ECONET Project
PRAGUE 2007 - WORKSHOP REPORT

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

2007, 3-7 September

supported by

[1]LINA - FRE CNRS 2729- 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 Prague 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 Department of Software Engineering of Charles University in Prague. This workshop was the first one of the ECONET Project Nr 16293RG entitled, "Behaviour Abstraction from Code, Filling the Gap between Component Specification and Implementation".

The workshop was convened in response to the objectives of the first year of the project plan which are reminded here

- Determination of the field of application (boundaries of Java concepts and idioms).
- Settings of the major principles to abstract behaviours for software components (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 Reverse Engineering (behaviuor abstraction from code).
- Integration of the verification of guards using Object Constraint Language OCL (and the OCLE tool).
- Documentation, research report and workshop preparation.

More precisely, the aims of the workshop were (1) to improve the knowledge of the participants on each other activity and background, and (2) to take concerted decisions on the project issues (concrete objectives, tasks, organisation, responsabilities, deliveries, planning...).

On these points the workshop was a satisfactory step in the project, thanks to the motivation of the participants. The following issues have been discussed: detailed competences of each participant on the project topics, comparison of component models and development approaches, concrete objectives of the project, selection of the kind of code to be abstracted, shared techniques and tools, common benchmark, etc. The working sessions enabled (1) to establish a common and shared concrete vision of the project, (2) to delimit the project objectives and context (nature of the Java code, benchmark, subset of the abstract models), (3) to divide the project into parts, which are easier to handle, (4) the definition and repartition of individual tasks.

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:

- Structural abstraction from Java code.
- Behavioural abstraction from Java code.
- Metamodel definition and consistency verification.

Each part (subproject) constitutes a challenge since there is ongoing research on it. Interfaces between the parts have been roughly defined. A common benchmark has been proposed to avoid support mismatch.

A key recommendation from the workshop is that the participants should tackle the objectives in a limited (but extensible) context in order to produce results in a 2-year project. Nevertheless a contribution on the related research fields is expected. A secondary recommendation from the workshop is to pursue a cross fertilisation on the abstract component models and metamodels.

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

This report relates what happened in the Prague's workshop (2007).

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.

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, PP07a]. 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, AAA06, 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.

1.2 The Workshop at Charles University of Prague

The workshop is a major milestone for the first year of the project. This section presents the project evolution until the workshop was held.

1.2.1 Preparation

Since the project has been accepted by Egide in march 2007, the exchanges between the partners became more frequent and precise at this date. Some of the exchanges are stored in the Project Wiki.

This wiki was installed at LINA (University of Nantes) in april 2007. It includes discussions, a repository for project and workshop material, etc.

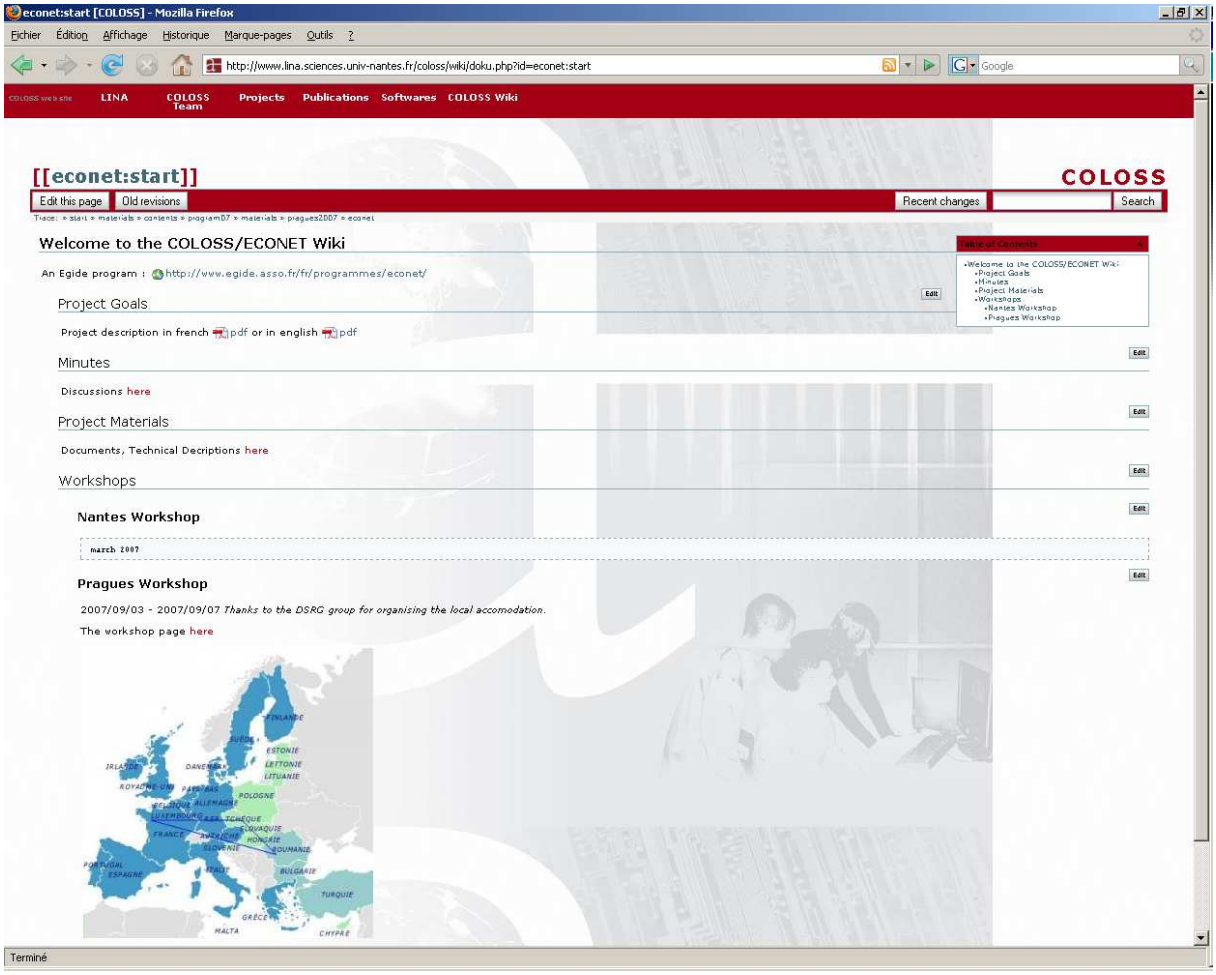


Figure 1.1: Project Wiki

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

The workshop was initially planned on the end of June, possibly joinly with the TOOLS conference. This was not possible and it has been delayed to the first week of september.

A special group of pages have been written for the Workshop (figure 1.2). The URL address is:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007

1.2.2 Organisation

The workshop organisation was handled by Pascal André and Ondrej Sery. The local organization committee included Ondrej Sery, Frantisek Plasil, Petr Hnetynka and Jan Kofron.

Detailed information is given on the wiki site (figure 1.3).

http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:organization

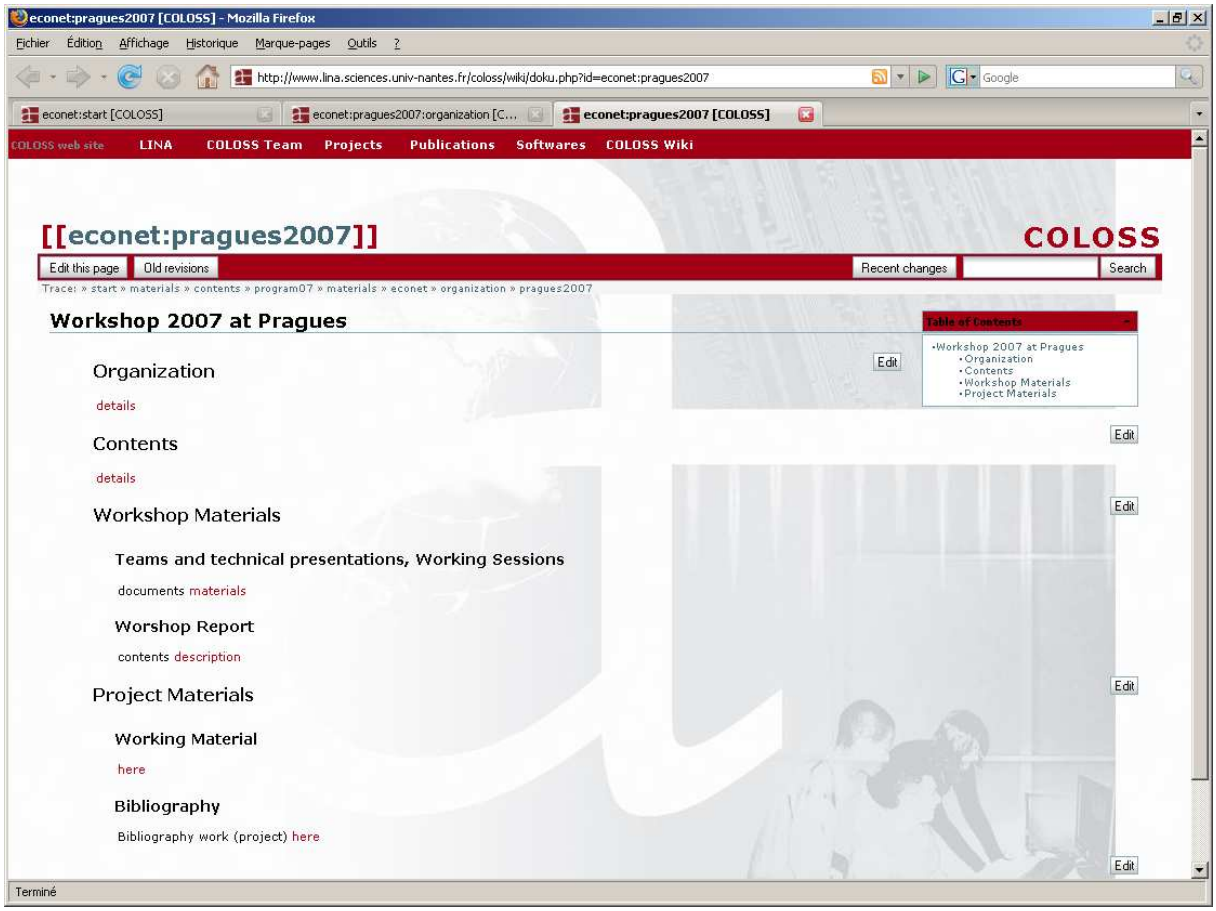


Figure 1.2: Workshop on the Wiki

1.2.3 Objectives

The following 'Workshop Objectives and Delivery' statement was a first throw and kept many issues open.

- Objectives ⇝ State of the art + clear application context
  - build a reference bibliography of the reverse engineering domain
    concepts, related work and comparison, mains issues, approaches, plateforms and tools (JPF, Bandera,...)
  - set the source area
    subset of Java concepts, libraries, components, examples
  - set the target area(s)
    SOFA, Kmelia, Vercors, ... - contracts, protocols, inheritance...
  - initiate some directions to follow in order to process the reverse transformation
    patterns, rule based system, combination of several existing tools
- Delivery
  A report for the project first year evaluation + plan the second year with individual objectives

1.2.4 Participants

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

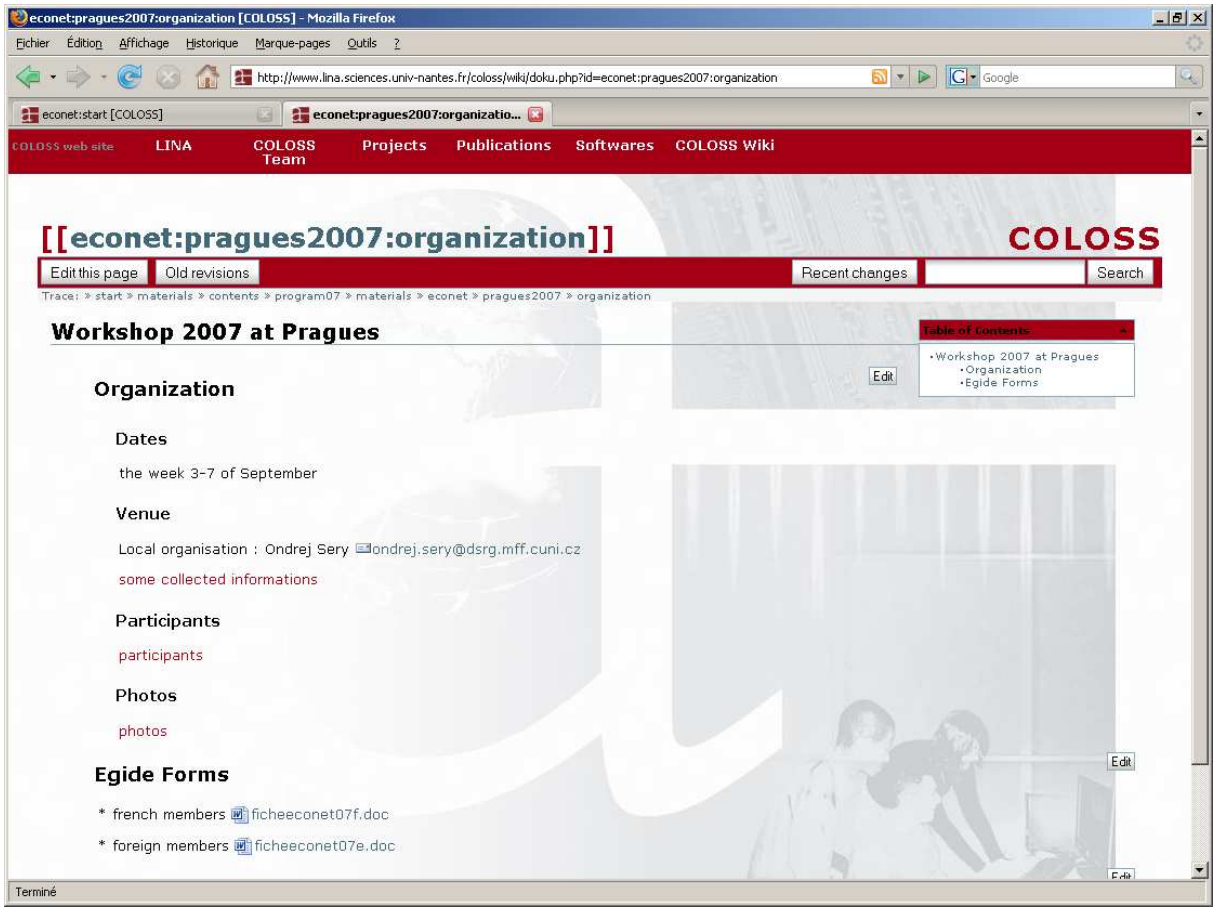


Figure 1.3: Workshop Organisation on the Wiki

- *Dan CHIOREAN* - LCI
- *Dragos PETRASCU* - LCI
- *František PLÁŠIL* - DSRG
- *Gilles ARDOUREL* - COLOSS
- *Jacques NOYE* - OBASCO (excused)
- *Jan KOFROŇ* - DSRG
- *Jean-Claude ROYER* - OBASCO
- *Jiři ADÁMEK* - DSRG
- *Ondřej ŠERÝ* - DSRG
- *Pascal ANDRE* - COLOSS
- *Pavel PARÝZEK* - DSRG
- *Petr HNĚTYNKA* - DSRG
- *Tomáš POCH* - DSRG
- *Vladiela PETRASCU* - LCI

1.2.5 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 teams (recent work, projects, tools, ...)
  - Technical presentations
- Day 3, 4 and 5 are dedicated to the project work (context, goal, process, tools, practical organisation and responsabilities)

- Social events

More details are given on the Workshop Wiki at:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:program07

1.3 Report Contents

In the remaining of the report, we provide more informations on the presentation sessions (section 2.1 of chapter 2) and the working sessions (section 2.2 of chapter 2). In chapter 3 we present the project architecture which is the main result of the workshop.

Chapter 2

Workshop Sessions

This chapter collects the main events and informations of the workshop sessions. We first begin by the presentation sessions where the participants present themselves, their team and technical contributions (section 2.1). Then we summarise in section 2.2 the contributions of the working sessions where the participants discussed on the project (issues, structure, tasks, technical aspects, tools...).

The detailed program is given on the wiki at:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:program07

The slides, pictures and discussions are stored on the wiki at:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:materials

2.1 Team and Technical Presentation Sessions

2.1.1 Introduction

During the development process of an information system several models are produced (manually or automatically). A main objective of the ECONET partners are to contribute to improve both the models and the production. In particular, all partners are concerned with the verification of models and programs.

To simplify, let consider at least two levels for the models produced during the development process: an abstract model and a concrete model written in a programming language *i.e.* in Java. From the verification point of view, the goal is to assert properties (by proof or by model checking) on both the abstract and the concrete models. One way (the formal method approach) is to assert properties on the abstract model, then to refine it into a concrete model and prove the refinement. Another way is to assert properties on both levels.

