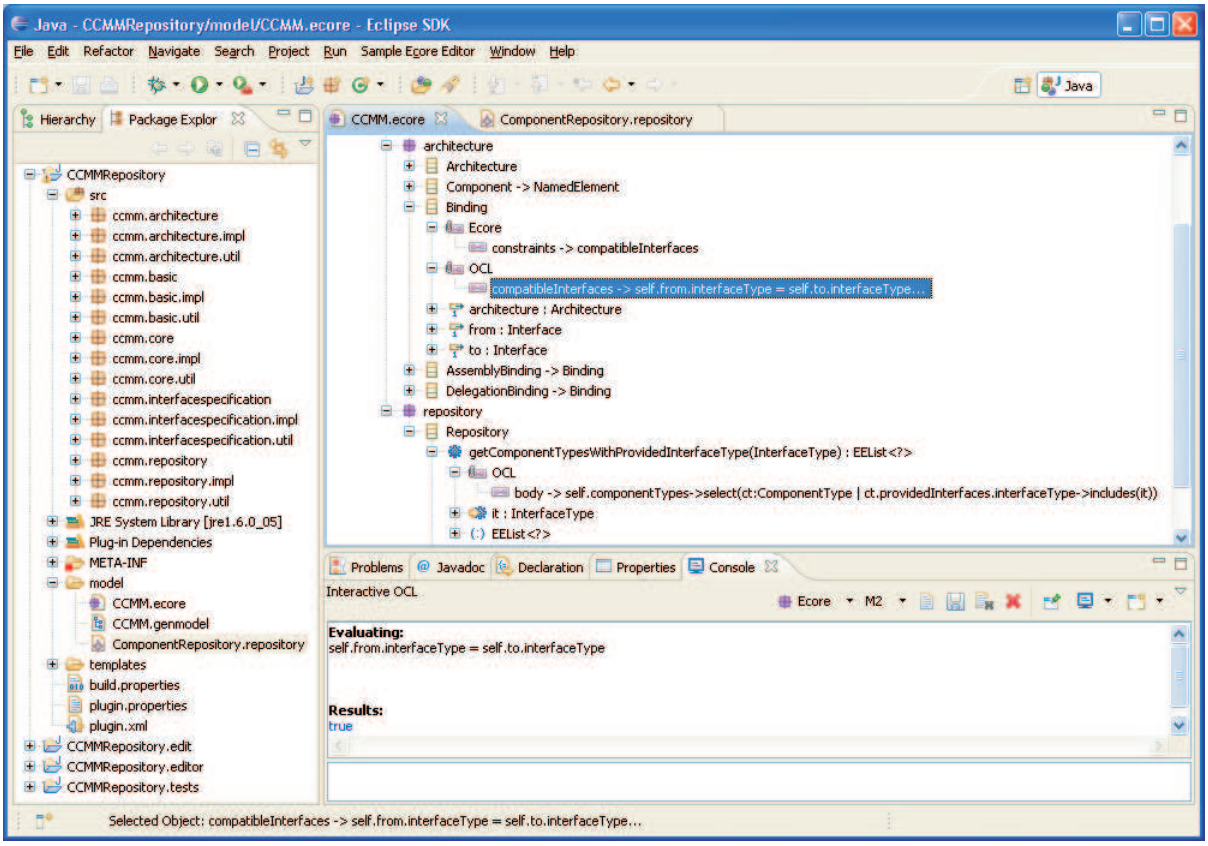


Figure 3.2: An Ecore metamodel including WFRs and observers

oAW Demo summarized

(1) Since oAW 4 supports EMF based metamodels (among other types of metamodels), this tool demo has used the same metamodel representation as the previous one did. In oAW, metamodel level invariants are isolated in .chk files and are defined using the declarative constraint language Check [VKEH06]. Check is an OCL-like language, thus it has an OCL similar syntax, to which it adds the possibility of defining custom error or warning messages to be displayed whenever a constraint is violated. With the intention of keeping metamodels as simple and clear as possible, in oAW all additional properties are defined externally in .ext files, using the oAW Xtend language. This has been also the case with our CCMM observers. In

order to be able to refer to the metamodel classes within the expressions contained in Check and Xtend files, a line importing the metamodel should be included at the beginning of these files. This makes the text editors metamodel-aware. The editors provide syntax coloring and code completion facilities to the user. Compilation of constraint and extension files is automatically done at the moment they are saved, and appropriate error messages are displayed, if the case.

(2) In oAW, all model operations are coordinated by means of a workflow. As shown in Figure 3.1.2, such a workflow consists of an ordered collection of workflow components, each component executing a well defined model related task. There are some standard workflow components offering functionalities such as: reading (loading) a model from a file, checking the model, transforming it, persisting (writing) the transformation, or generating code based on it, but user defined components are allowed as well. Within a workflow run, the check component verifies a model against the Check constraints specified at the metamodel level. If validation errors occur, these are reported on the console. The error messages contain information related to the name of the constraint's context metaclass, the name of the instance that breakes the constraint, plus the error/warning message specified by the user. No other support for identifying and correcting the error is provided, such as automatic object selection and partial evaluations.

(3) oAW includes a generator workfow component, that allows creating code in a programming language (e.g. Java) starting from a model file and some code generation templates. This is actually a model-to-text transformation. The template definitions are written using the Xpand language and contained in .xpt files. We have used this facility in order to simulate a forward engineering approach, by generating component interfaces' code, starting from a model. Generating a metamodel repository using oAW requires thus defining our own templates. This seems as a quite flexible alternative, but it has not been materialized yet.

(4) The shape of the code, its simplicity and inteligibility directly depends on the way templates are written by the user.

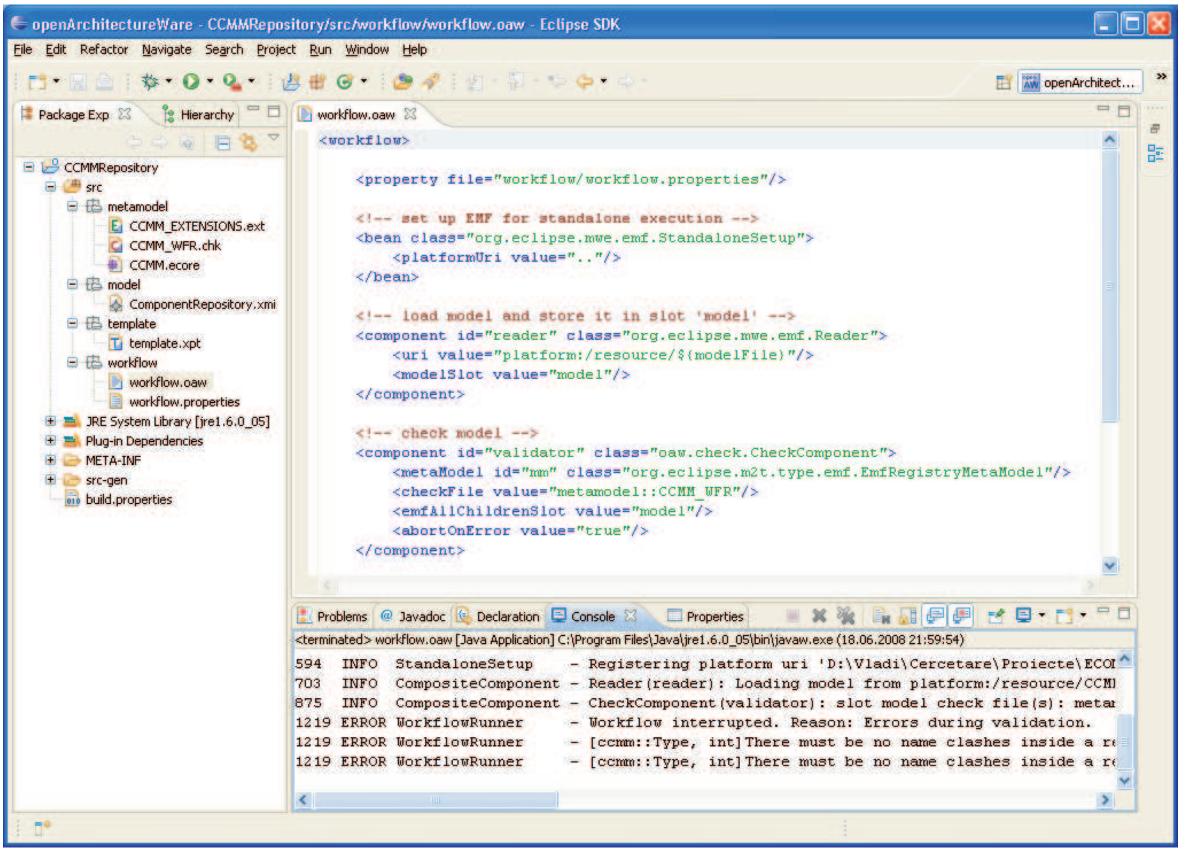


Figure 3.3: oAW workflow run

3.2 Process B: Structural Abstraction Subproject

Writer: Pascal André

Process B provides structural informations to process A (Fig. 1.1): an instance of the component metamodel with a corresponding annotated Java code. More precisely, process B is to build a couple (structural component model, annotated Java code) from a plain Java code and user-defined information. The two elements of the couple should be consistent.

In this section, we recall the initial goals and design of *process B*, present an assessment of the subproject, technical elements and future work.