In this workshop we are mainly interested in the model checking approach for property verification. Model checking is the process of checking whether a given model satisfies a given property (*e.g.* a logical formula). The concept is general and applies to all kinds of logics and their models. A simple model checking problem is to test whether a given formula in the propositional logic is satisfied by a given model. In case of properties to be checked, the most common way to express them is using a temporal logic (LTL, CTL) and in the form of assertions. However, it is also possible to check for a predefined set of properties - deadlocks or properties specific to a certain class of systems such as device drivers.

One can admit that model checking at the abstract level is well-known and there are techniques and tools for model checking abstract models. But the model checking of concrete models is still to explore. Model checking of software is a popular research topic nowadays, mainly because there are several issues that have to be solved before the technique can be used for real-life applications.

How can we contribute? We can check the code itself with appropriate tool. This is not convenient because the tools are not mature in this domain and the concepts at the concrete level are quite different from those of the abstract level, the informations are numerous and more detailed, model checking quickly faces the problem of state explosion. Another approach is to filter the informations:

- by selecting informations of the code model checker (as explained in the presentation of Pavel Parisek)

- by abstracting informations from the code , this is the main issue of our Econet Project: extract abstraction *i.e.* in order to apply model checking at the abstract level. Two cases are possible whether an abstract model exists or not.

The talks given during the presentation sessions addressed contributions on the above topics. Many of them present an abstract model and model checking techniques (Petr Hnetynka, Jan Kofron, Jean-Claude Royer, Pascal André). The talk of Pavel Parizek is more precisely about the problem of model checking concrete models against abstract ones). The talk of Dan Chiorean was more general because it works on models and metamodels, moreover it assumes a quite different view of model checking, that is the verification of consistency and completion of models.

Note that we summarize the main issues of the team presentation and forget among others, history and members details.

2.1.2 Monday, September 3, 2007

Time	Title	Speaker
10:30	Welcome and Program	Pascal André
|	Participant presentation	each participant
|	Local Organisation	Ondrej Sery, Petr Hnetynka
12:00	DSRG Team presentation and position	Frantisek Plasil
13:30	SOFA overview	Petr Hnetynka
|	(Extended) behaviour protocols + demo (from CoCoME contest)	Jan Kofron
17:00	Checking behaviour protocols against code using Java PathFinder	Pavel Parizek

DSRG

Frantisek Plasil presented the Distributed Systems Research Group, one of the research groups of the Department of Software Engineering of Charles University. Its areas of interest are:

- Software Performance (Regression benchmarking),
- Software Components (SOFA, Fractal)
  - Architecture/Component models,
  - Design (Use cases, behaviour specification of components - Model checking),
  - Deployment (Connectors - Addressing environment heterogeneity)

DSRG focuses on research in distributed systems, particularly the construction of component and service middleware and its software engineering aspects - system architecture, formal definition and verification of behavioural properties, performance evaluation. This focus is reflected in the long running SOFA project, which deals with a distributed component model that provides advanced features such as formal verification of component properties, and which is provided as an open source platform. Other major projects that DSRG recently participated in include the ITEA PEPITA & OSMOSE & OSIRIS projects and industrial projects with partners such as France Telecom, Borland and Iona.

The areas of contribution to ECONET are in general a cross fertilization and joint publications. More specifically, they are providing SOFA, enhancing EBP by additional features from STS/eLTS (to enhance expressive power of EBP), extracting protocols (EBP) from code (at least BP from code), defining well-formed rules for SOFA components, providing a benchmark (the CoCoME example).

SOFA overview

As chief architect, Petr Hnetynka presented an overview of SOFA 2. SOFA 2 is the new version of the SOFA component system. It focus on removing limitations of the current SOFA implementation (inconsistencies between implementation and specifications e.g., protocols vs. connectors, architectures vs. dynamic reconfiguration, ...), clear design and properly balanced support of advanced features. These are mainly:

- model-driven design

- hierarchical architectures
- support for dynamic architectures
- support for multiple communication styles
- formal modeling of both functional and non-functional parts of components
- transparent distribution
- behaviour validation

It is implemented in Java (as well as the previous SOFA). The implementation is freely available (LGPL license)[1].

The presentation included detailed informations on

- Component model
  - metamodel
  - dynamic reconfiguration (dynamic architectures)
  - connectors
  - control parts (non-functional)
  - versioning
  - behaviour specification
- Implementation
  - component lifecycle
  - runtime environment
  - usage, tools, current status

Questions

About consistency (Metamodel well-formedness rule) the project organization in two parts on the group (components architecture / behaviour specifications) lead to some independent (and not compatible) evolutions. Well-formedness rule are informal except for protocols.

The micro-components have a similar concept in Fractal (monitoring methods and repository).

The 'aspects' are not really those of Aspect Oriented Programming.

About the link between specification and code, non-SOFA components can be encapsulated in wrappers, connectors can be RMI, Corba, the implementation is java specific with SOFA knowledge and annotations (future).

Communication style in connectors can be any communication style (method invocation, shared memory, ...).

Modeling and Verifying behaviour of Software Components in SOFA 2

Jan Kofron presented the (original) behaviour Protocols of SOFA and the extensions of behaviour Protocols (EBP) of SOFA 2. Each part was organised as follow (presentation, description, verification, experience, demo).

The problem of behaviour verification is undecidable in general. There are two ways to face it: (1) To use behaviour description languages which describe behaviour of the software precisely and to put up with the fact that the tools will never stop for some inputs (behaviour descriptions). (2) To use behaviour description languages, which are not expressive enough to describe behaviour of software precisely, but the verification of the specifications is decidable. We have chosen the second approach. Therefore, a behaviour protocol should be seen rather as an approximation of a component's behaviour. The most important benefit of this approach is the existence of a fully automatic behaviour verification procedure (implemented in our behaviour protocol checker).

The purpose of behaviour protocols is to specify the behaviour of software components, so that interesting properties of their behaviour can be verified (not only the operation signatures) using model checking techniques for example. Model checking is one of the approaches to formal verification of finite state hardware and software systems. A model checker usually accepts a finite model of a target system and a property expressed in some

[1] http://sofa.objectweb.org/

property specification language, and checks whether the model satisfies the property via traversal of the state space that is generated from the model.

behaviours are an abstraction of the component interactions and can be represented by LTS, use cases, sequence diagrams, process algebra expressions, etc. In SOFA, behaviours are described by processes (regular expressions and communication primitives as in process algebra). Checking protocol compliance (both horizontally and vertically) is similar to checking component behaviour compatibility. Compliance means: absence of communication errors (bad activity, no activity, infinite activity) which can be found automatically and verified separately for particular levels of nesting (hierarchy).

In SOFA two checkers were implemented: the behaviour Protocol Checker (BPC) (proprietary explicit state model checker for BP, written in Java, uses Parse Tree automata for state space generation, able to verify state spaces of the order 10^7 states, may run several days) and the dChecker (again proprietary tool, distributed state space traversal, significantly faster than BPC, state spaces of the order 10^7 for each computer i.e., entire state space of the order of 10^8 state).
Several flaws of BP were identified during the specification: lack of synchronization mechanisms (impossible to synchronize more than two components), lack of expressiveness (absence of macros caused parts repetition in the specification which make it hard to fix the errors, absence of variables caused overspecification, absence of a way to express common patterns, e.g. until loops, caused unreadable specification).

This led to extensions of BP on data (method parameters and local variables can be of enumerated types), synchronization (special events: joining events for synchronization of more than two components) and until loops (a syntactic abbreviation to enhance the readability). Performance issues were also a motivation, the new solution is a transformation of EBP into Promela, the input language of the Spin model checker. The performance was comparable on the same benchmak (CoCoME).

Questions

Guards are possible, they are branches on local state.

A visual representation of the protocols is possible via parse trees. The component picture is manually done with Visio in a UML flavor.

Local checks are performed. Promela checkes both horizontal and vertical compliance. One to many is in the metamodel.

About feedfack, there are solutions (...).

Checking behaviour protocols against code using Java PathFinder

Pavel Parizek addresses a topic that is close to the project concerns[2]: verification of conformance between behaviour specification and code. The work approaches the general problem of model checking programs and especially Java code. Remind that in the previous presentation, Jan Kofron talked about (abstract) model checking. The problem addressed here is to do it at the implementation level.

A general problem of model checking is the necessity to create a model of the system to be checked. Manual construction of the model is an error-prone process, and even if the model is automatically extracted from a specification of the system or from the source code, it is an abstraction - therefore, a model checker may find errors in the model that are not present in the original program and vice versa. A solution is to use a model checker that does not need to have a model, but works directly with the implementation of a target system. There are several difficulties encountered with such an approach: code model checker is needed, the models and properties handled are low-level, there is a problem of state explosion since an abstract state collapse many concrete states, partial models are to check and not only full programs. The talk of Pavel deals with these difficulties.

As to software model checking at the program source code level, a crucial problem is the size of state space triggered by the model of a program (i.e. the problem of state explosion). Despite that, there exist such model checkers. For Java programs, these are most notably the Java PathFinder (JPF) and Bandera tools. (An advantage of JPF over Bandera is that the most recent release of the latter is an alpha version, not being fully stable yet, and that JPF is also more extensible). The properties checked are either predefined (e.g. absence of a deadlock) or specified in LTL (Bandera) and via assertions related to the code (JPF). A typical feature of both Bandera and JPF is the combination of static program analysis and model checking. The former is used to create a program model; to decrease the state space size, abstraction techniques are applied - these include partial order reduction and data

[2] See the remarks in the introduction section

abstraction. State explosion can be also mitigated by the decomposition of a software system into small and well-defined units, components. Typically, a software component generates smaller state space than the whole system and therefore can be checked with lesser requirements on both space and time. Nevertheless, model checking of code of software components usually brings along the problem of missing environment, which means that it is not possible to model check an isolated component, because it does not form a complete program with an explicit starting point (e.g. the main method). In order to solve this obstacle, it is necessary to create a model of the environment of the component subject to model checking, including the specification of possible values of method parameters, and then check the whole program, composed of the environment and component. A specific feature of software components is the existence of ADLs (Architecture Description Languages) used to specify component interfaces, and first of all composition of components via bindings of their interfaces (i.e to specify the architecture of a component-based application at a higher level of abstraction than code). Some ADLs even include the option to specify behaviour of the components, typically in a LTS-based formalism. An obvious challenge, not addressed yet to our knowledge, is to check the code of software components against a high-level behaviour specification provided at the ADL component specification level.

In this work, it is assumed that each primitive component (Java code) is compliant with its frame protocol (we thus have an abstract model). It has to accept/issue exactly those method call related event sequences on its frame that are specified by the frame protocol. The goal is to design an algorithm and a tool for checking compliance between Java implementation of a primitive component and its frame protocol via Java source code or byte code analysis.

The experimentation is done with JPF (Java PathFinder) a model checker for Java programs which is highly customizable and extensible ; it is able to check only low-level properties (such as deadlocks, uncaught exceptions) with non-deterministic value choice. It supports a `Publisher/Listener` pattern that can watch the course of the state space traversal and check for specific properties. Checking by JPF is not directly possible because JPF is able to check only low-level properties. So an extension of JPF is necessary to check the compliance between Java code and frame protocol. It is a high-level property: *a cooperation between JPF and the BP checker is defined with a backtracking policy*.

Moreover JPF accepts only complete Java programs: *an automated generation of component environment is proposed to solve the problem of missing environments* (for components). The model should force the environment to call a certain method of a particular provided interface at the moment the component expects it and to accept a certain method call issued on a particular required interface at the moment the component "wishes" to do so. Two options are possible:

- *Inverted frame protocol* which is constructed from the frame protocol by replacing all the accept events with emit events and vice versa. This option models the most general valid environment

- *Context protocol* which specifies actual use of the target component by the other components in the particular hierarchical architecture. This option models the simplest valid environment.

Context protocol is more suitable because the component application typically exploits a subset of functionality provided by the target component and completeness and feasibility are taken into account at the same time. But context protocol is time consuming and there are no Java construct for acceptance of a method call calling protocol.

Last, two techniques are proposed to reduction of environment's complexity (state explosion problem) : reduction of level of parallelism (static code analysis, concurrent metrics) and reduction of repetition.

Questions

You used the base BP model checking for this experimentation, did you investigate it in the context of EBP with Promela and SPIN. Not really, we should have the same problems probably.

Do you think JPF can be used for reverse engineering? Not exactly JPF provides execution abstractions through the traversal of execution flows on objects and classes. It can be connected to other tools via its listeners.

2.1.3 Tuesday, September 4, 2007

Time	Title	Speaker
09:00	Introduction	Pascal André
|	LCI Team presentation and position	Dan Chiorean
12:00	(UML) Model Checking - an OCL Based Approach	Dan Chiorean
13:30	OBASCO Team presentation and position	Jean-Claude Royer
|	Introduction Illustrating The STSLIB Project: Towards a Formal Component Model Based on STS	Jean-Claude Royer
|	COLOSS Team presentation and position	Gilles Ardourel
18:00	The Kmelia Component Model Hierarchical Service Description and Analysis	Pascal André

Welcome

Pascal André introduced the program of this second day of presentations including three team presentations and technical talks.

LCI

Dan Chiorean presented the Computer Science Research Laboratory (Laboratorul de Cercetare Ştiinţifică în Informatică - LCI). The Computer Science Research Laboratory was established in 1990, as an auxiliary department of the Mathematics and Computer Science Faculty of Babeş-Bolyai University (Universitatea Babeş-Bolyai - UBB) in Cluj-Napoca, Romania. The projects handled by the laboratory are included in contracts with the Ministry for Research and Technology and certain academic institutes, its main purpose being the promotion of research activity in a top field, that of object-oriented technology.

The main products designed and implemented in the lab so far were the mFOOPS environment, and the ROCASE and OCLE tools. The UBB-LCI team participated in the EU Research Project IST 1999-20017 NEPTUNE. The LCI team obtained recognised results in the area of model checking. Continuing the work begun in the NEPTUNE IST project, and using some of the results obtained in that framework, the LCI team designed and implemented OCLE, one of the most complete OCL tools existing today. The LCI group is very keen on continuing its work in the above-mentioned domains, with special interest in MOF based repositories, OCL support for all abstraction levels, use of OCL in transformation languages, OCL extension to support actions, improved code generation, suggestive use of OCL in Model Driven Development (MDD) applications.

In the context of this ECONET project, the UBB-LCI team brings its experience in designing and implementing a powerfull repository for the CoreComponent Metamodel. Namely, the LCI team will be involved in:

- specifying a complete set of Well Formedness Rules at the component metamodel level;
- specifying all the API metamodel's observers;
- generating the Java code corresponding both to assertions and Additional Operations;
- injecting this code in the repository code produced using the EMF framework, the OCLE tool and, possible, other state of the art tools;
- testing and refactoring the repository code.

(UML) Model Checking - an OCL Based Approach

Dan Chiorean exposed his point of view on model checking UML models. This research area is taken in the context of the latest modeling approaches (MDA, MDE, LDD, DSL), which are characterized by the use of metamodeling and the use of modeling languages which are more specialized compared with UML. The OMG vision promotes a uniform approach, all modeling languages being MOF instantiations. In this context, a robust and thoroughly tested MOF model would be in the benefit of all modeling languages. Specifying a precise syntax and semantics requires usage of rules. Taking into account that the standard formalism used in specifying rules is OCL, an appropriate support for the constraint language is needed. (U)ML uses different formalisms for specifying architectural and behavioural information, therefore checking (U)ML models is quite complex. The results and conclusions obtained in UML model checking can be used in checking models specified using other MOF based modeling languages.

The objectives of (UML) model checking are: to ensure that the model conforms to different kinds of rules ((UML) Well Formedness Rules - WFR, Profile Rules - PR, Business Model Rules - BMR, Methodological Rules - MR, Behavioral Constraints Rules - BCR), to use accepted and easy to understand standards in the modelers community, to validate the standard specification on real models. The following results are expected: realised applications comply with requirements, applications become more reliable, the time needed for developing them is diminished, the application costs decrease, reuse is promoted to all model element levels (classes, components, patterns, frameworks).

In the context of the new modeling approaches, model compilation has to become a mandatory requirement (like source code compilation). In order to accomplish these objectives and to benefit from the expected results, an incremental and iterative approach is proposed: rule specification, rule validation, identification of reasons for rule failures, model updating - applied to different modeling layers.

Then, Dan described some features of UML models related to OCL and verification rules, and some examples of rules from each of the above mentioned categories (WFR, PR, BMR, MR, BCR). UML model checking concerns the following properties: model completeness, model consistency, model correctness and model architecture accuracy. Model checking related activities must be supported by appropriate CASE tools. Some questions are: What could be done in order to improve the state of facts? or, more specifically, Is the needed information disposable? Is the textual formalism appropriate?

As UML is grounded on different formalisms - OCL, STD (ASM), graph theory (Petri nets) - UML model checking requires different approaches. Dan discussed the following aspects: multi-view models, lack of a complete formal semantics for modeling languages and the management of inconsistencies (inherited cyclic composition inconsistency, dangling type reference, and connector specification missing).

The last part of the talk is dedicated to tool support.

First Dan sketched some UML/OCL tools (commercial and academic), stressing that working with medium or large models is currently difficult. The following drawbacks restrict a widespread use of tools that support OCL: the lack of coordination between UML services and OCL services, a weak support for interchanging UML models, the proprietary architecture used for repositories, the lack of OCL support at M2 level, the lack of support for reusing OCL specifications at M2 level, the weak support for updating UML models in an interactive manner. Compiling UML models before transforming them is not yet an usual activity. Unfortunately, only a small part of UML static semantics was tested. For some rules, the informal specification is ambiguous. Many specialists make general statements about the UML specification drawbacks, but they haven't tried either to identify the rationale of these drawbacks or to precisely mention the drawbacks. Surprisingly, misconceptions and incorrect statements about using OCL and about the language potential can be encountered in many papers.

Next, Dan presented the OCLE tool, developed at LCI, and made a demonstration of its functionalities. What distinguishes OCLE from other similar tools is: it allows simultaneous access at both M2 and M1 level, it implements OCL 2.0 specifications (ensuring compatibility with former OCL versions), it offers extended support for the "def, let" mechanism, it supports simultaneous multiple views of the same information, it aids users in correcting the errors identified by evaluating the rules specified at the M1/M2 level, it allows semantic checking of XML documents, it ensures MDA support both by defining and checking profile rules, it proposes a friendly GUI, it promotes reuse by means of OCLE projects, and it permits compiling and evaluating OCL specifications spread in different files. In order to better accomplish the requirements related to the intuitiveness, rigor, usefulness, the following functionalities were implemented in OCLE: transitive closure operation, neutral printing operations, explicit context specification, flexible specification for operations without parameters and the compatibility with the syntax of previous OCL versions. Some benefits are: the possibility to specify simpler OCL expressions, extended possibilities for debugging OCL specifications, and support for reusing earlier specifications without any changes.

Checking UML models against Well Formedness Rules using an appropriate UML/OCL tool is the natural way to check UML models and to correct various errors. Many bugs found in the UML static semantic specification v1.5 were fixed and the majority of the rules were tested. Using the OCLE code generator, the Java code implementing the additional operations behaviour was generated completely and used in the OCLE repository. The OCL 2.0 specification was extended with a set of practical features. The solution we proposed for checking XML documents overcomes the drawbacks that all the "classical" solutions carry.

Remark

The definition of model checking is quite different from the one in SOFA, KADL, and Kmelia approaches. The main reason is that the vision considered by OCLE takes into account UML model's conformance to WFR. In fact, this represents the first check that all UML models have to pass.

Questions

A first question was about components in UML and verification tools. An approach is that described in Cheesman and Daniels' book *UML Components*, published by Addison Wesley in 2001.

Another question was about a formal semantics of OCL 2.0. Dan explained that beginning with the OCL 2.0 version this is described using the OCL 2.0 metamodel, included in the standard.

OCLE (and ROCASE) have been used for both education and industry.

Other questions are summarised in a draft version extracted from PA notes.

- Completeness and consistency of WFR: this is a difficult problem.
- Extension of OCL to some reduced kind of Action Semantics to settle Functional Computation descriptions: it can be done since there exists in fact a family of OCL languages.
- Standard evolutions (what happens when a new version is adopted): the metamodel is downloaded, the crucial point is the MOF description.
- Import-export facilities by XMI files and visual representations by DI (Diagram Interchange): tools do not fully comply XMI, and DI is not adopted yet. UML models differ from MOF models. OCLE is not a standalone tool but I have to show and navigate on class diagram, snapshots, use case
- Different problems : non-standard visual notations first tool Rocase (C++), OCLE code generation in MDE, java code generation (model, assertions) parts of the repository have been produced by the tool.
- Java profile for code generation. We use our own policy - run the rules.
- Implementation aspects: the JGraph library was used for the graphical editor, and the Velocity template was used in order to generate the Java code. OCLE XMI parsers were entirely implemented at LCI.
- A big problem concerns the tool's maintenance: the people involved in realizing OCLE left the laboratory due to the lack of financial resources.

OBASCO

Jean-Claude Royer presented the OBjects, ASpects, and COmponents (OBASCO) team. OBASCO is a joint research group of (the CS Departement of) Ecole des Mines Nantes and INRIA's research center in Rennes, IRISA (see also OBASCO's home page at IRISA). OBASCO is also a team of Laboratoire Informatique de Nantes Atlantique (LINA, FRE CNRS 2729). The LINA laboratory is mainly a cooperation between Ecole des Mines de Nantes and the University of Nantes. It is specialized on two axes : distributed software architectures and computer-aided decision systems.

The OBASCO (Objects, Aspects and Components) research group addresses the general problem of adapting software to its uses by developing tools for building software architectures based on components and aspects. We are (re)using techniques developed in the programming languages domain, and more precisely in object-oriented programming.

The objectives are to solve scalability problems in software engineering and to improve software architectures adaptation. Two main directions followed are the separation of concern (specific programs for specific problems) and correct composition of existing programming artefacts.

The research domain covers topics of the Software Engineering:

- Software components and scalability
- Programming languages
- Post object-oriented programming
- Generative programming
  - Sequential, concurrent and distributed
  - Mechanism for separation and composition

  - Objects versus aspects versus components
- Model driven engineering: transformation techniques

according to three swim lanes

- Aspect-oriented programming
  - To explicit links between metaobject and aspect
  - To formalize aspect-oriented models
  - To design and implement a language
  - Reverse engineering of legacy code with aspects
- Software component
  - Explicit protocols for components
  - Property verification for components and architectures
  - Understand relations between aspects and components
- Domain specific language
  - Domain specific language
  - Expressiveness, extensibility and compilation
  - Aspect languages, composition and DSL

OBASCO brings its experience to the ECONET project with an abstract model based on algebraic specification and LTS, and its derivation to Java code (engineering process while in ECONET we work on the reverse-engineering process). A first work was to verify some properties in component systems with data, we have some results here, which make it possible to abstract component protocols with data. We have also shown how to generate Java code from protocol descriptions, such that components can communicate according to their declared behaviour. Finally, we have investigated aspect languages for the modularization of crosscutting concerns defined in terms and through modification of protocols. A question, which we did not study, was the compatibility between some Java code and a protocol. OBASCO is also interested in extending the SOFA approach to introduce data, parameters and guards. Furthermore, we intend to study the integration of these techniques with our on-going effort on the static analysis of properties of protocol-modifying aspects.

Introduction Illustrating The STSLIB Project: Towards a Formal Component Model Based on STS

Jean-Claude Royer presented an approach that covered the whole development models from abstract description to Java implementation.

Component-based software engineering is becoming an important approach for system development. It is assumed here that explicit protocols are integrated into component interfaces to describe their behaviour in a formal way. In this case, explicit protocols are often dissociated from component code, it is not ensured that component execution will respect protocols rules. A crucial issue is to fill the gap between high-level models, needed for design and verification, and implementation.

This talk introduces first a component model with explicit protocols based on symbolic transition systems. Some related models are quickly overviewed: Java/A, CADP, behaviour Protocol models (SOFA, PROCOL, CO-OPN), FSP java, JCSP.

The model is presented through an example: The Cash-Point Case Study. The main concept is STS, for Symbolic Transition System, that is a dynamic description coupled with a data type description. A Labelled Transition System or LTS is an automaton with simple labels, states are sometimes called control states in this document. STS add to LTS full data types, guards and input/output values. A transition is defined by `[guard] event communication / action`. The communication can be an emission (simple or multiple) or a receipt. The data types describe the operation semantics of the emitters, the guards, the generators and the actions. An algebraic style with positive conditional axioms is used to described the data types.

An architecture is almost a UML communication diagram rather than a UML component diagram: a link between ports denote event synchronization with communications. This was rather the KADL model which is dedicated to

analysis and verification purposes: the communication glue is simplified which is very expressive in KADL and some other features are not taken into account: no inheritance between STS, ADT description are less general, no component parameterization.

The cash point example is illustrated on two points: defining a STS, defining the architecture. The existing elements or experimentations for tool support are explained.

- Defining an STS
  One first implementation was implemented in Python (it has been rewriting in Java 1.5 under Eclipse) for the STS: the user writes a `.sts` file (a parser exists for the dynamic part), an interface generator builds an ADT skeleton `.adt` (the user has to fill the axioms), a Java translator generates a Java class from the ADT (experimental) but the user may also write a Java class or reuse an existing one.

- Defining an architecture
  A grammar allow to express component types, local instances, bindings and exports. This is a bit more complex since the parser and the loader have to be recursive. It also needs to call the STS parser and loader. When loading an architecture we have also to specify where are the Java classes. The instantiation process is also a bit more complex. It is not completely stable since some choices about bindings are pending.

The talk pursue by the level devoted to verification. In the past, an approach using PVS (Theorem prover) has been experimented. The new approach rather close to model checking is based on the notion of Configuration Graph (or CFG) which is a LTS but with a data value associated to each control state. This allows formal analysis methods to analyze component and their interactions. Currently the verification process is to compute the synchronous product and check it or to compute the configuration graph and prove some properties. This tackle the difficult problem of state explosion when model checking with data.

Then a Java implementation for it is presented (runtime interpreter) that relies on a rendez-vous mechanism to synchronize events between component protocols. This talk showed how to get a correct implementation of a complex rendez-vous in presence of full data types, guarded transitions and, possibly, guarded receipts.

A tool support is made of a library with parsers and analysis tools STSLIB is a project devoted to the design of sophisticated concurrent systems, their verification and code generation. Currently, this is rather a Java API and the targeted runtime language is also Java. It may be used in the context of Eclipse or as a Java application.

Questions

One question was about the assistance for specification: only textual representations, the systematic usage of a 'self' parameter for data type operations, manual chain of specification processing. These points should be improved with the future (planned) GUI (Graphical User Interface) on Eclipse.

Another was about functional (for specification) and imperative (for implementation) styles for ADT. Only a restricted form of ADT is accepted.

A last question was about communication primitives and especially the multiple one. Only the multiple rendez-vous is implemented.

Time aspects were also discussed.

COLOSS

Gilles Ardourel presented the Relaible Components and Softwares (COLOSS - COmposants et LOgiciels SûrS) group, a team of the LINA (see the OBASCO presentation above).

The research activities of the COLOSS team range from fundamental aspects of software to applications. The main goal is the elaboration of concepts, methods and techniques supported by tools for software designers and developers. We explore formal approaches to assist software analysis and development:

- Multi-formalism specification and analysis of software systems,

- Specification, verification and validation of components and software architectures.
  The goal is to ensure correctness of components used and their composition in complex software systems.

The motivations (fundamental challenges) are correct the software construction, the software quality and safety and the support for specific development methods. The main research areas are

- Multi formalism specifications, multi-faceted analysis.
  The mono formalism approaches are limited by a partial covering of problem and a partial analysis. COLOSS studies the formal methods integration and faces some of its challenges (decomposition, semantic interoperability, formal analysis). Multi-platforms experiments use B, PVS, Spin, Grafcet, Petri nets...

- Design and verification of model properties.
  We investigate the use of formal methods and tools for modular modeling (component, objects) and property verification (system and model properties). In the CBSE context, the motivation is to provide models and practical tools to assist users in formal component-based development ; it covers the abstract definition of components and composition (simple, flexible and expressive), the verification of properties (safety, consistency, compatibility...) and the binding from components to code (refinement or code generation).
  In the UML context, the motivation is to improve the confidence in large and multifacet (diagrams) specifications where a formal semantics does not exist and where properties are quite complex *e.g.* model consistency is made of plenty (sub) properties. A single property can be decomposed into finer ones, can concern several groups of model elements, can be verified at different levels of completeness, and can be verified using several techniques with various costs and performances. We work on a generic verification process composite verification processes (with ordering, filtering, results propagation and annotation of faulty elements. . .) that supports the classification of verifications and properties (levels, diagrams...) and the abstraction of the results of different tools and formalisms.

COLOSS contributes to ECONET by providing Kmelia (an abstract component model) and its associated COSTO tool, its experience on formal methods and property verification, its experience on Java programming and a first experimentation on reverse-engineering Java to UML. COLOSS also expects fruitful exchanges with KADL and SOFA to gain mutual enrichments.

The Kmelia Component Model Hierarchical Service Description and Analysis

Pascal André presented the Kmelia Component Model and the Property Formal Verification towards the COmponent Study TOolbox (COSTO). The COSTO modules are being developed with JAVA. The modules have been integrated into the Eclipse IDE as plug-ins. Both the specification and analysis were illustrated on a Bank Automatic Teller Machine (ATM) case study and its withdrawal service.

The presentation begins by an overview of the model. The Kmelia model is a simple, formal and abstract component model based on the description of complex services.

- The simplicity relies on the few number of concepts that are used to describe the components and their assemblies. The main characteristics are: components, services, component assemblies, protocols, pre-post conditions, specification of complex interaction between services.

- The components are abstract, independent from execution environments and therefore non-executable. Kmelia can be used to model software architectures and their properties, these models being later refined to execution platforms. It can also be used as a common model for studying component or service model properties (abstraction, interoperability, composability).

- A formal model is defined for components and services, including syntax and type checking, pre-post conditions and LTS. This is the basis for any automated processing support.

- The services are first class elements and not only messages, which means that (1) services may be defined by a dynamic behaviour in addition to their signature and pre-post conditions and (2) services may be built from other services. Each service has an enhanced service interface which includes a service dependency composed of the provided services and the required services which are used in its context. The notion of service is central to Kmelia and a component interface describes mainly services. This means that the components are connected via their services (functional connections). The interaction model is therefore simple: required services are directly linked to provided services.

The model is extensible: it is possible to add new concepts from the kernel *e.g.* protocols, adaptors, aspects...

A component is a structuring unit that encapsulates a state and services; it has an interface with usage constraints. A component has an interface made of provided services and required services. A service is an abstraction of a functionality (signature, contract); it has a behaviour (dynamic evolution). The behaviour is defined by an extended LTS where transitions accept functional actions or communications (message send or receipt, service

call or return). A service also has an interface including provided services and required services that may lead to a hierarchical structuration. Assembly links are a simple support for component connection and interaction. Promotion links are a means for simplifying encapsulation.

The second part of the talk is dedicated to the verification of properties and its support with COSTO. The verification principles are: a formal verification of properties, a reduced user interaction (automation), customisable verification process, interconnection with appropriate tools (open framework). For this last point, the specific property analysis modules is realised by COSTO modules and the general property analysis are checked by connection with existing tools (model checking or theorem-proving). This is illustrated on the ATM example for the behavioural compatibility property. Its scope is the correctness of components assemblies and compositions, the availability of components and services, the compatibility of linked interfaces, the service compatibility and a diagnosis on mismatch. The service compatibility is defined at four levels of control: service signatures, enhanced service interfaces, contracts (pre/post conditions), and the behaviours (correct interactions between the caller service and the called service via the required service). An illustration with MEC an existing powerful model-checker for synchronised LTS. MEC has a compatible definition of STS that allows a systematic translation into input formalisms and feedback. A verification is done for each triple of services based on a first level assembly link. In the ATM example, a deadlock is detected and visualized.

Questions

Questions were asked all along this talk on the hierarchical aspects, the communication primitives, the behaviour specification, the external tools used, the link with B specifications, etc.

2.2 Working Session

2.2.1 Introduction

The goals of the working sessions are mainly to fix a roadmapfor the project. This means to clarify and delimit 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...). One first issue was to check that each partner has a compatible (or even better the same) vision of the project. Another one is to structure the work in several parts because a monolithic task cannot be realised by four geographically separated entities.

These sessions are highly dynamic so that a strict program was not established. Thus the detailed program given in the following subsections are post-workshop programs. Nevertheless, the initial program covering included three aspects for the working session:

- decision on the source and target area boundaries,
- discussions on the way to get reverse transformations,
- find application examples.

In the following we will summarize the session contents day by day.

2.2.2 Wednesday, September 5, 2007

This is a half-day session.

Time	Activity	Speaker
09:00	Introduction to the working sessions	Pascal André
|	LCI Point of view	Dan Chiorean
|	DSRG Point of view	Frantisek Plasil
12:00	Round tables and discussions	all

Introduction to the working sessions

The slides of this (first plan) introduction are provided on the Workshop Wiki at:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:program07

Pascal introduced the working session by giving its view of the current project situation and provided some tracks for the discussions. Consequently the Working Session Roadmap was organised around four points

1. Convergence on the objectives.
2. Convergence on the means.
3. Definition of the tasks.
4. Production

The three first points define an initial objective for each day of the working session.

1- Convergence on the objectives

There is a clear agreement on the "abstract" context:

1. We have abstract component models (SOFA, Kmelia, KADL) and tools. Some have elements for Java code generation (engineering).
2. We assume Java Code Specifications and Programs. This code contains informations about structure and behaviours (even if it was not designed in a CBSE).
3. We want to practice Reverse Engineering = from code to abstract models. There exist general techniques and tools such as code analysers, patterns, extractors...