3.2.1 Goals

The main goal of Process B was to abstract a component structure (components and architectures) from Java code and additional user-defined information. The goals stated on the Prague 2008 Workshop are recapitulated in the rest of this section.

A general view of the process B is given in figure 3.4; from plain Java code and user interaction, process B should produce an annotated Java code and a corresponding component model (both results must be consistent). Some restrictions apply to the first program release:

- Input
  - Annotations are those related to the Common Component Meta Model (CCMM) but do not include other component models yet (Fractal, Sofa, ...). The latter will be called **extended annotation**.
  - UML models are not accepted as direct inputs but are read by the user.
- Output
  - Only flat component models are targetted.
  - Process B is not directly responsible of the consistency between a model and the corresponding Java annotated code.
  - The conformance of the produced component model is checked at the metamodel level.

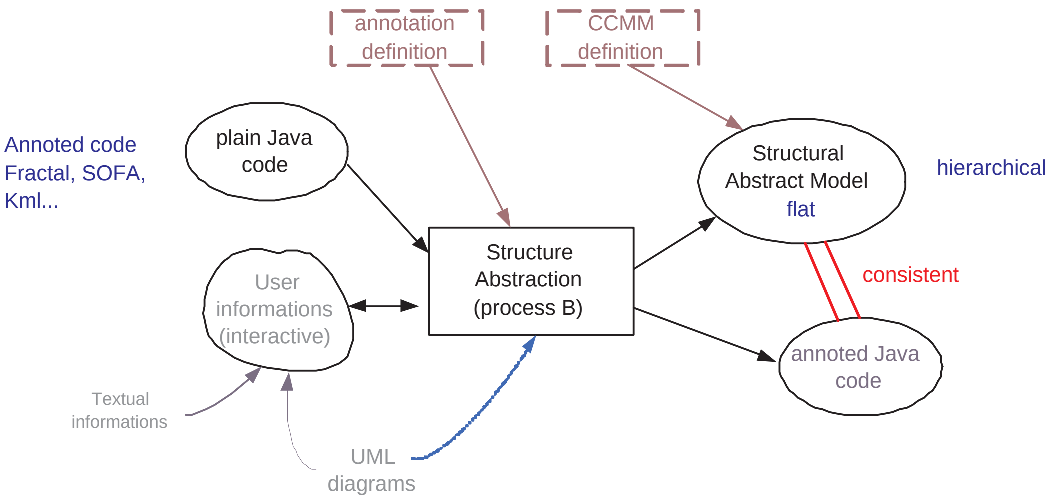


Figure 3.4: A general view of the process B

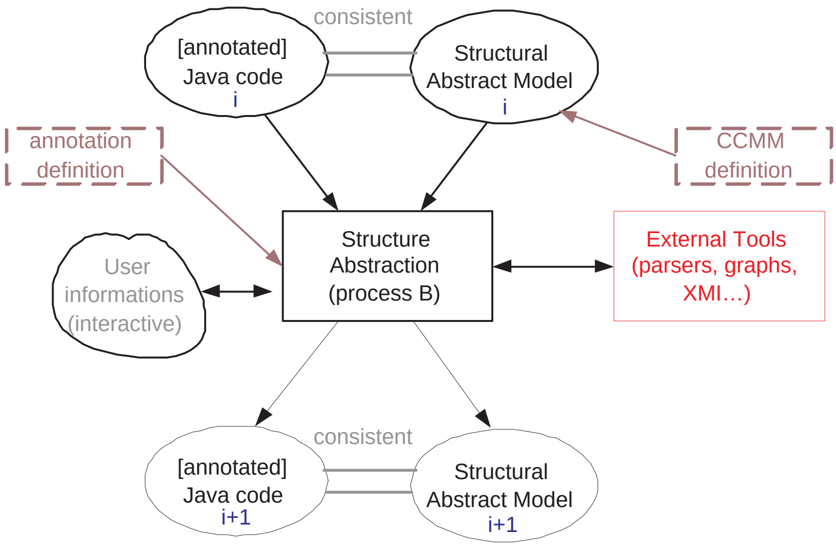


Figure 3.5: An iterative view of the process B

3.2.2 Design

The process B was designed as an iterative process (Fig. 3.5). This process is architectured around a toolbox (Fig. 3.6). One step in the process is the application of one of the tools (one transformation).

The inputs are

(1) An input model which is a couple $< cm, jac >$ where cm is a component model (an instance of the common component meta-model) and jac is a java annotated source code.

(2) User informations from any kind (textual, annotations, UML, user interactions...)

The output is a new couple $< cm', jac' >$.

External tools are used to process the transformations.

The input jac may be plain Java only. The input cm may be empty or disconnected from any jac.

The idea is to combine primitive transformations and develop a customised (or human driven) process B. Here are some of the primitive transformations:

(1) Annotate a Java program from user information.

(2) Build a component model from an annotated Java source.

(3) Build a component model from a plain Java source.

(4) Analyse a distributed program to detect components (deployment).

(5) Extract cluster using graph tools (grouping class into components, or grouping components into composite).

(6) Process model transformations such as fusion, selection... on the couple (code, model).

(7) Property Verification

- Check the consistency of a couple $< cm, jac >$.
- Check the completeness of a couple $< cm, jac >$.
- Check special system properties (various kind of compatibility, ...)

(8) ...

Important remarks:

(1) Note that combining transformation 1 and 2 provides a first result of process B which can be reusable in process A.

(2) Note also that input and outputs need format filters (reader, writer) which are common to all subprojects.

(3) Note also that some of these transformations ought to be used in the other subprojects.

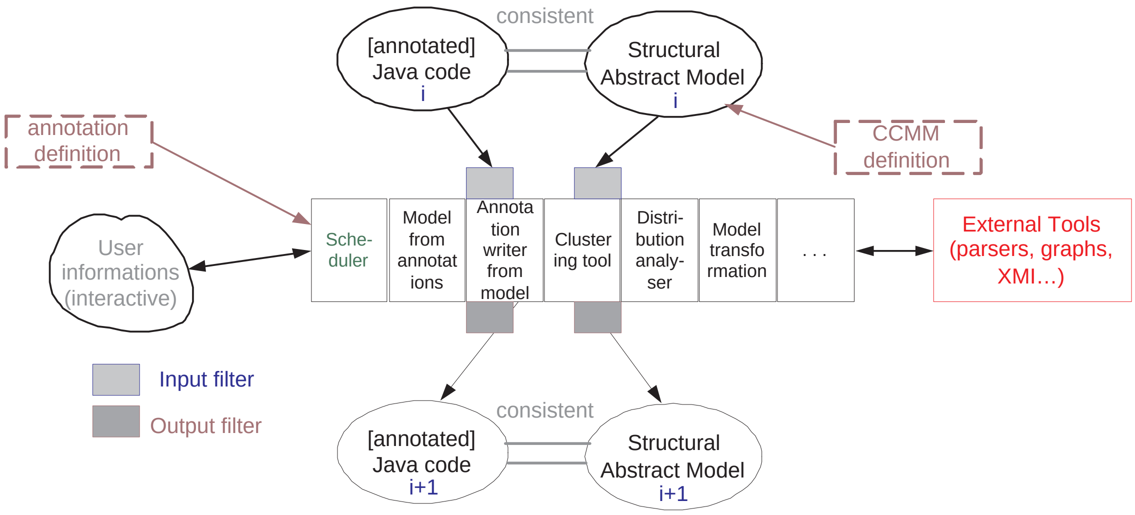


Figure 3.6: An architectural view of the process B

3.2.3 Assessment

A first prototype of the toolbox was implemented by a group of four students of a Master of computer Science the University of Nantes. A compressed archive of their work is available on the SVN repository on directory `processB` named `MasterOpProjectFinal.zip`. This files include the source programs and the documentation. The work overpassed the context of process B because it also required and implemented a simple metamodel management (using the CMM 1.0 specification). The experimentations were led with a small subset of the CoCoME case study. A report relates their work [BFFD08]. Here are some rewrited pieces of this report.

Project Goals

The goal of this master project is to contribute in the conception and in the implementation of the collective toolbox (Fig. 3.6). Our work contains several steps. In the first part we should understand the concept of components architecture, the global architecture of the reverse-engineering application and the components metamodel. In a second part, we should understand how the annotation language and the Java code management tools works. In the last part, we must implement the tool wich instanciate model from annotations and generate code from models.

Project Organisation

To be as productive as possible, we divided the development stage in two parts, each of them are realised by a couple of students.

- The management of annotations :
  This work was also divided in two parts :

  (1) The reading of annotations :
  We have to create the library of annotations. Afterwards, it will be necessary to write a grammary with differents annotations for read them in a JAVA code. When this work is satisfied, we can extract the structure of the Java code in a Component Model.

(2) The writing of annotations :
To do that, we need the Component Model. Thanks to the last and the library of annotation, we can give a Java code annotated, which respect the structure of Model given.

- The management of models :
As following, we have divided this part in two steps.

(1) The models transform :
If we give a description of a model in XMI, for exemple, we can transform it in SOFA or Kmelia. In this step, it have to write the transform rules.

(2) The instanciation of model :
If the user gives a Component MetaModel and complementary informations, this part allows to obtain a Component Model.