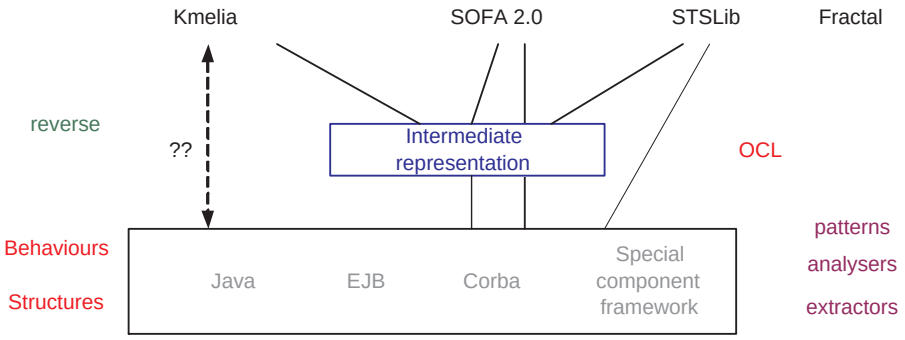


Figure 2.1: ECONET Project: "abstract" context

There is a fuzzy vision of the "concrete" context.

- What sort of Java code nature do we want to process?
  - Bytecode - the one that exists at run-time.
  - Plain source - the one usable for an open source project.
  - Annotated Source - a Java program (source or compiled) that includes informations usable for connecting to an abstract model. This is the case when the program is (partially) generated from a component abstract specification by some code generator.
- Assuming we have a Java program, to what extend should the code be structured?
  - plain Java: there is no special structure.

  - "componentised" Java (EJB, Corba, .NET, issued from a code generator - SOFA, Fractal...): the code has a component flavour.
  - "behavioural" Java (threads, communication primitives, issued from a code generator...): the code has a behaviour flavour.

- What are the reengineering issues? It means toward what objective do we look for abstractions?
  - legacy code recovery/discovery: the usual goal of reverse-engineering. You have an existing code but it was not designed according to the same abstract concepts.
  - compare code and specifications (conformance): we want to check whether a concrete representation and an abstract representation (of the same system!) are consistent.
  - roundtrip: the code is generated from a model and complete that model. The development process includes engineering and reverse-engineering activities. This approach is frequent in MDA.
  - ...

Goal of day 3 = Clear agreement on the "concrete" context

2- Convergence on the means

Once we agree on the right objectives, the question is how can we reach them.

- Collaborative State of the Art Study.
  The idea here is to share the individual experiences both on domains and tools in order to accelerate the collaborative work. The state of the art covers works and tools related to: reverse engineering (in general, for components), Java reverse engineering, Java code manipulation (analysis, annotation, extraction...).
- Re-engineering techniques.
  The objective is to be up-to-date in techniques available in the open source community.
  - Java Compilers and Analysers to handle the Java code.
  - Patterns, rule based systems for reverse-engineering issues.
  - Used notations and Intermediate layers (models) for a step-by-step process.
  - ...
- Separate modules (e.g. structural / behavioural / metamodels). Do we work on separate parts and levels with different applications or shall we use one program only? Do we propose a framework for general purpose or do we design a single purpose application?
- Benchmark example.
  It is quite difficult to find Java code related to components. A collaborative work implies that we work together on the same Java program.

We also have experience on OCL and metamodels.
(optimistic) Goal of day 4 = organise the means tracks and find the benchmark

3- Definition of the tasks

Once we agree on the right objectives and we have a somewhat clear idea of the means we provide, the question is to organise the work *i.e.* the project management.

- What to do?
  Define a list of tasks to do and the delivery.
- Contributions?
  Define the actors and collaborations (who do what).

- Synchronisation points
  Point out the bottlenecks of the process.

- Planning
  Define who do what and when.

- ...

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

4- Production

Here are some guidelines for that part.

- Workshop Report
  - Collect paper and slides
  - Summary of the discussions

+ Bibliographical Notes

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

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

These are also part of the initial 'Second year objectives'

LCI Point of view

Dan talked about various aspects of reverse engineering and especially those related to models (MDA). Reverse engineering is about finding the concepts of the models by analysing the code and inject the model. There are several models: the programming language (Java) model, the component models. He stressed three aspects:

1. About reverse engineering (RE).
   He expects a more clear definition of the context and goals. The domain is quite difficult and unexplored. There are different programming languages (Eiffel, C++, Java...) and different tools. There are some existing work on the subject, we should have a look at them. He worried about the practical result of Component RE (the value of RE, the mandatory results). Do SOFA applications have non-SOFA parts.

2. About Behaviour Checking.
   Do we want to detect possible deadlocks in the code? What can we do if component instances relate to the same component type.

3. About pre-post conditions.
   This information is rarely present with a clear format in programs (even for Java assertions).

Dan also noticed that most tools work with the architecture of the application. He also remind that LCI had a previous experience with reverse-engineering UML from C++ programs.

DSRG Point of view

The slides of this intervention are provided on the Workshop Wiki at:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:program07

Frantisek recalls that the ECONET project is a small project (budget, duration). He commented the initial project definition (see section 1.1). As a general remark he recalled that the pragmatic outputs of the project are a cross fertilization and joint publications.

He distinguished four area of contribution by DSRG:

1. Providing SOFA.
   This is one of the abstract models. Providing it covers the language definition, the tool support, examples, the knowledge..
2. Enhancing EBP by additional features from STS/eLTS (to enhance expressive power of EBP). The idea is to take concepts from other abstract models..
3. Extracting protocols (EBP) from code (at least BP from code). This is a part of the reverse engineering if we consider that there are two complementary aspects in a program: the structure and the behaviour..
4. OCL: WFR (well-formed rules) for component diagrams, activity diagrams? WFR for SOFA components, The idea is to improve the SOFA language definition using the LCI experience..

Note that point 2 and 4 are indirectly related to reverse-engineering but can participate to define a more general framework.

Frantisek brings an answer to one of the *means* issues: the benchmark. He proposes to take an exemple from a previous work on a Dagstuhl Modelling Contest, the CoCoME (Common Component Modelling Example) http://www.cocome.org/. The advantage is to benefit from a Java program that implements a component application (this is rare enough to have both a specification and a program) and two abstract models that have been manually built on it. Two trivial examples are more fruitful.

Discussions

The discussions took place from the three above interventions.

Here are some Gilles's random ideas:

1. We are interested in behavior models.
   (a) Getting them from plain Java code without structural information about the component architecture is far beyond the scope of this project.
   (b) We at least need some kind of "boundaries".
2. There are two first steps :
   (a) Our benchmarks shouldn't be spaghetti code (while not sticking to a particular component model, the application should have something that look like a component architecture). A proposed task is finding fitting benchmarks.
   (b) Then let's pretend we already got most of the structural info about the component architecture (manually, automatically from specification or code...).
3. We are about to target different abstract models.
   (a) We need meta information somehow common to the models : a minimal structural component model (component hierarchy, one or several interfaces by component). A proposed task is finding this minimal meta information. We don't need a Unified Component Language -just the minimal stuff to work (remember the size of the project).
   (b) Additional model-specific meta information (because we might want to do something beyond this project's scope some day).

Annotations examples

```
code source part (class, package, method, whatever..)  "is part of componet id #12344"
code source part (class, package, method, whatever..)  "is part of interface id #12344"
```

Here is a short summary.

- Benchmark
  The CoCoME benchmark is adopted. It has many interest: existing informations on requirements, on component design, available Java program.

- Input
  Of course the fine input would be a component program but it does not exist yet. A better input is an object program with information (comments, annotations) on components. Again, it does not exist yet. We decided that the input is a plain source Java with possible annotations or information (comments, separate documentation files, UML models...). Having a source code allow to use parsers, extraction techniques, comments and various informations that are not available for Java bytecode. These annotations and/or informations should inform on some component abstractions. We are not sure to find such informations but we assume that as soon as component design is be applied (later), it will help to inject annotations in the plain Java code.
  A practical question remains: which information or annotations do we need?

- Input Model
  We do not want to have an instantiated model of the Java programs *i.e.* we carry only some informations of the program not all of them (no model transformation for instance).

- Reverse engineering goal
  The goal is to find abstractions than realise full reverse engineering. The abstraction help to analyse some properties of the program, to get an abstract model, either to compare with an existing one (conformance) or to document the application.

- Separation of concerns
  We discussed about two main problem issues: structure/behaviour, one or more target.

  – We distinguish the structural information from the behavioural. Both aspects are orthogonal and existing works mainly relate to one aspect only. Moreover finding the behavioural abstractions depends on both the Java code and the structural model. The structural part is at first a delimitation of what are the components in the Java code.

  – We may want to target several abstract models. Shall we implement a tool for each of them or try to share both procedure and knowledge. Note that some of these models can have only structural features. One way to proceed is to use metamodels (pending issues to be studied on thursday).

  This separation of concerns also make it easy to split the project realisation.

- Background
  Other contributions of the project are recalled: cross LTS extensions, WFR definitions.

2.2.3 Thursday, September 6, 2007

Time	Activity	Speaker
09:00	Welcome	Pascal André
|	CoCoME Contest	Jan Kofron
|	"ECONET tasks" proposal	Petr Hnetynka
|	CoCoME in SOFA 2.0	Jan Kofron
12:00	Round tables and discussions	-
13:30	Java Path Finder, Bandera	Jiri Adamek
|	Project modules and interfaces (discussions)	-
17:00	Toward an annotation language (discussions)	-

Welcome

Pascal introduced the second day working session by giving an abstract of the discussions and the gainful decisions of the first day. Since the goals of day 3 were roughly reached we can now fight the ones of day 4, which are convergence of the means (see section 2.2.2) and especially try to advance on the input format, the benchmark and the available techniques. To this end a presentation of the benchmark has been planned in the morning and another one on Java Path Finder (and Bandera) in the afternoon. The rest of the day (morning and afternoon) is reserved for discussions.

The slides (second plan) are provided on the Workshop Wiki at:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:program07

CoCoME Contest

Jan Kofron presented Common Component Modeling Example contest.

CoCoME http://agrausch.informatik.uni-kl.de/CoCoME is an international contest aiming at comparison of different modeling approaches on a common assignment organized by Technische Universität Clausthal, Universität Karlsruhe, Politecnico di Milano, Charles University in Prague. Sharing a common close-to-real-life assignment should reveal strengths and weaknesses of the different modeling approaches and thoroughly test their applicability in practice.

The assignment is a trading system, a business application for managing chain of stores with about 200 stores and 8 cash desks per store. Participants had as input a partial specification via UML, informal specifications as use-cases for the requirements, deployment diagrams for the distribution and a reference implementation (realised by a teacher and students).

The workshop held at Dagstuhl in Germany. Eighteen teams were involved in that contest. The objective was to model the application on various aspects (structure, deployment, behaviour modelling extends behaviour protocols). Note that UML models were not fully consistent with the code and the components were given.

The slides of this intervention are provided on the Workshop Wiki at:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:program07

"ECONET tasks" proposal

Petr Hnetynka proposed some answers to pending questions on the objectives and the means.

To the question "What is an input?" his answer is "plain Java".

For obtaining a behavior specification we need also a model (i.e. architecture). To the question "How to obtain model?" his answer is "it is already given or we extract it from the plain Java sources.

We need a single concrete metamodel (for specifying models). The problem is that each component system has its own means for specifying models but most of component systems are similar (black-box component, provided/require services, nesting), thus we can use the core of the SOFA 2 metamodel as a concrete metamodel.

In addition to the objectives of the day he also proposed a first task assignment proposal: model extraction (-?) and behavior specification extraction (DSRG).

The slides of this intervention are provided on the Workshop Wiki at:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:program07

CoCoME in SOFA 2

Back to the CoCoME, Jan Kofron overviewed a SOFA solution for the Common Component Modeling Example. He skipped the SOFA 2 presentation to focus on the general design solution and the verification of a part of the system.

The following summarizes modeling of the CoCoME assignment using SOFA 2.

- Architecture
  First, the SOFA architecture of the TradingSystem (Fig. 2.2) was created, forming basis for further work.

- Behavior modeling
  All CoCoME components were specified using EBP. Parts of the CoCoME assignment are ambiguous and even contradictory; the behavior specification is mainly based on the reference implementation.

- Verification
  Checking communication of components for errors (Bad-activity no-activity (deadlock) infinite-activity), Compliance checking (Translation of EBP specification into Promela using the SPIN model checker to search the state space),
  Code checking (Via combining Java PathFinder with Behavior Protocol Checker).

- Performance

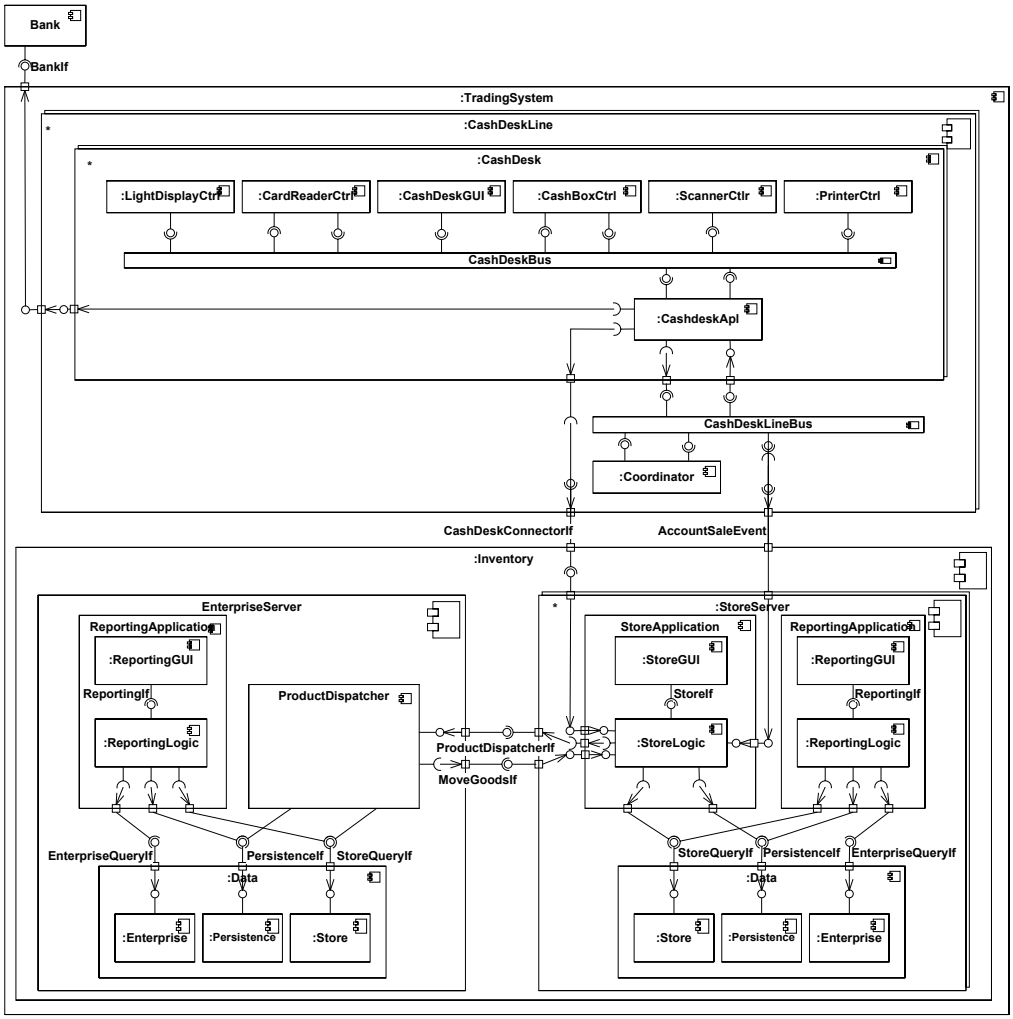


Figure 2.2: CoCoME: Sofa static architecture

Benefits : As SOFA 2 explicitly supports components throughout the entire software lifecycle, the architecture erosion is mitigated. Moreover automatic connector generation provides seamless component distribution. Behavioral modeling via EBP and subsequent verification allows for reasoning about correctness of the design even before actually having an implementation. Then with the implementation available, one can use code checking to find out, whether the implementation obeys its EBP specification.

The solution involves (i) modeling architecture in SOFA metamodel, (ii) specification of component behavior via extended behavior protocols, (iii) checking behavior compliance of components, (iv) verification of correspondence between selected component Java code and behavior specification, (v) deployment to SOFA run-time

environment (using connectors that support RMI and JMS), and (vi) modeling of performance and resource usage via layered queuing networks. We faced several issues during implementation of the CoCoME assignment in SOFA 2. Most notably, the architecture was modified in order to improve clarity of the design. In particular, the hierarchical bus was replaced by two separate buses and the Inventory component was restructured. Extended behavior protocols for all the components are based on the provided plain-English use cases, the UML sequence diagrams, and the reference Java implementation (the assignment does not include a complete UML behavior specification e.g. via activity diagrams and state charts).

Jan showed the SOFA solution for CoCoME: static view, behavior view (BP [sofa, fractal], EBP [Sofa 2]), deployment view, performance view (model performance), comparison with UML, flat / hierarchical models, multiplicity, evaluation.

The slides of this intervention are provided on the Workshop Wiki at:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:program07

Morning Discussions

Here is a short summary of thursday discussions and decisions.

Petr drew a quick picture of a Sofa metamodel (Fig. 2.3) which should correspond to all our component models (with different names).