In the following diagram, we summarize the division of work :

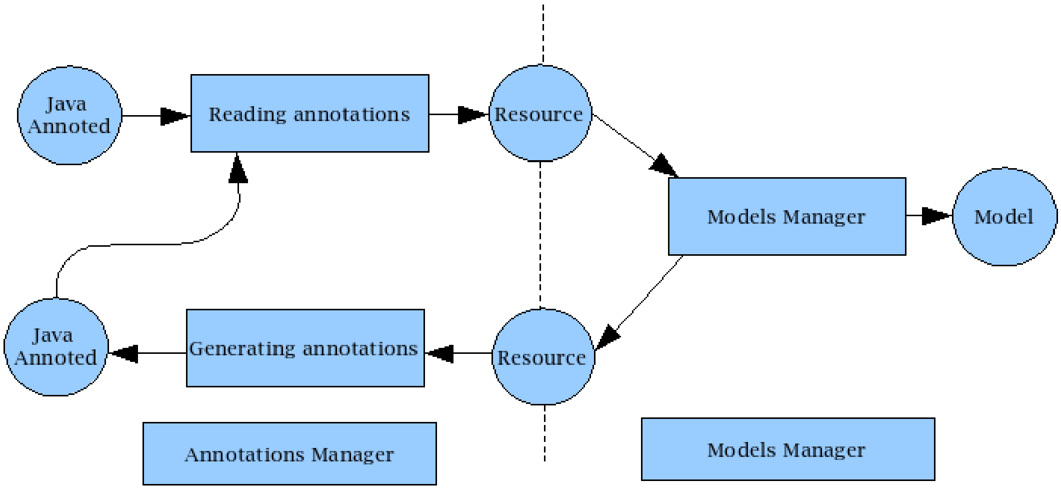


Figure 3.7: Process B: Master Project Organisation

Both parts are relatively distinct. The management of annotations was realised by Tanguy and Claire and the management of models was realised by Guillaume and Vincent.

Integration

The whole process was implemented by an Eclipse Plugin (the documentation is available on SVN).

Experimentations

The experimentations were led with a small subset of the CoCoME case study. We use for the tests the three following components present the CoCoME case study: `:CashBoxController`, `:PrinterController`, `:ScannerController`. These three components are contained in the component `:CashDesk` (Fig. 3.8).

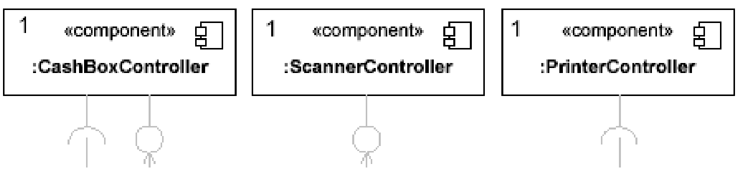


Figure 3.8: Process B:Master Project: CoCoME subset

Each component has a package name beginning with `org.cocome.tradingsystem`. The following package name is the composition of the name of the composite components which contain this component. There are a Java interface of the component and also a folder named `impl` which contains the java classes that implement the interface.

At first time, we tested a single class that contained all the annotation. This class allowed to generate the structure that is required for the generation of the model. In a second part, this structure is used to instanciate the metamodel. After, the structure is exported to another structure to the part that writes the annotaations. In the part that manages the writing of the annotation, we write the annotation corresponding to the intanciated model in Java classes that are not annotated. We check that the automatic annotated classes are exacly the same that the clesses that we annotated manually.

```
@InComponent(annotation_scr = {"Manual"}, component_name = "CashDesk")
public class CashBoxControllerEventHandlerImpl implements MessageListener,

        CashBoxControllerEventHandlerIf {

    final String CHANNEL_CONNECTION_FACTORY = "ChannelConnectionFactory";

    private String topicName;

    private Context jndiContext;

    private TopicPublisher cashBoxPublisher;

    private TopicSession topicSession;

    private Logger log = Logger
            .getLogger(CashBoxControllerEventHandlerImpl.class);

    @Businessattribute(annotation_scr = {"Manual"})
    private CashBox cashbox;

    @Initmethod(annotation_scr = {"Manual"}, name_of_the_component = "CashBox")
    protected CashBoxControllerEventHandlerImpl(CashBox cashbox,
            String eventchannel) {
        try {
            this.cashbox = cashbox;

            topicName = eventchannel;

            jndiContext = new InitialContext();
```

Figure 3.9: Process B:Master Project: One class of CoCoME annoted

Then, we tested the three CoCoME components. The previous approach has been used on the classes of these three components, we checked that the classes annotated by our program were the same that the classes that we annotated manually.

3.2.4 Tools and techniques

In this section we provide technical elements for the Master project.

Annotation Management

`+++ to write +++`

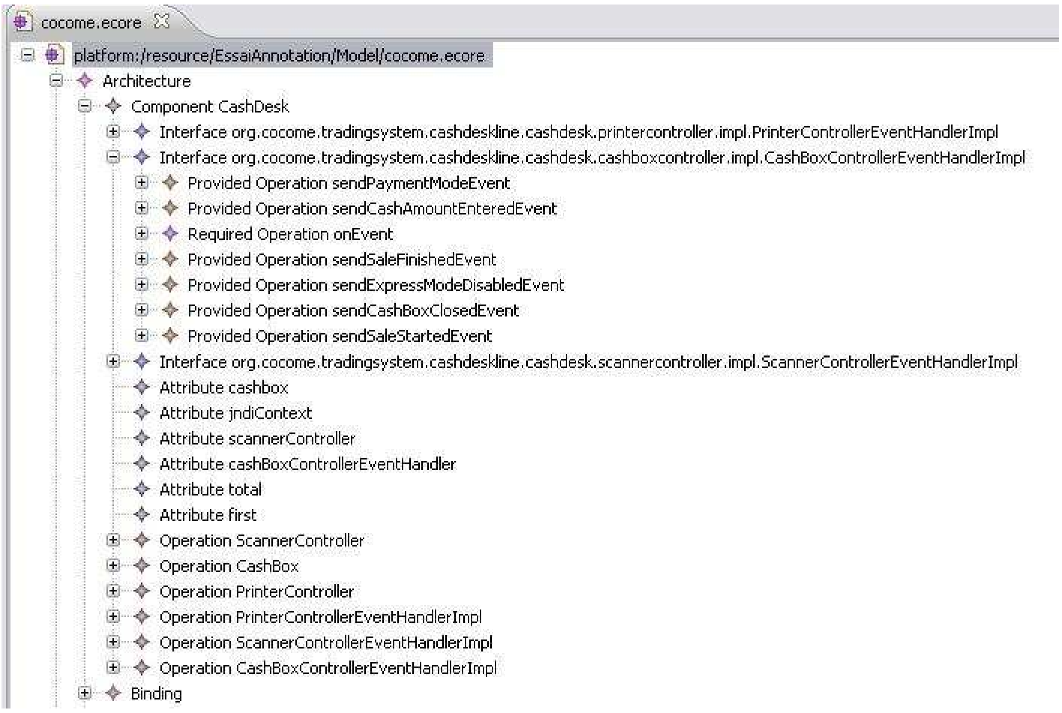


Figure 3.10: Process B:Master Project: Extract of CoCoME generated model

Reading annotations

Generating annotations

CCM Model Management

```
+++ to write +++
```

Instanciation of models

Reading models slides GA

The Student Project Write Java annotations and APT processing Write yet another version of a CCMM ;-) : CCM Architecture.ecore Instanciate a Model from the Annotated Java code Generate Annotated Java Code from the model

Before the Demo What was done before the Demo DL eclipse 3.3 DL EMF Copy the CCM Architecture.ecore in the workspace Add ProjetOp 1.0.0.jar to eclipse plugins directory

Current Status The project is on the econet svn Most students are about to be unreachable Multi source annotations to be done Problems with automatic build Code Generation is not working correctly

B transformations and tools

This is a simple overview. Recode opened a new perspective (see section 4.3.2 of chapter 4)

(1) Annotate a Java program from user information.
This program needs input/output functions for annotating Java sources.
Some tools are

- Java parsers, analysers... see section B.1 on page 53.

- JDK 5.0 Java Annotation Processing Tool APT[1].
- A program that lead the interactions.
- XML reader/writer.

(2) Build a component model from an annotated Java source.
Having an annotated Java program, one can build the corresponding model, providing we have the good filters and formats (see the adapted transformations).

(3) Build a component model from a plain Java source.
This can be obtained by combining other transformations. Since the input model is empty, the human must provide many informations and can be helped by the cluster tool.

(4) Analyse a distributed program to detect components (deployment).
One way to find components is to analyse the distribution framework. Components in this case are linked to deployment nodes. We can use RMI analysis for example (or corba ?).

(5) Extract clusters (grouping class into components, or grouping components into composite).
We need graph tools to analyse component architectures.

(6) Process model transformations such as fusion, selection... on the couple (code, model).
In collaboration with the team working on the metamodel we have to develop transformations on models and their pending Java annotation transformations.

(7) Consistency checker.
In collaboration with the team working on the metamodel we have to develop tools that check the consistency between models and their corresponding annotated Java programs.

(8) Filters.
In collaboration with the other teams we have to define the formats and to develop utilitary programs to read and write on the adopted format (XMI, MOF-XMI, Ecore, Java Model API, ...).

(9) Scheduler.
This program will chain the transformation in order to build interactive B processes.

3.2.5 Future Work

This task is led by the COLOSS group; the OBASCO group also contribute significantly to the toolbox; the LCI team will bring its experience on reverse-engineering tools.

Full Annotation Management

`+++ to write +++`

The program is designed as a set of tools which can be developed independently provided the interfaces are well defined (see section 3.2.6). The list of tools is open and will be extended each time we need another tool.

We have to distribute the transformations on the participants and to define which transformations are to include in each delivery. Transformations 1 and 2 are basic transformations and have to be implemented later with a core abstractation process (transformation 8) in the beginning of February 2008. These transformations are mandatory to test the model management module and the interface adequacy. Then the other modules will be added by team members.

First results on the structural analysis tool are expected by the time of the second workshop (Nantes 2008). Results on extraction back-ends are expected till the third workshop (Cluj 2008).

[1] java.sun.com/j2se/1.5.0/docs/guide/apt/

Metamodel API integration and experimentation

Metamodel API integration

OBASCO mŔne une Őtude (prŐsentŐe Ĺ Cluj) sur l'abstraction de classes en composants, par exemple par aggrŐgation utilisant des techniques de graphes.

Peut-etre faire un topo general

As I promised during the workshop, I've just put the implementation of the jAbstractor tool into the repository (svn://aiya.ms.mff.cuni.cz/econet/processA/jabstractor).

Paragraph sur Madeleine

OBASCO mŔne une Őtude (prŐsentŐe Ĺ Cluj) sur l'abstraction de classes en composants, par exemple par aggrŐgation utilisant des techniques de graphes.

Automating the embedding of Domain Specific Languages in Eclipse JDT peut-etre voir

Writer: Pascal André

+++ TODO: *This is currently a draft version.* +++

3.3 Process A: Behavioral Abstraction Subproject

Writer: Ondrej Sery, Tomas Poch

The goal of Process A was to analyze options of reverse engineering behavior specification from Java code and additional architectural information in the form of Java annotations. The architectural information is the expected outcome of Process B. Moreover, prototype implementation of a Generic analysis tool (GAL) was anticipated. The goals stated on the Prague 2008 Workshop are recapitulated in the next section.

3.3.1 Goals

Three of the groups participating in the project have developed their own formalism for behavior specification. Therefore, in order to allow extraction of behavior in any of the formalisms, the goal is to design the behavior reverse engineering process as general as possible.

To be more specific, the formalisms considered are:

- Enhanced behavior protocols (EBP) developed by DSRG,
- eLTS developed by COLOSS,
- STS developed by OBASCO.

The individual behavior specification formalisms differ a lot, which makes creation of a general tool a challenging task. However, steps common to extraction of any behavior specifications (in particular behavior protocols and LTS-based formalisms eLTS and STS) might be identified. Thus, the general approach is to divide all necessary steps of behavior extraction into two parts: (i) steps common to all formalisms, and (ii) steps specific to a particular formalism.

The first part will be implemented in a General analysis tool, while the second part will be performed by back-ends specific to a particular formalism. To prevent reinvention of the wheel, the analysis tool is to be implemented using existing libraries/tools/platforms (for parsing Java sources and annotation extraction, etc.). To sum it up, the goals of reverse engineering behavior specification are as follows:

(1) Find a suitable libraries/tools/platforms for analysis of Java sources.

(2) Create a generic Java analysis tool which produces an intermediate representation of behavior suitable for subsequent creation of concrete behavior specifications in a chosen formalism.

(3) Create formalism-specific back-ends for extraction of behavior specification from the intermediate specification.

3.3.2 Assessment

So far, a prototype implementation of the GAL—called *jabstractor*—has been created. The use a Recoder library [6] to parse Java source codes and then employs a set of transformations over the abstract syntax trees (AST). Figure 3.3.2 depicts the transformation process from Java sources to a form of either LTS or regular expression. The LTS form is designed to preserve as much information from the original sources as possible. This is essential for further transformation into other formalisms (e.g., STS and eLTS). However, these transformations are out of the scope of the project.

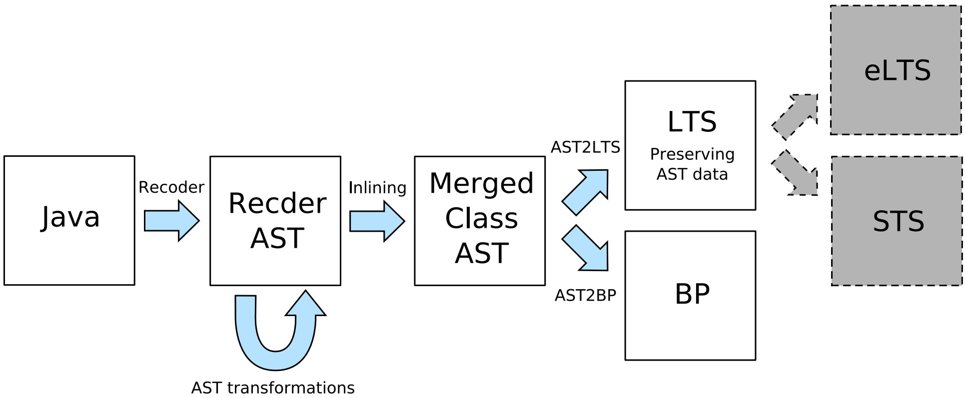


Figure 3.11: Workflow of the process A

3.3.3 Tools and techniques

The input of the jabstractor tool is a set of annotated Java sources, a name of a component and a specification of intended usage of the primitive component (Fig 3.12). The annotations were defined in [ACPR07]. The sources are parsed using the Recoder tool which results in an *abstract syntax tree* (AST) of the involved Java classes. As the Recorder tool is specialized for Java, it provides many useful features; e.g., resolving references, side-effect removal and so on. Moreover, it provides a framework for building user defined transformations based on the visitor pattern.

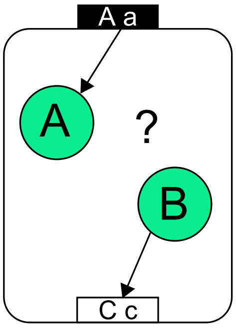
In the next step, Recoder transformations are applied to make the original Java code closer to the capabilities of target formalisms. The result is still a parse tree of a Java code. The strategy is to stick with Java parse tree as long as possible and perform the transformation into the target formalism as the last step. The motivation is reuse of transformations independently on the target formalism. The target formalisms have the power of finite automatons, while Java is Turing complete. However, as the target formalisms are intended to capture just the behavior on a component boundary, internal computations, where the complexity is often hidden, may be omitted. In particular, the omitting works in terms of following definitions:

Definition 1 *Let* a *and* b *be AST nodes. We say that* b *is reachable from* a *if*

- b *is in the subtree of* a
  or
- *there is a method* f *declaration, such that node* call f *is reachable from* a, *and* b *is reachable from method* f *declaration*

The omitting transformation the set R_{Prov}, all AST nodes reachable from a *provided* method declaration, and R_{Req}, all AST nodes, such that a *required* or *business* member variable reference is reachable from it. The sets of provided methods, required and business member fields are defined in source codes by annotations. Finally, an intersection $I = R_{Req} \cap R_{Prov}$ is computed. Then all statements (AST nodes) that are not in the set I are removed together with all declarations which are not referenced any more. There are also other transformations, which can be applied at this point, depending on the target formalism (side effect elimination, removing of a method parameters, removing of recursion).

The result of transformations is a set of simplified Java classes (Fig. 3.13). In the next step, these classes are merged into single one, roughly corresponding to a component. The merged class contains:




```
public class A{
  @business boolean cond;
  B _b = new B();
  @provided
  public void a(){
    for (int i = 0; i<10; i++)
      {b();b();}
    }
  public void b(){
    if (cond) hp();
      else d();
    }
  public void hp(){
    // halting problem
  }
  public void d(){_b.x();}
}

public class B{
  @required C c;
  public void x(){c.c();}
}
```

Figure 3.12: Example input of the process A. There is one instance of the A class and one instance of the B class within the instance of the component

```
public class A{
  @business boolean cond;
  B _b = new B();

  @provided
  public void a(){
    for (int i = 0; i<10; i++)
      {b();b();}
  }

  public void b(){
    if (cond) NULL;
      else d();
  }

  public void d(){_b.x()}
}

public class B{
  @required C c;
  public void x(){c.c()}
}
```

Figure 3.13: Example after omitting an internal behavior

- Method declaration for each method provided by the component
- Constructors, thread definitions
- Field for each required interface
- Field for each business member

A method of the merged class may only access a business field and invoke methods on required fields. Merging of classes involves method and member fields inlining. There is a number of issues regarding both control flow (recursion, method overloading, inheritance, virtual methods) and data (points-to-analysis, method parameters). Typically, these are often related to the halting problem. In such cases, overspecification is applied.

Having the merged class in hand, the final step—transformation into a particular target formalism—can be done (Fig 3.14).

3.3.4 Objectives and organisation

In order to proceed and provide a working tool chain, following tasks must be done. First, annotations used by the process A and process B should be synchronized. Also, the jabstractor tool should be improved to use method

```
public class Merged{
  @required C c;
  @business boolean cond;
  bool _b_mode;

  @provided
  public void a(){
    for (int i = 0; i<10; i++){
      if(cond)NULL;
         else if (_b_mode) c.c()
      if(cond)NULL;
         else if (_b_mode) c.c()
    }
  }
}
```

```
?a{
  {
    switch(cond){
      case TRUE:
      case FALSE: !c.c;
    };
    switch(cond){
      case TRUE:
      case FALSE: !c.c;
    }
  }*
}
```

Figure 3.14: The merged class and the result in EBP

parameters. At the Nantes workshop, opportunities for use of the Recoder tool also in process B emerged. In order to minimize effort, a wrapper encapsulating the Recoder functionality used by both processes should be created.