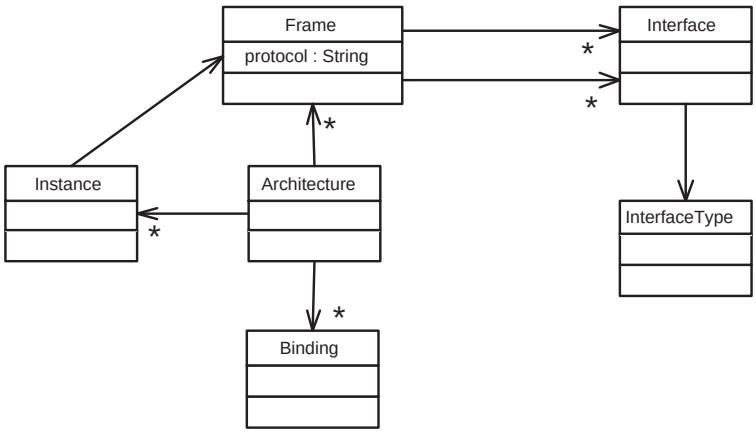


Figure 2.3: Sofa: short metamodel

Then we discussed about the application design (Fig. 2.4). The figure shows a three parts architecture with a common metamodel and a dependency between structural and behavioural abstraction. The contribution is an annotation definition, a first prototype (Java analyser, xDoclet) that must be incremental. See also Spring and Fractal.

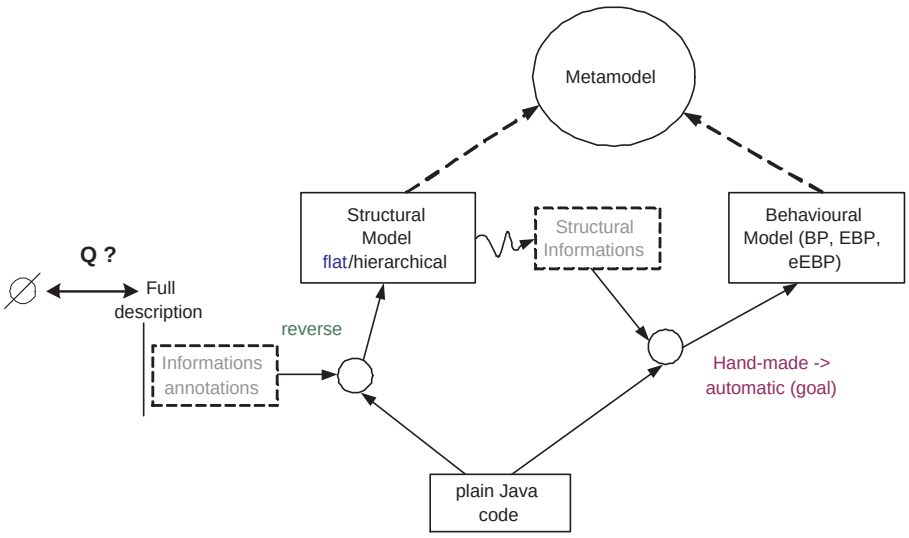
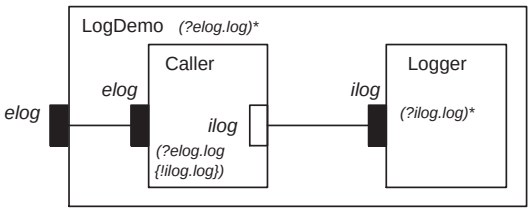


Figure 2.4: Econet Architecture: draft 1

After, Tomas explained a short example of code generation from SOFA architecture. He took the `LogDemo` example and illustrated different versions of the Java generator.




```
module logdemo {
  interface LogInterface {
    void log(in string number);
  };
  frame Caller {
    requires:
      LogInterface Log;
  };
  frame Logger {
    provides:
      LogInterface Log;
  };
};
```

There are two primitive components - `Caller` (or `Caller` in his example) and `Logger`. `Logger` provides the interface `LogInterface`, `Caller` requires this interface, in the component `logdemo` these two components are instantiated and their provision and requirement are bound together.

In the first translation, plain Java code is generated with a specific implementation of requirements (variable + methods).

```
 // LoggerImpl.java
package SOFA.demos.logdemo;

public class LoggerImpl implements LogInterface {
  public void log(java.lang.String msg) {
    System.out.println("**** LOG: "+msg);
  }
}

// CallerImpl.java
package SOFA.demos.logdemo;

public class CallerImpl implements Runnable {
  SOFA.Component.DCUP.DCUPComponentManagerImpl cm;
```

```
  LogInterface LogRequirement;
  boolean end;
  boolean stopped;
  public CallerImpl(SOFA.Component.DCUP.DCUPComponentManagerImpl _cm) {
    ...
  }
  private void setRequirement() throws SOFA.Component.NamingException {
    LogRequirement = (LogInterface) cm.getRequirement("Log");
  }
  public void run() {
...
  }
  public void setEnd() {
  }
  public boolean isStopped() {
  }
}
```

In a second translation, the provisions and requirements are specified as annotations in Java for a post processing.

```
 // LoggerImpl.java
@provision(log)
public class LoggerImpl implements LogInterface {
  ...
}
```

```
// CallerImpl.java
@requirements(log)
public class CallerImpl {
  boolean end;
  boolean stopped;
  public CallerImpl() {
    ...
  }
  public void run() {
...
  }
  public void setEnd() {
  }
  public boolean isStopped() {
  }
}
```

The architecture is generated or predefined. There are templates for connectors.

Last, we discussed about flat component / hierarchical components and the composition operators. At first, we cannot re-engineer the composite components from scratch. One can build hierarchies on performance criteria or distribution criteria. But it is enough for instance to build the primitive components and then to build the structures manually. Jiri proposed launching the code with introspection rather than parsing the code.

Java Path Finder, Bandera

The afternoon started with a short description of JPF (and Bandera) by Jiri Adamek. The slides of this intervention are provided on the Workshop Wiki at:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:program07

Bandera

Jiri explained that Bandera was unstable at the time he studied it and the second version was not really public on the web site[3]. Bandera is a 6-years old source transformer. Bandera allows to check in a semi-automatic way different properties for different abstractions (Fig. 2.5). Two years ago a model-checker for Java, called BOGOR[4], has been developped ; it can be a future source of information and tool.

[3] http://bandera.projects.cis.ksu.edu/
[4] http://bogor.projects.cis.ksu.edu/

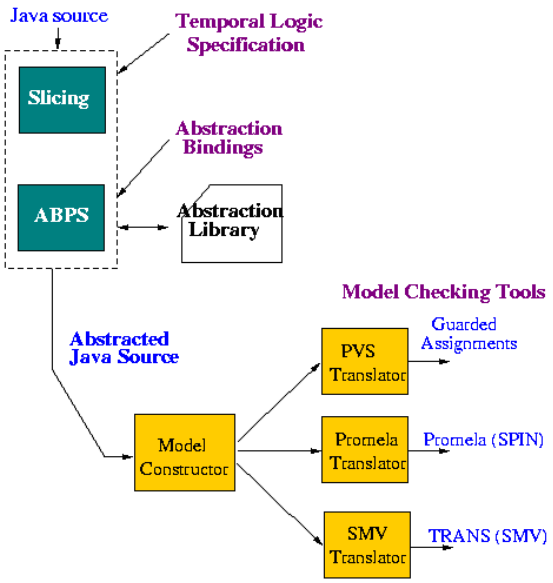


Figure 2.5: Bandera overview (from web source)

Java Path Finder

JPF is a Java model checker developped at NASA Ames Research Center[5].

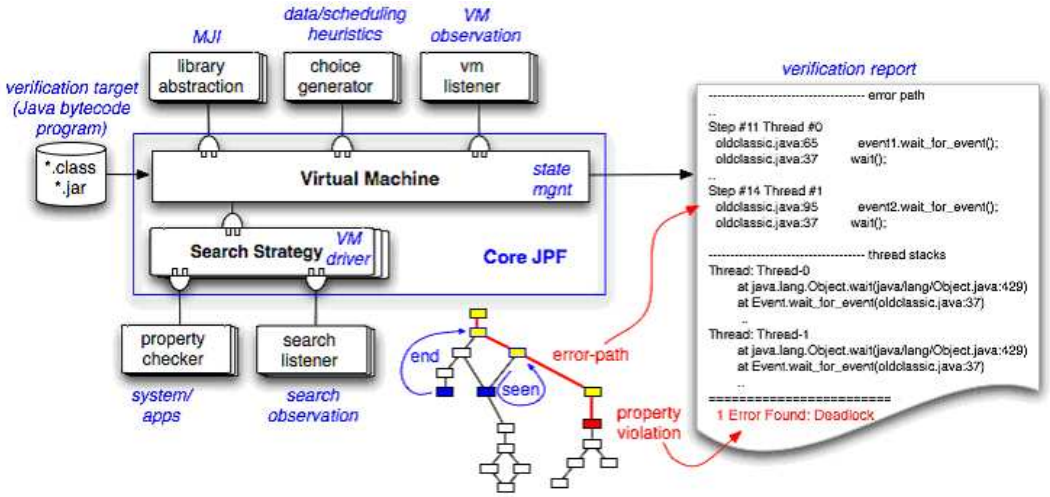


Figure 2.6: JPF model of operation (from web source)

In general, JPF is capable of checking every Java program that does not depend on unsupported native methods. The JPF VM cannot execute platform specific, native code. This especially imposes a restriction on what standard libraries can be used from within the application under test. While it is possible to write these library versions (especially by using the Model Java Interface - MJI mechanism) there is currently no support for java.awt, java.net, and only limited support for java.io and Java's runtime reflection. Another restriction is given by JPF's state storage requirements, which effectively limits the size of checkable applications to 10kloc (depending on their internal structure) if no application and property specific abstractions are used. Because of these library and size limitations, JPF so far has been mainly used for applications that are models, but require a full procedural programming language. JPF is especially useful to verify concurrent Java programs, due to its systematic exploration

[5] http://javapathfinder.sourceforge.net/

of scheduling sequences - an area which is particularly difficult for traditional testing.

One problem in a component view is that it accepts only closed java programs. It explores the state space. Out of the box, JPF can search for deadlocks and unhandled exceptions (e.g. NullPointerExceptions and AssertionErrors), but the user can provide own property classes, or write listener-extensions to implement other property checks. A number of such extensions, like race condition and heap bounds checks are included in the JPF distribution. One advantage of JPF is that it accepts **listeners** for custom search algorithms and an access to a Model-Java interface via its API (Fig. 2.7).

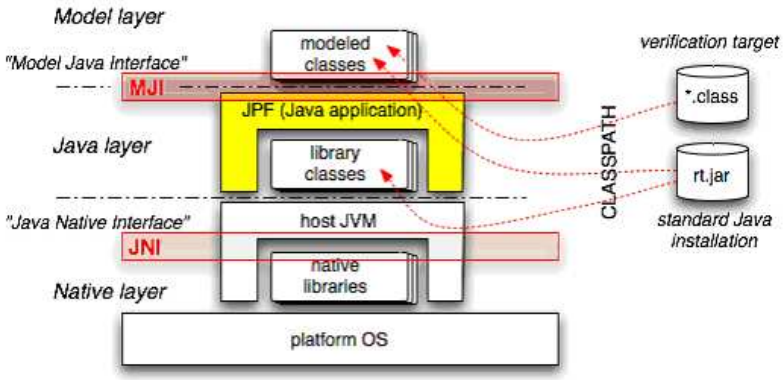


Figure 2.7: JPF Java layers (from web source)

Project modules and interfaces (discussions)

In this part we discussed more precisely on the project architecture (Fig. 2.8), its modules and the minimal information we need for behavioural abstraction.

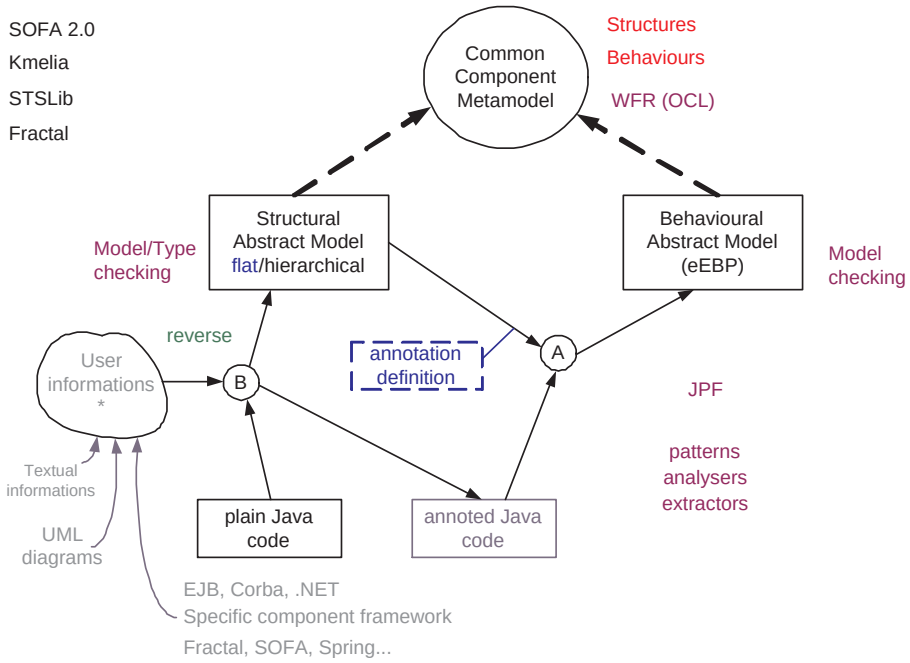


Figure 2.8: Econet Architecture: final version

The abstraction process ((Fig. 2.9)) should infer components and structure for the behavioural abstraction. The information are given directly by the source code via Java annotations. A main information for the latter is the "entry point" information (intialisation). The entry point denotes the active component.

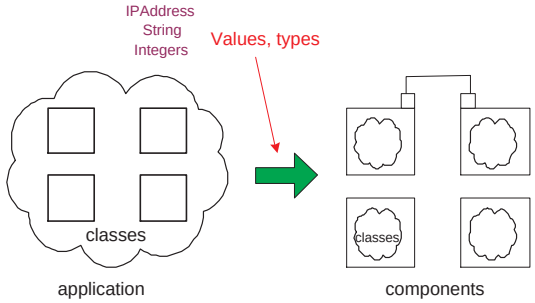


Figure 2.9: Abstraction Process

The B process is more general than the A one since it can apply to non-behavioural abstract models. *It provides more complete results with less information*. The borders delimit frontiers and component contexts (separate business and management for example) and link classes to components. The goal is to extract interfaces (one or more) using hierarchical graph analysis for example. The approach uses annotation based methods and techniques coupled with user friendly graphical interface to get the missing informations from the user. Possible starting points and resources are: Java only, UML models, component models, EJB program...

Toward an annotation language (discussions)

To define the required informations (and annotations) we tried to map the model concepts (Fig. 2.10). We separate business functionalities from non-business ones (related to Java computations or to management handling e.g.

setRequirements). We should find annotations about provided interfaces (which is absent or difficult to find in plain Java). Provided interface are annotated on Java interfaces and methods. Required interfaces can be annotated from Java attributes. All the classes belonging to a component should be annotated. One question is: how many instances of a component do we accept, only one?

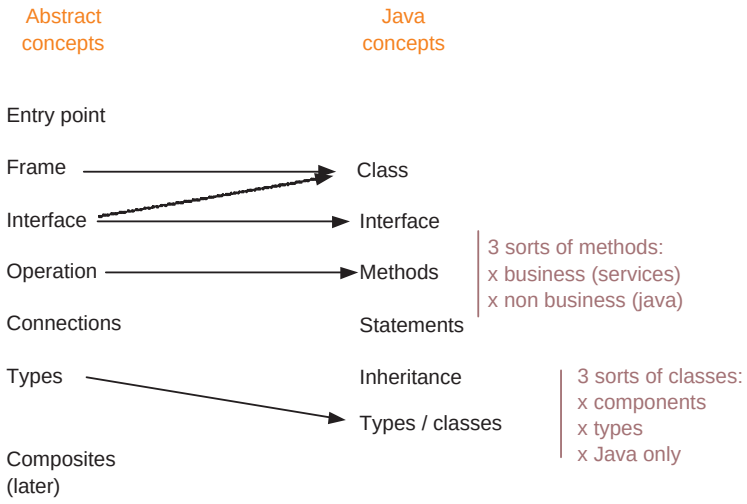


Figure 2.10: Mapping concepts

Remark 1: there is one protocol per component (frame) and not per interface.
Remark 2: the annotations must be compliant with other Java annotations (metamodel).

2.2.4 Friday, September 7, 2007

Time	Activity	Speaker
09:00	Welcome	Pascal André
|	Comparison of Abstract Models (discussions)	-
|	Common MetaModels (discussions)	-
12:00	Round tables and discussions	-
13:30	Technical discussions (RE techniques and tools,	-
|	annotation Language, collaborative tools...)	-
17:00	Closure	Pascal André, Ondrej Sery, Petr Hnetynka

Welcome

Pascal introduced the third day working session by giving an abstract of the discussions and the gainful decisions of the previous days. We significantly progressed on the project structure and module interfaces and reached partially the convergence of the means (see section 2.2.2). This can be summarised as follows

- DSRG experience - CoCoME, Behaviour Extraction, Tools (JPF, Bandera)
- Project Architecture (Fig. 2.8) 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

There remain things to do for this last day: the detailed tasks definition, the responsibility repartition and the planning building. Moreover some discussions were mandatory on technical informations: models (abstract and concrete models toward a common metamodel), techniques (control flow or parsing, detailed annotation language, ...). The (optimistic) goal of day 5 is that each participant has a somewhat clear idea of what he will do later.