Tasks related to the jabstractor tool (and process A) are to be carried out by the DSRG team. Synchronization of annotations is to be done in cooperation with COLOSS.

Chapter 4

Working Sessions

This chapter relates the working sessions.

4.1 Introduction

The goals of the working sessions are mainly to capitalise the experience and to fix a roadmap for the project continuation. This means to clarify the common issues:

(1) Metamodel: validate the metamodel in order to benefit from an aggreed one for the end of the project.

(2) Interface: define better requirements and provision of the subprojects including annotation definition, tools sharing, special requirements, API...

(3) Case study: define a convenient subset of the benchmark used by all subprojects.

From the organisation point of view the objective of the working sessions is to refine the task initial definition and planning (*the detailed objectives in a feasible manner, to define clearly the concrete and coordinated contribution of each partner, to define task, products and results, to organise tasks (responsibilities, contributors, schedule...) until the next workshop*. Last, everyone was invited to think about a project continuation and valorisation by publishing results.

4.2 Metamodel Specification

Chapter 1 and 2 of [AP08] are a detailed explanation on the work leaded in this working group. The reader is invited to consult these chapters. The whole document [AP08] is the result of the validation process leaded by this group in the working sessions.

4.3 Annotations and interfaces

A working group was build upon the interface between processes it included annotations, tools and special requirements.

4.3.1 Annotations Update

The annotations defined in the workshop of Prague have been refined in order to take into account experience gained from the work on processes A and B, and also to allow arrays of sources.

Component - Class Relation

```
/**
 * One or more Java classes can be assigned to a single component. Such an
 * assignment is specified by this annotation.
 */
@Target(ElementType.TYPE)
public @interface InComponent {
  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();

  /**
   * @return the array (one entry per annotation source) containing component
   *   Names which the annotated class is assigned to. If a single
   *   source declares the class to participate in several components,
   *   its entry should be a comma-separated list of component name
   */
  String[] componentName();
}
```

Entry points

```
/**
 * This class is the first instantiated and is responsible (its constructor) for
 * the instantiation and initialization of the component's content.
 */
@Target(ElementType.TYPE)
// Should be just a class
public @interface InitClass {
  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();

  /**
   * @return the array (one entry per annotation source) containing component
   *   Names for which the annotated class provides the initialization.
   *   If a single source declares the class to participate in several
   *   components, its entry should be a comma-separated list of
   *   component name
   */

  String[] componentName();
}

/**
 * The component content is instantiated and initialized by a method ( it can be
 * a constructor, a static method or an initialization method to be called after
 * the default constructor).
 */
@Target( { ElementType.CONSTRUCTOR, ElementType.METHOD })
public @interface InitMethod {
  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();

  /**
   * @return the array (one entry per annotation source) containing component
```

```
   *   Names for which the annotated method provides the initialization.
   *
   */
  String[] componentName();
}
```

Interfaces

Provided

```
/**
 * In Java sources, a provided interface might be in a form of a class
 * attribute. The attribute stores a reference to a class implementing the
 * provided interface.
 *
 */
@Target(ElementType.FIELD)
public @interface Provided {
  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();

  /**
   * @return the array (one entry per annotation source) containing the name
   *   of the interface represented by this field
   */
  String[] modelIfaceName();
}

/**
 * All methods of the specified Java interface (which the annotated class has to
 * implement) are marked as a part of the provided interface of the component
 *
 */
@Target(ElementType.TYPE)
public @interface ProvidedIf {
  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();

  /**
   * @return the array (one entry per annotation source) containing the name
   *   of the component interface represented by this type
   */

  String[] modelIfaceName();

  /**
   * @return the array (one entry per annotation source) containing the name
   *   of the java interface which is defining one component Interface
   *   If a single source declares to participate in several components,
   *   its entry should be a comma-separated list of java interface
   *   names (for instance {"ActionListener,WindowListener"}
   */

  String[] javaIfaceName() default { "" };
```

```
}

/**
 * The method is a part of the provided interface of the component
 *
 */
@Target(ElementType.METHOD)
public @interface ProvidedMethod {
  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();

  /**
   * @return the array (one entry per annotation source) containing the name
   *   of the component interface which the annotated method is part of.
   *   If a single source declares to the method participate in several
   *   interfaces, its entry should be a comma-separated list of
   *   interface names
   */
  String[] modelIfaceName();
}
```

Required

```
/**
 * In Java sources, a required interface is present in a form of a class
 * attribute. The attribute stores a reference to another component, whose
 * provided interface is bound to this required interfaces. Therefore, the
 * target of the annotation for required interface is an attribute of a Java
 * class
 */
@Target(ElementType.FIELD)
public @interface Required {
  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();

  /**
   * @return the array (one entry per annotation source) containing the name
   *   of the interface represented by this field
   */
  String[] modelIfaceName();
}
```

Business elements

```
/**
 * all the instances of such type are important for a component behaviour.
 */
@Target(ElementType.TYPE)
public @interface BusinessType {
  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();
}
```

```
/**
 * Marks particular Java class attributes as important for business logic.
 */

@Target(ElementType.FIELD)
public @interface BusinessField {

  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();
}


/**
 * Marks particular method parameter as important for business logic.
 *
 */
@Target(ElementType.PARAMETER)
public @interface BusinessParameter {
  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();
}
```

4.3.2 Interface with Recoder

+++ TODO: *Gilles* +++

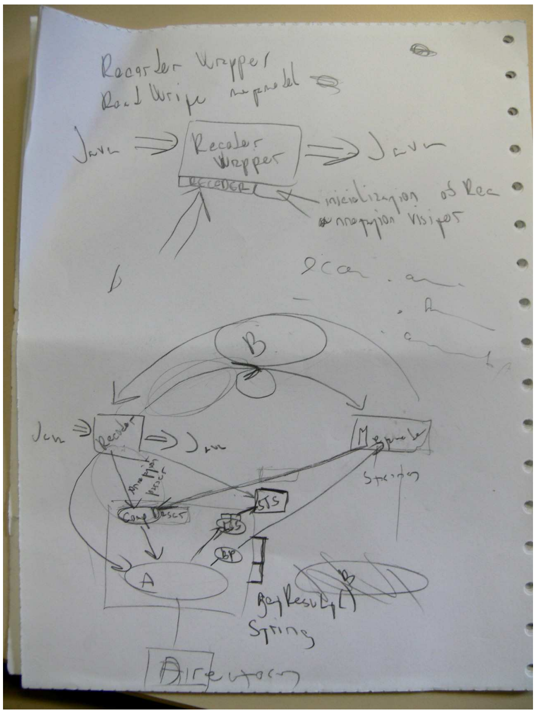


Figure 4.1: Recoder wrapper and processes

4.4 CoCoME

The CoCoME case study is used as a **benchmark** for each of the three subprojects. The whole benchmark is too big to serve as support for the experimentations. In order to select a subset of it as the experimentation field a short working group was installed.

The constraints are:

- The selected subset must be large enough to include representative examples for each subproject (concepts and constraints for the metamodel, primitive component for the behaviour abstraction, primitive and also composite components for the structural abstraction.
- The selected subset must be as small as possible to avoid time consuming instanciations.
- The slice is vertical (UML model and Java code).

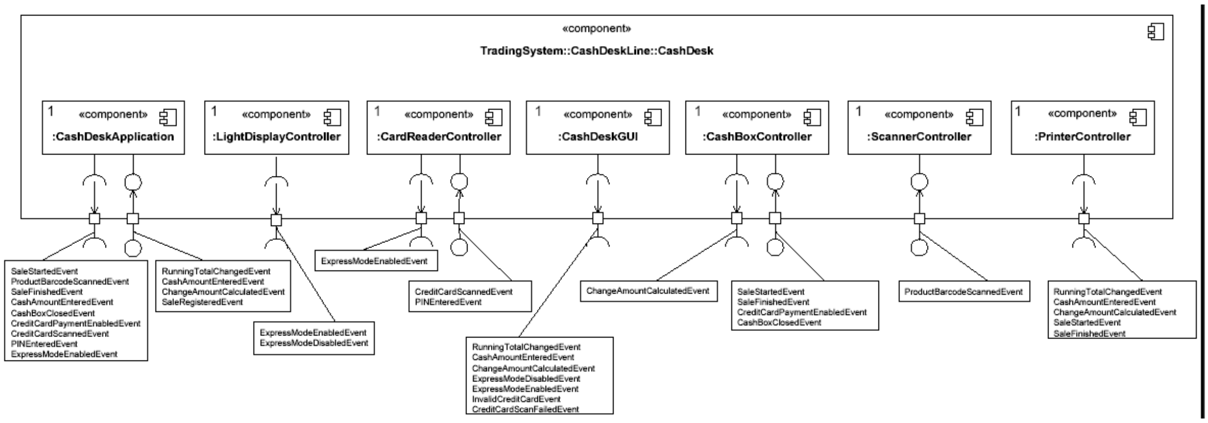


Figure 4.2: CoCoME subset 1

We retain two included subsets related to two deadlines:

- *Cluj*: The `CashDesk` composite component for the structural abstraction. We retain two included subsets:
  - The `CashDesk` composite component for the structural abstraction.
  - The `CashDeskApplication` primitive component, which is a component of the `CashDesk` composite component that helds a dynamic behaviour.
- *End of project*: The `CashDeskLine` composite component, which is the front-end subsystem of the application.

4.5 Task, responsabilities, schedule

Figure 4.4 is a snapshot of the discussions about tasks, responsabilities and deadlines for the processes subproject.

- Metamodel (Vladiela)
  - CCMM specification + special requirements (input)
  - Metamodel verification
  - API generation and testing
  - Deadlines
    - ∗ specification: 7 of june 2008
    - ∗ version 1 (EMF) : 22 of june 2008
    - ∗ version 2 (oAW) : end of june 2008

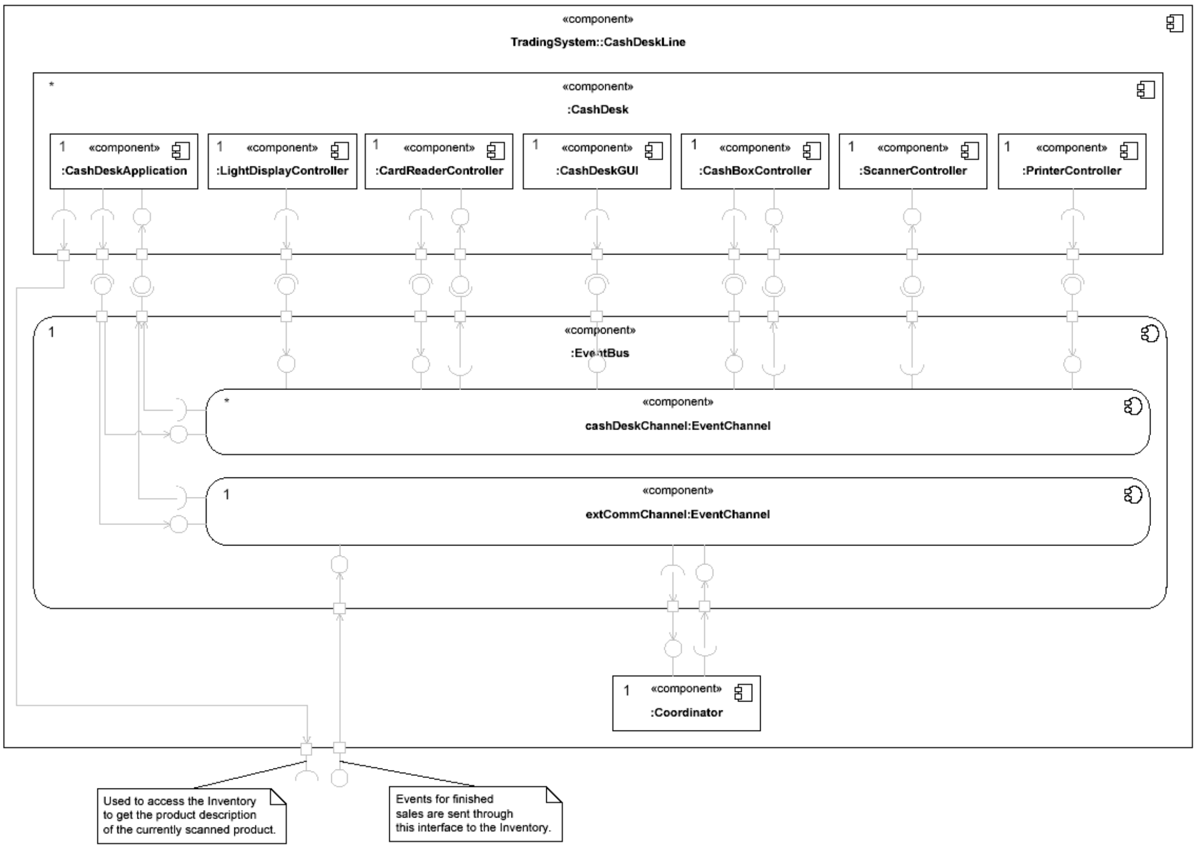


Figure 4.3: CoCoME subset 2

- Process A (Tomas)
  - Behaviour abstraction
  - Submodel instanciation
  - Deadlines: september (the last team in the dependency chain)
- Process B (Gilles)
  - CCMM instance of CoCoME + EMF API + Java files (input from LCI)
  - Input/Output of Java annotations
  - Deadlines : begin of july 2008
  - Studies for other tools of the toolbox
- Case Study (Petr)

We reminded the current (shared) set of tools and framework we use for the project:

- Code: RECODER/APT
- OCLE/EMF/oAW/ATL

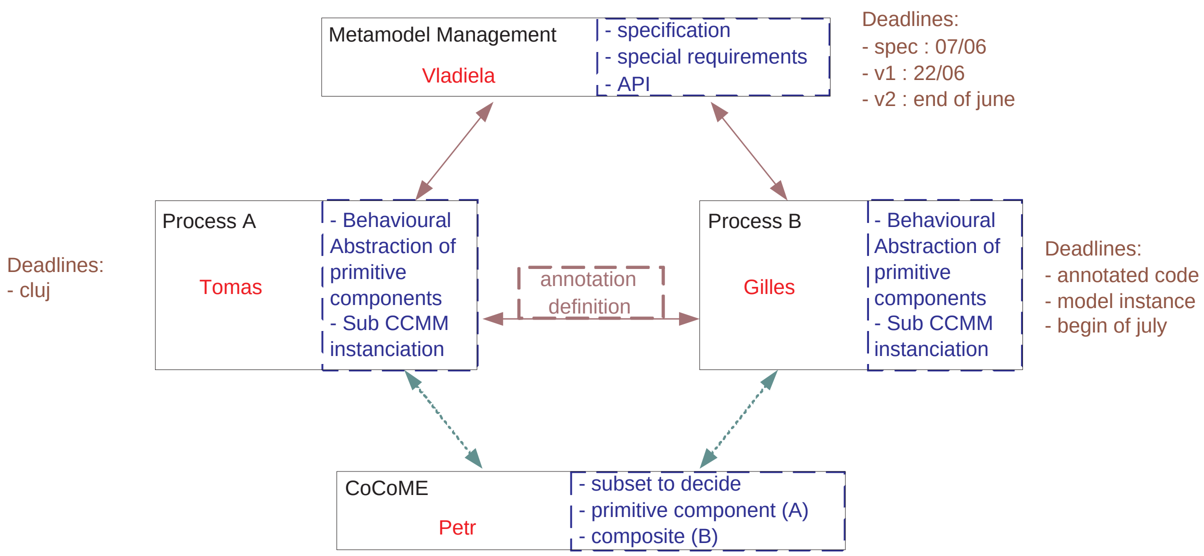


Figure 4.4: Workshop whiteboard 3

Chapter 5

Conclusion

We report many informations of the workshop in this document. This work has also been intended to be the technical part of the project second year report together with the metamodel specification document [AP08] produced in the same period.

The workshop indicates the current state of the project, which is a bit in hurry againts its planification. Small prototypes have been produced for each subproject, bringing some experience on the architecture and technical issues.

Common parts have been discussed and validated during the workshop in order to allow everyone to develop the solutions on step further until the next workshop in three months.

Bibliography

[AAA06a] Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking Component Composability. In *5th International Symposium on Software Composition*, volume 4089 of *Lecture Notes in Computer Science*. Springer Verlag, 2006.

[AAA06b] Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking Component Composability. In *5th International Symposium on Software Composition, SC'06*, volume 4089 of *LNCS*. Springer, 2006.

[ACPR07] Pascal André, Dan Chiorean, Frantisek Plasil, and Jean-Claude Royer. ECONET Project - Prague Workshop Report, September 2007.

[Ald05] Jonathan Aldrich. Open modules: Modular reasoning about advice. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th European Conference*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168, Glasgow, UK, July 2005. Springer Verlag.

[AP08] Pascal André and Vladiela Petrascu. ECONET Project - CCMM Specification v. 1.1 , June 2008.

[BFFD08] Tanguy Beneteau, Vincent Fouquet, Claire Fromonteil, and Guillaume Doux. Operationnal project: Reverse-engineering on JAVA. Master's thesis, MSc on Software Architectures, University of Nantes, March 2008. directed by Pascal André and Gilles Ardourel.

[BHM06] Tomas Barros, Ludovic Henrio, and Eric Madelaine. Model-checking distributed components: The vercors platform. In *International Workshop on Formal Aspects of Component Software (FACS'06)*, Prague, September 2006. Electronic Notes in Theoretical Computer Science (ENTCS).

[BR02] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[CCG+04] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. *IEEE Trans. Softw. Eng.*, 30(6):388–402, 2004.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, London, UK, 2000. Springer-Verlag.

[CKSY04] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ansi-c programs using sat. *Form. Methods Syst. Des.*, 25(2-3):105–127, 2004.

[Dam07] C.W. Damus. Implementing Model Integrity in EMF with MDT OCL, 2007. Eclipse Corner Articles, online at:
http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html.

[DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002 - Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188, Pittsburgh, PA, USA, October 2002. Springer Verlag.

[DLBNS06] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 79–88, Portland, USA, October 2006.