Here are some guidelines for the definition of the tasks:

- What to do? on the draft 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
  - Evaluation (october 2007)
  - Workshop Nantes (March 2008)
  - Workshop Cluj (August 2008)
- ...

The slides (third plan) are provided on the Workshop Wiki at:
http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:pragues2007:program07

We started the day by a comparison of abstract models and after several discussions on the CMM (Common MetaModel), the AL (Annotation Language), the tools and the task repartition.

Comparison of Abstract Models (discussions)

The discussions continued with a fast comparison of the three abstract models (SOFA, KADL, Kmelia) in order to grasp the structural and behavioural models and therefore the annotations and some kind of metamodel.

Concept/Model	SOFA EBPL	KADL	Kmelia
Attachment	Frame	Component	Service+ component
Operations (computation)	atomic assignments (constants?)	atomic functions (algebraic)	atomic action+ service calls
Types	Enums	any ADT	"complex but open" means ad hoc
Guards	logic + enum	logic + ADT	logic + ad hoc FL
Dynamic formalism	reg. expr.	state transition	state transition + "hierarchy"
I/O	!?	? ! *	? !?? !!
Labels	?iface.notified {!iface2.pre}	[guard] event com/action	[guard] action* (actions can be com or functions)

We studied the corresponding Java constructs in an engineering/reverse-engineering points of view.

Concept/Model	SOFA EBPL	KADL	Kmelia
Attachment	set of classes	set of classes	set of classes
Operations (computation)	plain methods user Java statements	plain methods algebraic translation	methods + behaviours generated code
Types	Java types	Java types classes (ADT)	Java types + classes
Guards	boolean expr.	boolean methods	conditions
Dynamic formalism	control flows (RMI...)	control flows (LTS Library)	various statements (control structure, messages, methods)
I/O	method calls parameters	method calls parameters	method calls parameters
Labels	assignments user Java statements	if-then-else patterns	statements (Kml-lang)

We started the day by a comparison of abstract models and after a discussions on the common metamodels and the annotation language, and the task repartition.

Common MetaModel (discussions)

The common component metamodel will include only the common part in its first design, leaving some holes for specific features. The structural concepts are quite similar in the target languages. The main features are those of the annotation language. They differ mainly on the representation of behaviours (Fig. 2.11).

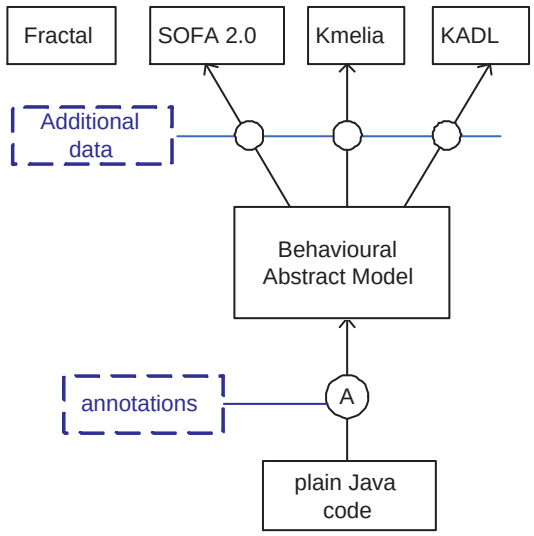


Figure 2.11: Common Component MetaModel

Technical discussions

First a debate opened on the behaviour abstraction RE techniques and tools. On one hand, the usual way to proceed is to build the control graph: the states are the steps in the system evolution - its state space - and the transitions are the events and actions performed (method call, statements, control structure). Then we can split the system behaviour onto components or just model check it. The problem is that it contains low level information and it should be abstract: group state and transitions. This technique is well fitted to LTS base components. On another hand, we can parse the java code and build parse trees (AST abstract syntax tree with terminal and non terminal tokens). This provides modular trees that can be merged/collapsed/extended to a more general flow. This technique seems to be interesting to build regular expressions for the SOFA model. It uses the SunAPI for Java and tools like `javacc`, `antlr`, `JavaCup`, etc. In any case, one has to cross the results with existing component structures. The second solution is chosen.

Some issues remain open:

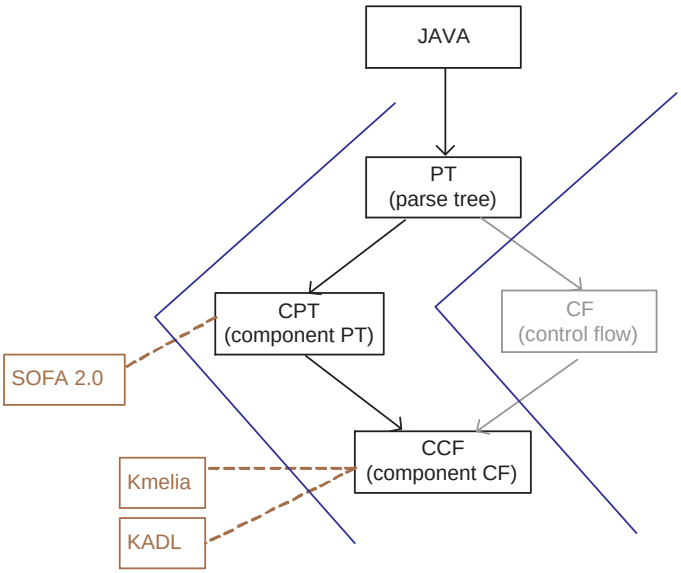


Figure 2.12: Java Behaviour Abstraction

- Parse Tree
  - Coloration used to collapse parts of the parse trees.
  - Tools?
- Control Flow
  - Coloration used to collapse parts of the control flows.
  - Tools?
  - Format? textual +API
- Find a good common format for both parse tree and control flow.

Then we exchanged views about the annotation language. The **constructs in the component model** are associated or derived from one or several annotations in the code.

Frame/Component

- Corresponding Java construct: one or more classes
- Corresponding Java annotation:
  - target class or interface `@InComponent(annotation_source, name_of_the_component)`

Interface

- Corresponding Java construct: one or more interfaces, or a set of methods from a class
- Corresponding Java annotation:
  - target class `@provided(annotation_source, name_of_the_model_interface, name_of_the_ja`
  - target method `@provided(annotation_source, name_of_the_model_interface)`
  - target attribute `@required(annotation_source, name_of_the_model_interface)`

things to think about later : (temp requirement passed as parameters)

"entry point"

- Corresponding Java construct: The main class of a component
- Corresponding Java annotation:
  - target class `@initclass(annotation_source, name_of_the_component)`
  - target method `@initmethod(annotation_source, name_of_the_component)`

  Gilles: `name_of_the_component` seems redundant because the method or the class should already belong to a component (see InComponent annotation)
  things to think about later: (consistency rules needed? is having several initclasses, methods in a component legal?)

Operation Do we need it? We already know methods from the provided annotation

- Corresponding Java construct: Method
- Corresponding Java annotation:
  - target method `@`

  Business methods have to be singled out

Types Goal: identify parameters of business methods
Problem: String or Integer may be used as business object sometimes and sometimes not

- Corresponding Java construct: class or interface
- Corresponding Java annotation:
  - target class or Interface `@initclass(annotation_source, name_of_the_component)`
  - target method `@businessparameter(annotation_source, parameter_name)` (this one takes precedence over the previous one)

  things to think about later: another annotation could be for abstracting data : telling that an integer only has a few cases...

Business types have to be annotated. In some cases (library without accessible source code) they can't be, so a textual configuration file will be used to list them or to list non-business types. White list, black list, regular expressions over full name?

Last we discussed about tasks, responsibilities and milestones. Every body participates to the common annotated bibliography on the wiki. Here is a pictured summary of the responsibility and participant repartition on the project structure (Fig. 2.13).

As a collaborative tool, a versioning system (SVN) will be installed by DSRG, in addition to the wiki platform installed by COLOSS. We want to share the tools (source and experience) used during the development.

Closure

Production

- Workshop Report
  - Collect paper and slides.
  - Summary of the discussions
- Bibliographical Notes

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

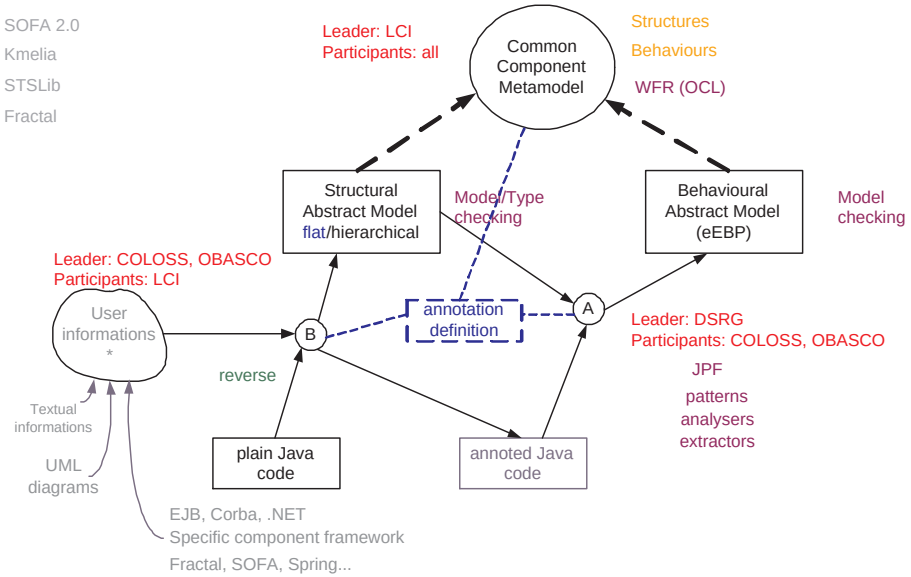


Figure 2.13: Econet Architecture: responsabilities

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

see also the initial 'Second year objectives'

The organizers thanks the participants for their contributions and take a date for a next workshop at Nantes in 2008.

Chapter 3

Project Architecture

The contents of this chapter has been defined individually after the workshop. It presents a detailed vision of the three subprojects, defined in the workshop. There remain some coordination to process on the common parts (especially on the interface formats and model tools) and also to share the experience on tools.

3.1 Structural Abstraction Subproject

Writer: Pascal André, Gilles Ardourel

This is currently a draft version.

3.1.1 Objectives and Goals

The objective of the process B (Fig. 2.8) is to build a structural component model and a corresponding annotated Java code. These two elements are inputs of the process A (see the detailed process in section 3.2). The model is also an instance of the metamodel (see the detailed description in section 3.3) that will control its consistency.

A general view of the process B is given in figure 3.1; 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.

The process B is in fact iterative because its source is variable (in the sense that it may include many informations from different nature) and target different goals. For instance one goal is to abstract structural elements of a component model from a plain Java code and user informations. Another is to read and interpret existing annotations. Another is to check the compatibility between one component model and an annoted Java code... On each iteration, the process accepts a Java program (with or without annotation) and a component model (possibly empty). It computes some information, sometimes using external tools and human interaction. This information modifies the a Java program and the component model.

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

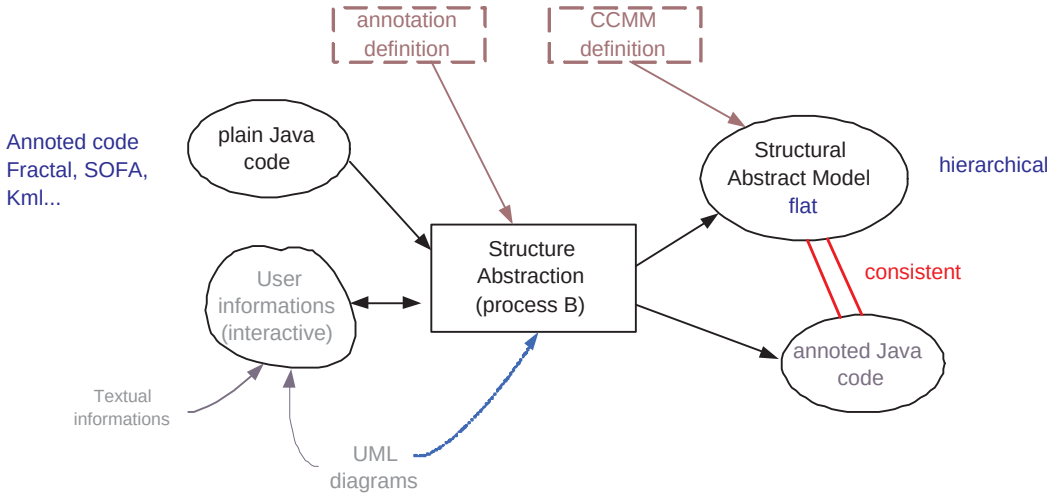


Figure 3.1: A general view of the process B

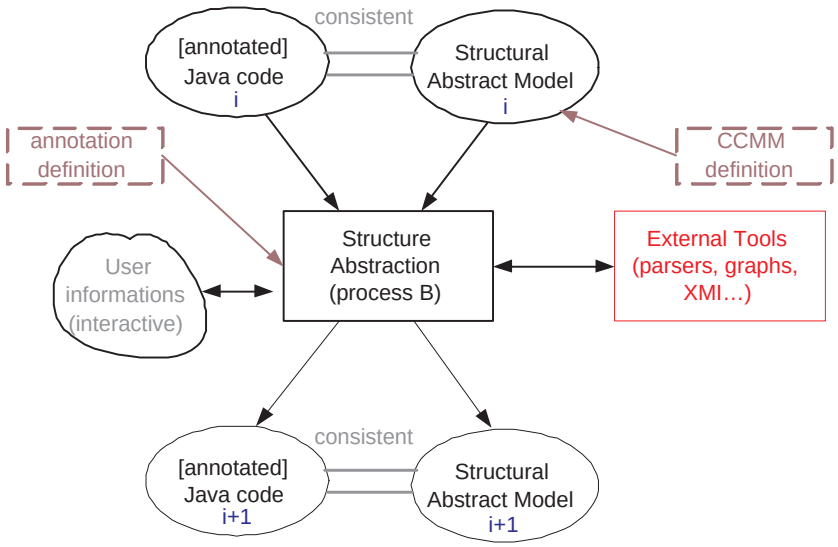


Figure 3.2: An iterative view of the process B

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. ...

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.

Consequently, process B is rather a tool box or a sequence of subprocess applications.

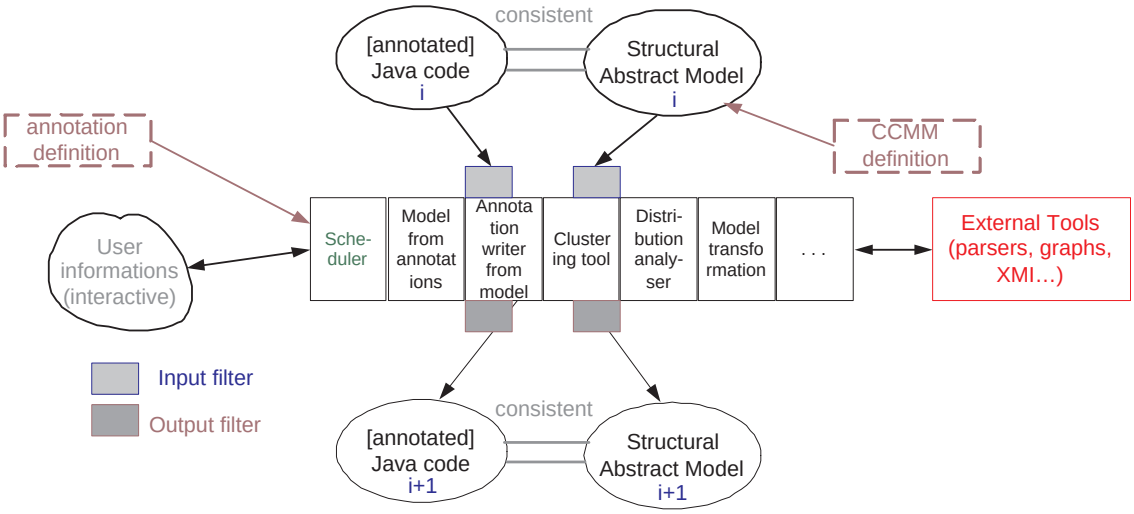


Figure 3.3: An architectural view of the process B

3.1.2 B transformations and tools

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 3.2.3 on page 49.
   - 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.

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

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.

3.1.3 Interface

The annotation language and component metal model have been discussed but the full definition of interface need an interoperable format. Especially we need a couple (format, tool) to support model and metamodel instanciation.

1. Metamodel format (XMI, ECore, MOF, model API...)

2. Model format (XMI, ECore, MOF, model API...)

3. File exchange vs. model repository.

It is not interesting to exchange text files ou XML files between the three subprojects because we should all have to write readers and writers modules and manage our own representation of the models. Instead we have to share this common part.

This part is a common part. It is discussed in section 3.4.2

3.1.4 Organisation

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.

The program is designed as a set of tools which can be developed independently provided the interfaces are well defined (see section 3.1.3). 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).

3.2 Behavioural Abstraction Subproject

Writer: Tomas Poch

Reverse engineering general Java application into component application consists of two tasks. First, extraction of an architectural view (identification of components, their interconnections, etc.); second, extraction of a dynamic behaviour specification of the components identified during the first task. Constituting an interface between the two tasks, the architectural information is to be stored in a form of Java annotation in the actual Java sources of the application being reverse engineered.

This section presents goals, means and organization of the second task in the scope of the ECONET project, and thus summarizing a part of the results of the Prague'2007 Workshop working sessions.

3.2.1 Goals

Each group participating in the project has developed its own formalism for behaviour specification. 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,
- *STS* developed by OBASCO.

The individual behaviour specification formalisms differ a lot. This makes creation of a general tool very difficult. However, steps common to extraction of any behaviour specifications (in particular behaviour protocols and LTS-based formalisms eLTS and STS) might be identified. Thus, the general approach is to divide all necessary steps of behaviour 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 behaviour 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 behaviour suitable for subsequent creation of concrete behaviour specifications in a chosen formalism.
3. Create formalism-specific back-ends for extraction of behaviour specification from the intermediate specification.

3.2.2 Annotations

Reflecting the goals stated in the previous section, the process of reverse engineering behaviour specification starts, where the first process (reverse engineering component architecture) ends—i.e. by extracting and using architectural information provided in a form of Java annotations directly in the Java sources and, of course, the Java sources themselves.

The necessary architectural information consists of:

1. assignment of Java classes to individual components,
2. identification of provided interfaces and their methods,
3. identification of required interfaces (as class attributes), and
4. demarkation of the components' initialization code.

Moreover, in order to help extraction of behaviour specification, it might be helpful to have explicitly annotated *ValueType* classes. By a ValueType, we mean a class used for storing and passing data rather than providing a specific functionality.

As the ValueTypes have to be reflected in the behaviour specification, hints about the abstraction may be necessary. By abstraction we mean replacement of the actual ValueType by a simpler abstract type (for example integer type by enumeration). Although the abstract type has less states, it captures all information necessary for the component behaviour. For example if the behaviour of the component depends on the sign of the integer parameter, all information we need can be stored in the single bit of information. Thus, the boolean type suffices for the parameter.

The information about ValueType abstraction could be specific to the target formalism and even more added manually by a human user. More detailed description of the proposed set of annotations follows.

Component

One or more Java classes can be assigned to a single component. Such an assignment is specified by the following annotation of a Java class:

```
@inComponent(annotation_src, component_name)
```

As in the rest of this section, the `annotation_src` specifies the origin of the annotation—i.e. a name of a tool which inserted the annotation in the code or whether the annotation has been inserted manually.

Interfaces

Methods of Java classes may be assigned to component's *provided interfaces* in two different ways. First, as an annotation of a Java class:

```
@provided(annotation_src, model_iface_name, java_iface_name)
```

This way, 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. The second possibility is to mark individual methods of a Java class by the annotation:

```
@provided(annotation_src, model_iface_name)
```

This is necessary in a case the component's provided interfaces do not correspond with the Java interfaces.

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:

```
@required(annotation_src, model_iface_name)
```

Component's initialization

In order to be able to derive the behaviour specification correctly, knowledge about a component initialization is crucial. By initialization, we mean instantiation of classes, assigning references to the required interfaces (i.e. to the class attributes), starting threads of active components, etc.

Typically, there are two ways of initialization. First, the initialization is performed by a class. This class is the first instantiated and is responsible (its constructor) for the instantiation and initialization of the component's content. To identify such a class, the following annotation is used:

```
@initclass(annotation_src, component_name)
```

The second possibility is that the component content is instantiated and initialized by a static method (an equivalent of the *main* method).

```
@initmethod(annotation_src, name_of_the_component)
```

Data and method parameters abstraction

Some of the internal component data and method parameters might be of crucial importance for correct extraction of the behaviour specification. Those should be also explicitly annotated, in order that the behaviour specification extractor could take them into account and provide a necessary abstraction.

However, it is hard to predict the needs of behaviour extraction in advance and thus these annotations are yet to be further elaborated. The initial proposal is to identify ValueTypes classes and interfaces, and therefore mark all their instances as important for the component behaviour.

```
@businessparametertype(annotation_src)
```

As some types might be important only in a specific context (e.g. String), there is also possibility to mark particular method parameters and Java class attributes as important for business logic.

```
@businessattribute(annotation_src)
@businessparameter(annotation_src, parameter_name)
```

For the actual behaviour specification extraction, hints on how to perform a data abstraction from the concrete values of these types have to be added by a human user. A particular format of these hints will be further analyzed.

External dependencies

Some architecture information cannot be introduced into the code as annotations. Mostly, these are the external dependencies on the class libraries used by the component code. Classes from the libraries can be also important for the behaviour extraction but in general, they cannot be directly annotated. Therefore, these dependencies are listed in an extra file. The complete code of the component is thus a set of classes annotated by both the `InComponent` annotations and also the content of the dependencies file.

3.2.3 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.

3.2.4 Generic analysis tool (GAT)

Behaviour specification extraction is divided into two steps. First step is to preprocess the annotated Java sources and propagate information about architecture to the actual Java code, so that Java statements can be categorized as either externaly visible events (method calls on required interfaces), visible actions (change of business attributes), changes in control flow (if and cycle statements), or internal invisible actions (work with local variables). The result of the process is an abstract syntax tree with nodes “colored” by the architecture information (obtained from the annotations).

In the second step, the produced cAST (colored AST) is used to create a control flow graph (CFG) representing the code. Again, nodes of the graph are "colored" by the architecture information (cCFG). Both cAST and cCFG are available by the defined API (all details about the API have to be analyzed yet) and/or in a serialized textual form.

A specific formalism back-end can choose whether cAST is sufficient for it or whether the usage of cCFG is necessary.

Based on the preliminary analysis, for the EBP back-end cAST is sufficient while the LTS-based formalism (eLTS and STS) requires an existence of cCFG.

The whole process is illustrated with Figure 3.4.

There is a bigger set of transformations which can be performed among AST, cAST, cCFG. Some of them, like coloring, are necessary part of the proposed process. Other transformations, like omiting actions on non-ValueType data, can be used by some back-ends or even the single one. These transformations should be available to the authors of the back-ends in order to prevent duplicate work.

3.2.5 Reverse engineering back-ends

Depending on the choice of the target behaviour specification formalism (BP, EBP, STS, eLTS, ...) a specific back-end of GAT is used to extract the behaviour specification. For example, the back-end for extraction of behaviour protocols abstracts from everything except externaly visible events and control flow (replacing `if` statements by alternative and cycles by repetition).

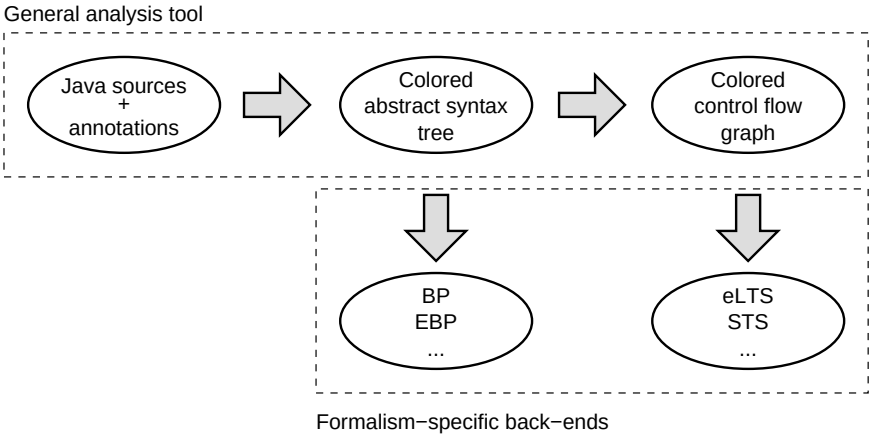


Figure 3.4: The two step process of reverse engineering behaviour specification

3.2.6 Organization

This task is led by the DSRG group; the OBASCO and COLOSS groups also participate. The first version of the set of annotations was created by all groups during the Prague 2007 Workshop. However, the initial version is expected to be further enhanced in order to allow automated (or at least semi-automated) abstraction of ValueTypes.

More specifically, DSRG is responsible for creation of the general analysis tool. For this tool, each participating group (DSRG, OBASCO, COLOSS) is going to implement their own back-ends specific to the behaviour specification formalisms they use.

First results on the generic 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).

3.3 Metamodel Abstraction Subproject

Writer: Dan Chiorean

3.3.1 Objectives and Goals

Designing and coding a powerful repository that implements the Common Component Metamodel architecture represents the main objective of the current sub-project. Accomplishing this objective will support modelers in creating and validating models that instantiate the Common Component Metamodel in an efficient manner.

3.3.2 Participants

LCI will represent the main participant, responsible with metamodel's design, implementation and testing. DSRG, COLOSS and OBASCO teams will provide the requirements. All teams will be involved in testing the repository, and, of course, in taking the final decision.

3.3.3 Means

The static semantics of the repository will be specified by means of OCL constraints. The repository's (metamodel's) API will also be described in a formal manner, by means of observers, specified in OCL.

The OCLE tool will be used both in validating metamodel's static semantics and API observers, and in generating the Java code corresponding to Additional Operations and OCL assertions.

3.3.4 Tasks and Schedule

1. Investigating the existent Component Models in order to extract the common parts and produce the informal specification of the Common Component Metamodel (CCM) architecture and metamodel assertions.
   Responsibles: DSRG, COLOSS and OBASCO teams
   Deadline: 15^{th} of November 2007

2. Specifying metamodel assertions and API observers in OCL.
   Responsible: LCI
   Deadline: 1^{st} of December 2007

3. Validating the OCL specifications on significant models.
   Responsible: LCI
   Deadline: 15^{th} of December 2007

4. Analyzing state of the art approaches and generating the java code corresponding to the CCM repository, including the code associated to assertions and Additional Operations. At least two repositories will be created (one containing the code generated using the appropriates EMF tools and another one using OCLE).
   Responsible: LCI
   Deadline: 1^{st} of February 2008

5. Testing and improving the above mentioned repositories using different models.
   Participants: all teams
   Deadline: 15^{th} of March 2008

6. Choosing the ECONET repository.
   Participants: all teams
   Deadline: 1^{st} of April 2008

3.3.5 Using assertions in modeling - an evaluation time view

Using assertions in modeling is incomplete if this is restrained just to better understand the problem and to emphasize the conditions that have to be accomplished by the clients and the provider of a functionality. The true benefits can be obtained only if assertions are used at runtime, assisting the user in preventing software runtime crashes or obtaining inaccurate results. Our interest is not limited to validation. In case of constraint violation, we are interested in identifying rationales and even more, in fixing bugs and errors. The price that needs to be paid for obtaining the maximum benefits from using assertions in modeling is to take into account the moment when constraints are evaluated. The main target of this paper is to highlight the manner in which the moment when assertions are evaluated influences their specification.

In order to support an easier understanding of our statements, we will consider the CoreComponent Metamodel - a simple model grouping the common features of most component-oriented modeling languages [BHP06]. Moreover, in Figure 3.5, only the elements referred in assertions are represented.

Model validation (model checking) can be regarded from two perspectives (views): static checking and dynamic checking. As we will see in the following, these views influence assertions' specification.

Static checking - The constraints aid in correcting and validating a previously constructed model

Models are metamodel instances - our objective is to check if the analyzed models comply with all the constraints associated to different metamodel elements. The problem is entirely similar with that of an UML model validated against WFR specified at the UML metamodel level. In this case, constraints are specified by means of invariants.

As mentioned in [Moo00], during its construction, the model is incomplete and sometimes it is incorrect against the constraints associated to the modeling language. Therefore, before doing model transformation, it is important to check model correctness and completeness (model compilability).

In case of the CoreComponent Metamodel, the XOR constraint between the unidirectional associations from `SubcomponentInstance` toward `Frame` and `Architecture` (graphically specified in Figure 3.5), can be expressed by means of the following invariant:

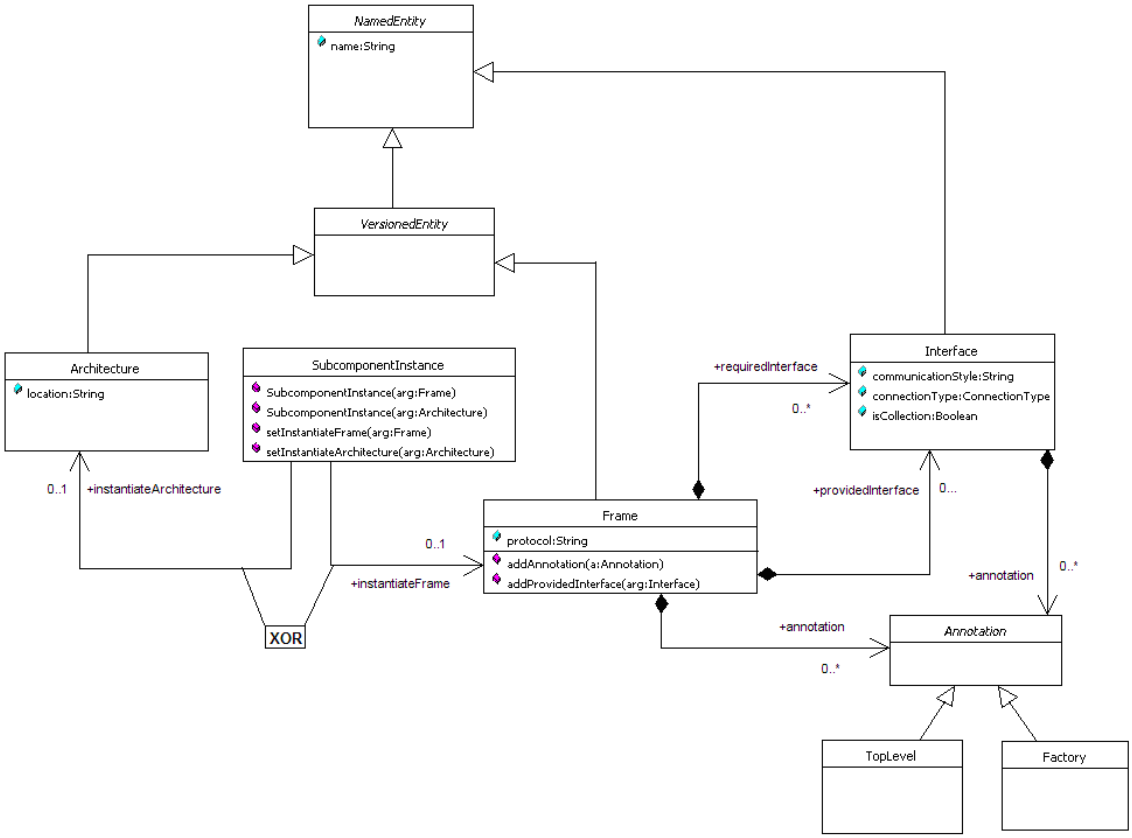


Figure 3.5: A part of the CoreComponent Metamodel

```
(1) context SubcomponentInstance
     inv FrameOrArchitectureAssoc:
       self.instantiateArchitecture.isUndefined xor
       self.instantiateFrame.isUndefined
```

If the above invariant's value is `false`, evaluating both its XOR sub-expressions supports the developer in identifying error's rationale, enabling, this way, error fixing.

The constraint concerning the name uniqueness of required interfaces associated to an instance of the `Frame` metaclass, specified by means of the following invariant:

```
(2) context Frame
     inv requiredInterfacesName:
       self.requiredInterface.name->isUnique(n | n)
```

does not support enough the user in identifying interfaces that caused this invariant's violation. This is because in case of many interfaces, a careful study of their names is time consuming, tedious and error prone.

A more appropriate specification, aiding the user in identifying interfaces with the same name is:

```
(3) context Frame
     inv requiredInterfacesName:
       let ri = self.requiredInterface in
       (ri->reject(e | ri.name->count(e.name)=1))->isEmpty
```

In case of an invariant violation, simply evaluating the collection of interfaces having identical names helps the user in navigating the above mentioned interfaces and in model updating (correction), in order to comply with this invariant (see Figure 3.6).