[Eis05] Cindy Eisner. Formal verification of software source code through semi-automatic modeling. *Software and System Modeling*, 4(1):14–31, 2005.

[emf] EMF website. http://www.eclipse.org/modeling/emf/.

[MK06] J. Magee and J. Kramer. *Concurrency: State Models and Java*. Wiley, 2nd edition, 2006.

[NN07a] Angel Núñez and Jacques Noyé. A domain-specific language for coordinating concurrent aspects in java. In Rémi Douence et Pascal Fradet, editor, *3ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2007)*, Toulouse, France, March 2007.

[NN07b] Angel Núñez and Jacques Noyé. A seamless extension of components with aspects using protocols. In Ralf Reussner, Clemens Szyperski, and Wolfgang Weck, editors, *WCOP 2007 - Components beyond Reuse - 12th International ECOOP Workshop on Component-Oriented Programming*, Berlin, Germany, July 2007.

[oaw] oAW website. http://www.openarchitectureware.org/.

[ocl] OCLE website. http://lci.cs.ubbcluj.ro/ocle/index.htm.

[PNPR05] Sebastian Pavel, Jacques Noyé, Pascal Poizat, and Jean-Claude Royer. A java implementation of a component model with explicit symbolic protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)*, volume 3628 of *Lecture Notes in Computer Science*, pages 115–125. Springer-Verlag, 2005.

[PP99] Radek Pospisil and Frantisek Plasil. Describing the Functionality of EJB using the Behavior Protocols, 1999.

[PP07] Pavel Parízek and František Plášil. Modeling environment for component model checking from hierarchical architecture. In *Third International Workshop on Formal Aspects of Component Software (FACS 2006)*, volume 182 of *Electronic Notes in Theoretical Computer Science*, pages 139–153. Elsevier B.V., 2007.

[PV02] F. Plasil and S. Visnovsky. Behavior protocols for software components, 2002. IEEE Transactions on SW Engineering, 28 (9), 2002.

[VKEH06] M. Voelter, B. Kolb, S. Efftinge, and A. Haase. From Front End To Code - MDSD in Practice, 2006. Eclipse.org, online at:
http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html.

Appendix A

Collaborative Tools

In this appendix chapter we provide informations on the Subversion repository and the wiki tools.

A.1 SVN Repository

The Subversion (SVN in short) repository was set up at DSRG (University of Prague) in october 2007. Reports, specifications and developments can be updated on this SVN repository.

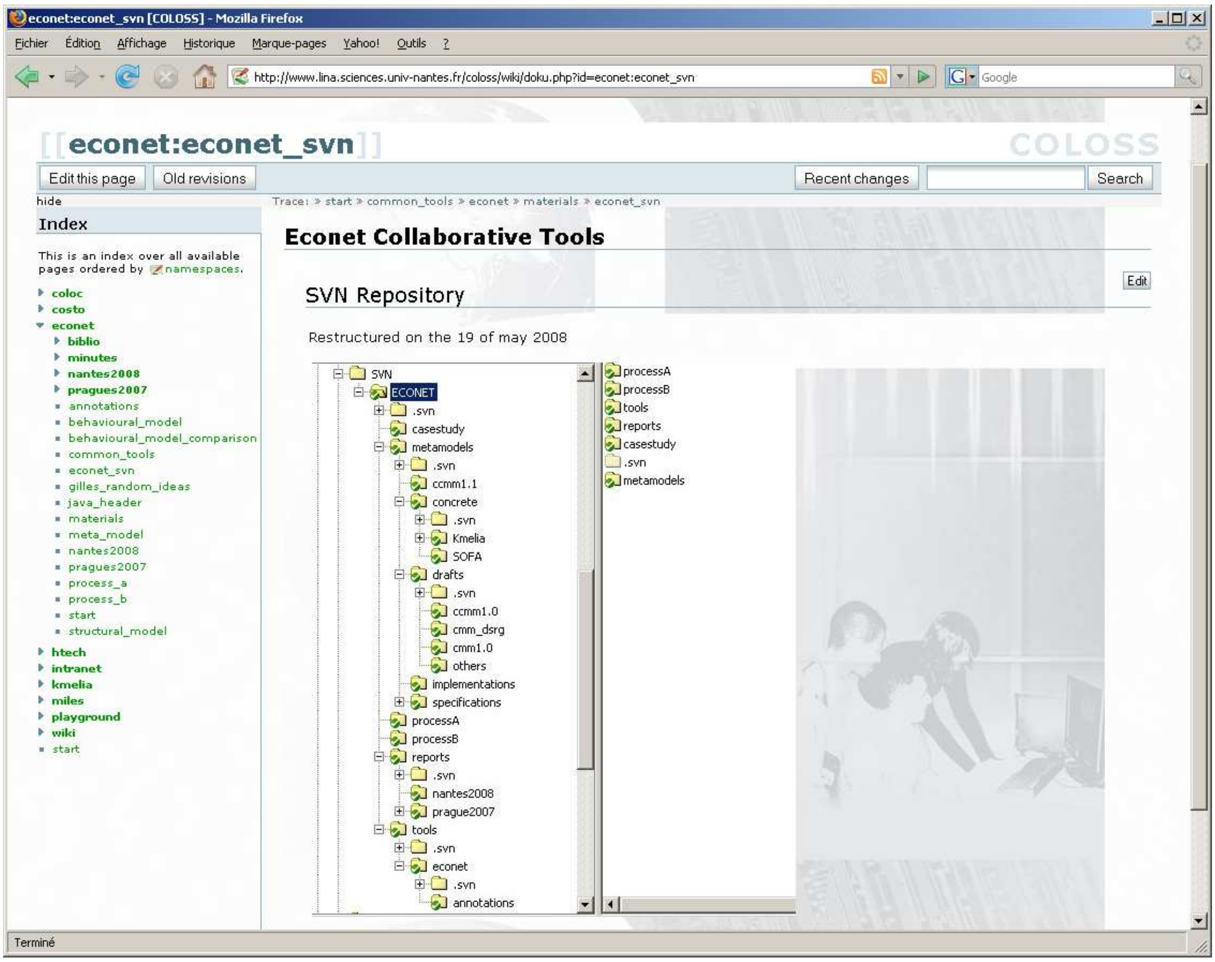


Figure A.1: Project SVN Repository

DSRG has set up a SVN repository for the project and put the report to it (in the directory reports/Prague2007).

The repository is running at svn://aiya.ms.mff.cuni.cz/econet In a separated email, I will send you login and password required to access the repository.

You can obtain svn from http://subversion.tigris.org/ Also, the svn documentation is available from the same site (direct url is http://svnbook.red-bean.com/).

A brief overview of the most important commands:

```
svn checkout svn://login@aiya.ms.mff.cuni.cz/econet directory
  Initial check out of the repository content to the specified
  directory.

svn commit
  Commits local changes to the repository

svn add <name_of_file_or_directory>
  Adds new file or complete directory to the repository.
  The command should be follow by "commit" (the "add" command
  just schedules files/directories to be added and "commit" really
  commits them and they become visible for others).

svn update
  Updates your (previously checked out) copy of repository
  by commits made by others.

svn help
  Overview of all commands

svn help <command>
  Detailed help about a particular command.
```

If you prefer a GUI client, you can use TortoiseSVN client (http://tortoisesvn.tigris.org/)

A.2 Wiki

This wiki was installed at LINA (University of Nantes, EMN) in april 2007. It includes discussions, a repository for project and workshop material, etc. The history of the project will be found on this wiki. In particular there are chapters for each workshop (see figure A.2.

Project material and documents are downloadable from both the wiki (figure A.3) and the SVN repository (figure A.1).