If the uniqueness condition concerns both required and provided interfaces, the specification could be:

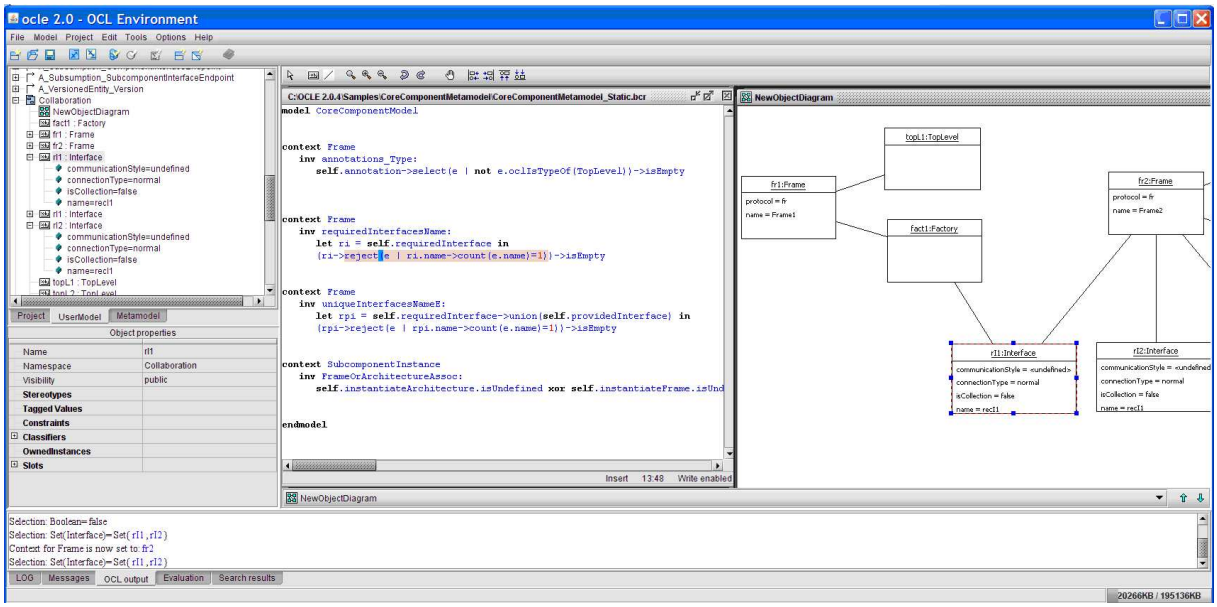


Figure 3.6: Identifying Interfaces that violated the Frame invariant requiredInterfacesName using OCLE

```
(4) context Frame
      inv uniqueInterfacesName:
        let i = self.requiredInterface->union(self.providedInterface) in
        (ri->reject(e | ri.name->count(e.name)=1))->isEmpty
```

Comparing the specifications presented in (3) and (4) with the specification presented in (2), we can notice that the price paid for an easier identification of interfaces violating the `Frame` invariant `requiredInterfacesName` (Figure 3.6) is a more detailed OCL specification.

In case of constraints restraining the type of elements that can be associated as `Frame` annotations, we will adopt a solution similar to the previous one:

```
(5) context Frame
      inv annotations_Type:
        self.annotation->select(e | not e.oclIsTypeOf(TopLevel))->isEmpty
```

Like for specifications presented in (3) and (4), the objective is not restricted to catch invariant violation. We are interested in identifying the rationale of this failure. Evaluating the collection returned by the select operation (5) supports users in identifying the annotations violating this constraint.

Dynamic checking - The constraints aid in preserving the model valid after each operation

This situation is similar to that of an object-oriented application, in which objects are created, destroyed or change their state. The universe of these objects has to be permanently valid. Therefore, we have to take appropriate decisions, enabling us to keep the system in a valid state after each call of a constructor or of a modifier. The strategy adopted will comply with the philosophy "better prevent than cure", therefore, the constraints will be specified mainly by means of pre and postconditions.

As in the static checking case, we will first analyze the constraint between the two unidirectional associations starting from the `SubcomponentInstance` class toward `Frame` and `Architecture`. The XOR constraint (graphically specified on the class diagram) states that starting from `SubcomponentInstance` we will always have a single association, either toward the `Frame` class, or toward the `Architecture` class.

According to the "design by contract" principle [Mey97], the invariant specified for the `SubcomponentInstance` class has to be satisfied by all its instances during their entire life, excepting the time when modifiers are applied on those instances. A first consequence is that `SubcomponentInstance` objects

must be created always using explicit constructors. In case of a single explicit constructor, among its parameters we should have a valid reference toward one of the `Frame` or `Architecture` classes and only one. Therefore, the other reference has to be null (`undefined`). An alternative for the invariant will be:

```
(6) context SubcomponentInstance::SubcomponentInstance(a:Architecture,
                                          f:Frame):SubcomponentInstance
     pre subcomponentInstance_connection:
       (a.isUndefined and f.oclIsTypeOf(Frame)) or
       (f.isUndefined and a.oclIsTypeOf(Architecture))
     post subcomponentInstance_connection_:
       (result.instantiateFrame = f) and
       (result.instantiateArchitecture = a)
```

The OOP philosophy requires the methods `setInstantiateFrame(f:Frame)` and `setInstantiateArchitecture(a:Architecture)` to update only the value of the `associationEnd` transmitted as a parameter. Therefore, in case of these modifiers, the pre and postconditions could be:

```
(7) context SubcomponentInstance::setInstantiateFrame(f:Frame)
     pre instantiate_Frame:
       self.instantiateArchitecture.isUndefined and f.oclIsTypeOf(Frame)
     post instantiate_Frame:
       self.instantiateFrame = f

(8) context SubcomponentInstance::setInstantiateArchitecture (a:
                                                      Architecture)
     pre instantiate_Architecture:
       self.instantiateFrame.isUndefined and a.oclIsTypeOf(Architecture)
     post instantiate_Architecture_:
       self.instantiate Architecture = a
```

If the requirements explicitly mention that in case of a `SubcomponentInstance` object it is possible to remove its association toward a `Frame` instance and to add a new association toward an `Architecture` instance (to switch from a `Frame` to an `Architecture`) or vice versa, the operation supporting this requirement must comply with a precondition similar to that above specified for the explicit constructor. In the body of that method, an appropriate `setInstantiate` method will be called just after setting the other possible reference to null.

Regarding the constraints ensuring the uniqueness of the names of `requiredInterfaces` attached to a `Frame`, we will take into account the fact that this constraint can be broken only when a new interface is added. Therefore, the assertions attached to the operation `addRequiredInterface(i:Interface)` would be:

```
(9) context Frame::addRequiredInterface(i:Interface)
     pre uniqueName:
       i.name.size > 0 and self.requiredInterface.name->excludes(i.name)
     post uniqueName_:
       self.requiredInterface->size = self.requiredInterface@pre->size + 1
       and self.requiredInterface.name->includes(i.name)
```

If changing the interface name after attaching the interface to a Frame is required, then, an appropriate precondition has to be specified for the method `Interface::setName(n:String)`. Also, if explicit constructors initializing the value of the requiredInterface role (toward `Interface` objects) were specified for the class `Frame`, then, appropriate preconditions have to be specified for those above mentioned constructors.

The last constraint concerns the type of instances that can be attached to a Frame as annotations. In this case, the assertion will also be specified by means of a precondition.

```
(10) context Frame::addAnnotation(a:Annotation)
      pre param_s_type:
        a.oclIsKindOf(TopLevel)
```

Conclusion

When specifying assertions, the evaluation time must be taken into account. In case of static evaluation, the assertions are mainly specified by means of invariants. The invariant specification has to help designers in identifying the rationales of invariant failure. The price to be paid is a detailed OCL specification. In case of dynamic evaluation, the assertions are mainly specified by using pre and postconditions. At runtime, assertions specified in OCL are translated in the target programming language. Therefore managing assertions failure must consider both the support of the programming language and the functionalities offered by the IDE used.

3.4 Common 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 usqble technologiesm we need to agree on the model and metamodel tools used in each subproject.

3.4.1 Java/Annotation Tools

Several tools will be used in more than one subproject.

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/

3.4.2 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

Chapter 4

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 first year report.

The workshop emphasis the (intuited) fact that the abstract models of the partners share a common basis on components, services and behaviours. The differences can be seen merely as enrichment rather than concurrency. A common metamodel can therefore be proposed, which can be augmented later to be a proposal for component model interoperability. The cross fertilisation seems also possible at the tool level.

A plan is a sketch for a first step proposal in component abstraction from Java code. We fixed a limited context and objectives to be achieved in one year and several months. The practical implementation will be led in the second year.

Appendix A

More informations on...

see the Project Wiki.

A.1 Workshop Material

Most of the elements are on the project and workshop Wiki.

A.2 Collaborative Tools

Some collaborative tools have been installed to exchange documents.

- Wiki
  `http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:start`
  For storing documents, discussions...
- Version Management. A SVN repository for the project is running at
  `svn://aiya.ms.mff.cuni.cz/econet`
- CoCoME
  `http://agrausch.informatik.uni-kl.de/CoCoME`
  `http://www.cocome.org/`

A.3 Annotated Bibliography

We summarise some useful papers on the subject. These papers can be downloaded on the wiki, project material.

A.3.1 General Papers

- Reverse Engineering: A Roadmap by Hausi Müller at al. [MJS+00]
- The Vienna Component Framework Enabling Composition Across Component Models by Johann Oberleitner et al. [OGJ03]
- A technique for automatic component extraction from object-oriented programs by refactoring by Hironori Washizakia et al. [WF05]
- Program and interface slicing for reverse engineering by Jon Beck et al. [BE93]
- A Simple Method for Extracting Models from Protocol Code by David Lie et al.[LCED01]
- An Intermediate Representation for Integrating Reverse Engineering Analyses by Rainer Koschke et al. [KGW98]

A.3.2 Java Reverse Engineering

- Experiences with the Development of a Reverse Engineering Tool for UML Sequence Diagrams: A Case Study in Modern Java Development Matthias Merdes et al. [MD06]
- Reverse Engineering a Large Component-based Software Product by JM Favre and al. [FED+01]

A.3.3 Patterns Reverse Engineering

- Pattern-Based Reverse-Engineering of Design Components by Rudolf K. Keller et al. [KSRP99]
- An approach for reverse engineering of design patterns by Ilka Philippow et al. [PSRN05]
- Reverse Engineering of Design Patterns from Java Source Code by Nija Shi et al. [SO06]
- A Comparison of Reverse Engineering Tools Based on Design Pattern Decomposition by Francesca Arcelli et al. [AMRT05]
- Automatic Detection of Design Pattern for Reverse Engineering by Hakjin Lee et al. [LYL07]
- Experiments on Design Pattern Discovery by Jing Dong et al. [DZ07]

A.3.4 Code Model Checking, Source code Analysis

- Model-checking Distributed Components: The Vercors Platform by Tomas Barros at al. [BCMR07]
- Source Code Analysis: A Road Map by David Binkley [Bin07]
- Formal verification of software source code through semi-automatic modeling by Cindy Eisner [Eis05]
- Counterexample-Guided Abstraction Refinement by Edmund M. Clarke et al. [CGJ+00]
- The SLAM Project: Debugging System Software via Static Analysis by Thomas Ball et al. [BR02]

A.3.5 Trace Exploration

- A Survey of Trace Exploration Tools and Techniques by Abdelwahab Hamou-Lhadj et al. [HLL04]
- Bandera: extracting finite-state models from Java source code by J.C. Corbett et al. [CDH+00]
- Tool-supported program abstraction for finite-state verification by M.B. Dwyer et al. [DHJ+01]
- Component Recovery, Protocol Recovery and Validation in Bauhaus by Thomas Eisenbarth et al. [EKV05]

A.3.6 Verification of Software Components and Code

- Roadmap for enhanced languages and methods to aid verification by Gary T. Leavens et al. [LAB+06]
- Modular Verification of Software Components in C by Sagar Chaki et al. [CCG+04]
- Predicate Abstraction of ANSI-C Programs Using SAT by Edmund Clarke at al. [CKSY04]

A.3.7 Members publications on the subject

DSRG

- Runtime Support for Advanced Component Concepts by Tomas Bures et al. [BHP+07]
- Modeling Environment for Component Model Checking from Hierarchical Architecture by Pavel Parizeka et al. [PP07a]
- Specification and Generation of Environment for Model Checking of Software Components by Pavel Parizek et al. [PP07b]

- Model Checking of Component Behavior Specification: A Real Life Experience by Pavel Jezek et al. [JKP06]
- Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker by Parízek Pavel, et al. [PPK07]
- Model Checking of Software Components: Making Java PathFinder Cooperate with Behavior Protocol Checker by Parízek Pavel, et al. [PPK06]

OBASCO

- Java Implementation of a Component Model with Explicit Symbolic Protocols by Sebastian Pavel et al. [PNPR05]

Bibliography

[AAA06] 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.

[AMRT05] Francesca Arcelli, Stefano Masiero, Claudia Raibulet, and Francesco Tisato. A Comparison of Reverse Engineering Tools Based on Design Pattern Decomposition. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 262–269, Washington, DC, USA, 2005. IEEE Computer Society.

[BCMR07] Tomás Barros, Antonio Cansado, Eric Madelaine, and Marcela Rivera. Model-checking distributed components: The vercors platform. *Electron. Notes Theor. Comput. Sci.*, 182:3–16, 2007.

[BE93] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 509–518, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[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).

[BHP06] Tomáš Bureš, Petr Hnětynka, and František Plášil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Fourth International Conference on Software Engineering, Research, Management and Applications (SERA 2006), 9-11 August 2006, Seattle, Washington, USA*, pages 40–48. IEEE Computer Society, 2006.

[BHP+07] Tomáš Bureš, Petr Hnětynka, František Plášil, Jan Klesnil, Ondrej Kmoch, and Tomas Kohan and-Pavel Kotrc. Runtime support for advanced component concepts. In *5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*, pages 337–345. IEEE Computer Society, 2007.

[Bin07] David Binkley. Source code analysis: A road map. In *FOSE '07: 2007 Future of Software Engineering*, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[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.

[DHJ+01] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Hongjun Zheng, and Willem Visser. Tool-supported program abstraction for finite-state verification. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 177–187, Washington, DC, USA, 2001. IEEE Computer Society.

[DZ07] Jing Dong and Yajing Zhao. Experiments on design pattern discovery. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 12, Washington, DC, USA, 2007. IEEE Computer Society.

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

[EKV05] Thomas Eisenbarth, Rainer Koschke, and Gunther Vogel. Static object trace extraction for programs with pointers. *J. Syst. Softw.*, 77(3):263–284, 2005.

[FED+01] Jean-Marie Favre, Jacky Estublier, Frédéric Duclos, Remy Sanlaville, and Jean-Jacques Auffret. Reverse engineering a large component-based software product. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, page 95, Washington, DC, USA, 2001. IEEE Computer Society.

[HLL04] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of trace exploration tools and techniques. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004.

[JKP06] Pavel Ježek, Jan Kofroň, and František Plášil. Model checking of component behavior specification: A real life experience. In Luis Barbosa and Zhiming Liu, editors, *International Workshop on Formal Aspects of Component Software (FACS 2005)*, volume 160 of *Electronic Notes in Theoretical Computer Science*, pages 197–210, Macao, Macao, 2006.

[KGW98] R. Koschke, J.-F. Girard, and M. Würthner. An intermediate representation for reverse engineering analyses. In *WCRE '98: Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, page 241, Washington, DC, USA, 1998. IEEE Computer Society.

[KSRP99] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 226–235, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[LAB+06] Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for enhanced languages and methods to aid verification. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 221–236, New York, NY, USA, 2006. ACM Press.

[LCED01] David Lie, Andy Chou, Dawson Engler, and David L. Dill. A simple method for extracting models for protocol code. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 192–203, New York, NY, USA, 2001. ACM Press.

[LYL07] Hakjin Lee, Hyunsang Youn, and Eunseok Lee. Automatic Detection of Design Pattern for Reverse Engineering. In *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*, pages 577–583, Washington, DC, USA, 2007. IEEE Computer Society.

[MD06] Matthias Merdes and Dirk Dorsch. Experiences with the development of a reverse engineering tool for uml sequence diagrams: a case study in modern java development. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 125–134, New York, NY, USA, 2006. ACM Press.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. International Series in Computer Science. Prentice Hall, 2 edition, 1997. ISBN 0-13-629155-4.

[MJS+00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 47–60, New York, NY, USA, 2000. ACM Press.

[Moo00] Michael Moors. Consistency checking, rose architect, spring issue. Technical report, Rational, April 2000.

[OGJ03] Johann Oberleitner, Thomas Gschwind, and Mehdi Jazayeri. The vienna component framework enabling composition across component models. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 25–35, Washington, DC, USA, 2003. IEEE Computer Society.

[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.

[PP07a] 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.

[PP07b] Pavel Parízek and František Plášil. Specification and generation of environment for model checking of software components. In *Formal Foundations of Embedded Software and Component-Based Software Architectures, FESCA 2006*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 143–154. Elsevier B.V., 2007.

[PPK06] Pavel Parízek, František Plášil, and Jan Kofroň. Model checking of software components: Making java pathfinder cooperate with behavior protocol checker. Technical Report 2, KSI MFF UK, 2006.

[PPK07] Pavel Parízek, František Plášil, and Jan Kofroň. Model checking of software components: Combining java pathfinder and behavior protocol model checker. In *30th IEEE/NASA Software Engineering Workshop (SEW-30)*, pages 133–141. IEEE Computer Society, 2007.

[PSRN05] Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, and Sebastian Naumann. An approach for reverse engineering of design patterns. *Software and System Modeling*, 4(1):55–70, 2005.

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

[SO06] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[WF05] Hironori Washizaki and Yoshiaki Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Sci. Comput. Program.*, 56(1-2):99–116, 2005.

List of Figures