http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:materials:start

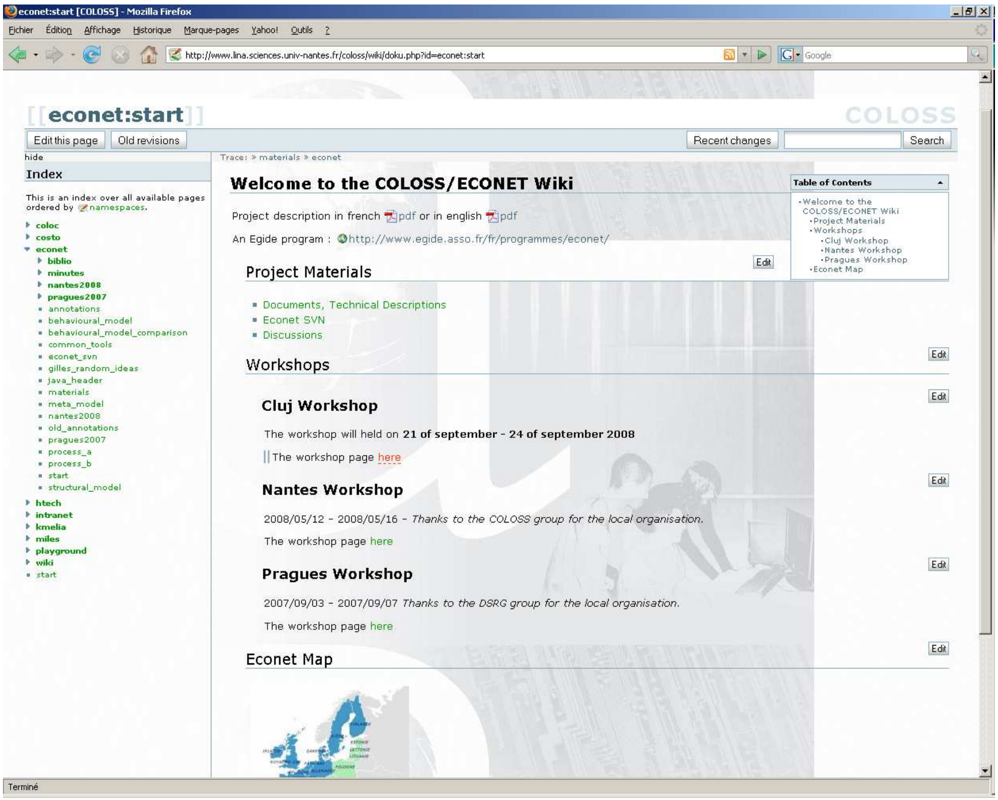


Figure A.2: Project Wiki

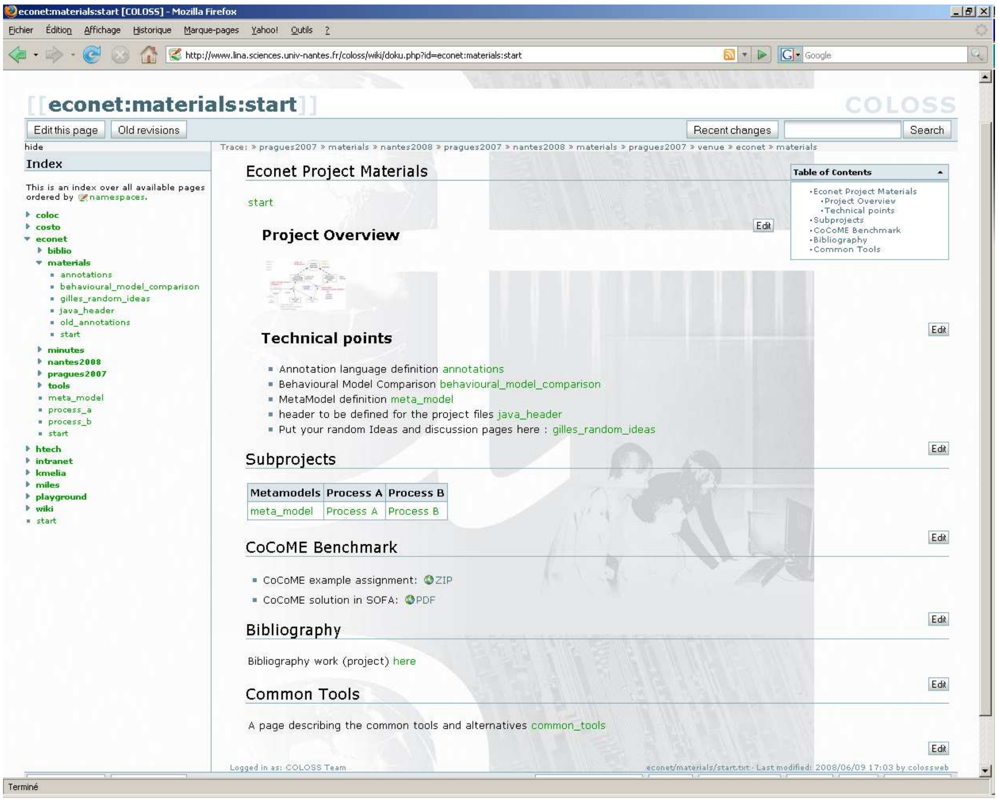


Figure A.3: Project material on the Wiki

Appendix B

Common Tools and Interface

In this appendix chapter we provide informations on the model and language tools. Interface between subprojects can be text files or XML files but this quite poor and each group will need to develop tools on Java and Models. In order to get a standard vision of the usable technologies we need to agree on the model and metamodel tools used in each subproject.

B.1 Java Tools

Java tools include annotation management and java code analysis.

B.1.1 Java/Annotation Tools

Several tools will be used in more than one subproject.

Tools Webography

(1) JavaCC, `https://javacc.dev.java.net/`

(2) Java Development Kit, `http://java.sun.com/`

(3) ANTLR, `http://www.antlr.org/`

(4) Java CUP, `http://www2.cs.tum.edu/projects/cup/`

(5) SableCC, `http://sablecc.org/`

(6) Recoder, `http://recoder.sourceforge.net/`

B.1.2 Tools for Java source analysis

Having the Java sources properly annotated, the question of how to extract the annotations and analyze the sources comes up. There is quite a choice of tools to be used for this purpose.

Possible options are:

- JavaC [2]—standard Java compiler from Sun—is a natural first option as it is standard part of the Java development kit (JDK) and features a reasonable interface for either annotation processing alone or to obtain the complete abstract syntax trees.
- JavaCC (Java Compiler Compiler) [1] is a generator of parsers. To create a parser, it uses a LL(n) grammar.
- ANTLR [3] is another parser generator which also uses LL(n) grammars.
- Java CUP [4] is also a parser generator, but in comparison to the previous ones it uses LALR(1) grammars. It is quite similar to the standard YACC and Bison tools. In contrast, it is written in Java.

- SableCC [5] is another LALR(1) parser generator.

In a case, the chosen parser generator does not provide a lexical analyser, a usage of tools like JLex and JFlex has to be considered.

Choosing the suitable tool will require deeper exploration and in-depth analysis of all features provided by the tools. The preferred option is to use JavaC, as it always guarantees to parse the current (and also older) version of the Java languages and also it does not introduce any third-party tool dependencies.

RECODER The *current choice* is the Recoder tool, available on a sourceforge project `http://recoder.sourceforge.net/`.
RECODER is a Java framework for source code metaprogramming aimed to deliver a sophisticated infrastructure for many kinds of Java analysis and transformation tools.

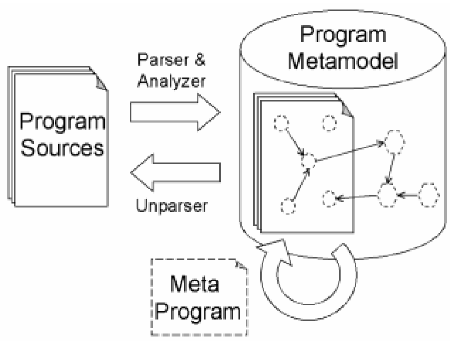


Figure B.1: Recoder Metaprogramming Cycle

The following table gives a short description of the different layers of RECODER features as well as the application perspectives that these layers offer:

- *Parsing and unparsing of Java sources*

  In addition to abstract model elements, RECODER also supports a highly detailed syntactic model - no information is lost. Comments and formatting information are retained. The pretty printer is customizable and will be able to reproduce the code (possibly improving upon it, but retaining given code structures) and to embed new code seamlessly.

  Possible applications: Simple preprocessors, simple code generators, source code beautification tools

- *Name and type analysis for Java programs*

  RECODER can infer types of expressions, evaluate compile-time constants, resolve all kinds of references and maintain cross reference information.

  Possible applications: Software visualization tools, software metrics, Lint-like semantic problem detection tools, design problem detection tools (anti-patterns), cross-referencing tools

- *Transformation of Java sources*

  RECODER contains a library of small analyses, code snippet generators and frequently used transformations.

  Possible applications: Preprocessors for language extensions, semantic macros, aspect weavers, source code obfuscation tools, compilers

- *Incremental analysis and transformation of Java sources*

  Transformations change the underlaying program model; for incremental and iterative use, this model has to be updated accordingly. Transformations have to take care of dependencies by updating their local data and setting back matching positions when necessary; however, RECODER will analyze change impacts for its model and perform updates automatically.

Possible applications: Source code optimization, refactoring tool, software migration programs (Smart Patches), design pattern, clichés and idiom synthesis, architectural connector synthesis, adaptive programming environments, invasive software composition

B.1.3 Model Engineering Tools

We need tools for model management, preferably on Eclipse. We already discussed on a modeling tool around Eclipse technologies (Ecore, XML, EMF, MOF...) that allows to

(1) describe and check component metamodels CMM (with structural and behavioural features, with a model that links to Java code)

(2) describe and check component models CM

(3) provide an API to navigate on and query models, to add operations and processing on models

(4) ...

LCI should maintain this (CMM-CM) layer since it relates to metamodels.

At first sight OCLE can provide the main elements on points 1 and 2 but it doesn't provide an API usable in process A (structure) and B (behaviour).

Other tools exist that can help to use Ecore without handling it directly:

- Kermeta (IRISA) `http://www.kermeta.org/`
- ATL (LINA) `http://www.eclipse.org/m2m/atl/`
- ArgoUML tool (OpenSource) `http://argouml.tigris.org/`
- others...

Information on this aspect can be found here:

- Generalities
  `http://en.wikipedia.org/wiki/Model-driven_architecture`
  `http://en.wikipedia.org/wiki/Model_Transformation_Language`
- Eclipse Modeling Tools
  `http://www.eclipse.org/modeling/`
- Kermeta (IRISA)
  `http://www.kermeta.org/`
- ATL (LINA)
  `http://www.eclipse.org/m2m/atl/`
- Tools
  `http://planet-mde.org/index.php?option=com_xcombuilder&cat=Tool&Itemid=47`

It would be helpful to compare tools

B.2 Java Annotations

In this appendix section we ould provide the Java definition of the annotations.

See section 4.3.1.

List of Figures