ECONET Project

COMMON COMPONENT METAMODEL - SPECIFICATION

Version 1.1

Pascal ANDRE[1] Vladiela PETRASCU[2]

June 9, 2008

supported by

[1]COLOSS - LINA - FRE CNRS 2729- 2, rue de la Houssinière, B.P.92208, F-44322 Nantes Cedex 3, France
[2]LCI - Computer Science Research Laboratory, Universitatea BABES-BOLYAI Mihail Kogalniceanu nr. 1 RO- 400084 Cluj-Napoca, Romania

Summary

This work is leaded in the context of the Egide-sponsored ECONET Project Nr `16293RG` entitled, "Behaviour Abstraction from Code, Filling the Gap between Component Specification and Implementation".

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

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

The metamodel part is shared by the two processes and constitues the foundation API (Application Programming Interface) for component model processing. A main issue of a component metamodel is to answer to the problem of handling several component models to get a generic reengineering process. In order to provide a convenient component model API, a metamodel specification is necessary to serve as reference guide.

A first sketch was drawn during the Econet Workshop of Prague in 2007. Experimentations were leaded and draft versions have been produced until the Econet Workshop of Nantes in 2008 where a first release has been validated after discussions. This document summarises the result of these works as a **Common Component MataModel** version 1.1.

The specification detailed in this document is structured using packages to separate concerns (basic elements, component model, instances and model management...). Each main package is presented in a separate chapter and the specification in based on UML diagrams, informal comments and restrictions by means of Well-Formed-Rules (WFR) written as OCL constraints.

A tool support for modeling, verification and code (API) generation is provided.

Acknowledgements The authors would like to thank Egide for its financial support of this `16293RG` ECONET project [1].

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

Contents

Chapter 1

Introduction

In the context of the ECONET project Nr `16293RG`, the partners are involved in the contribution to reverse engineering of component models fram Java code. One goal is to develop techniques for extraction of abstractions from code (including some component interface description) and for the verification of abstractions against the code.

The general project organisation has been drawn during the first project workshop in prague in september 2007 [ACPR07].

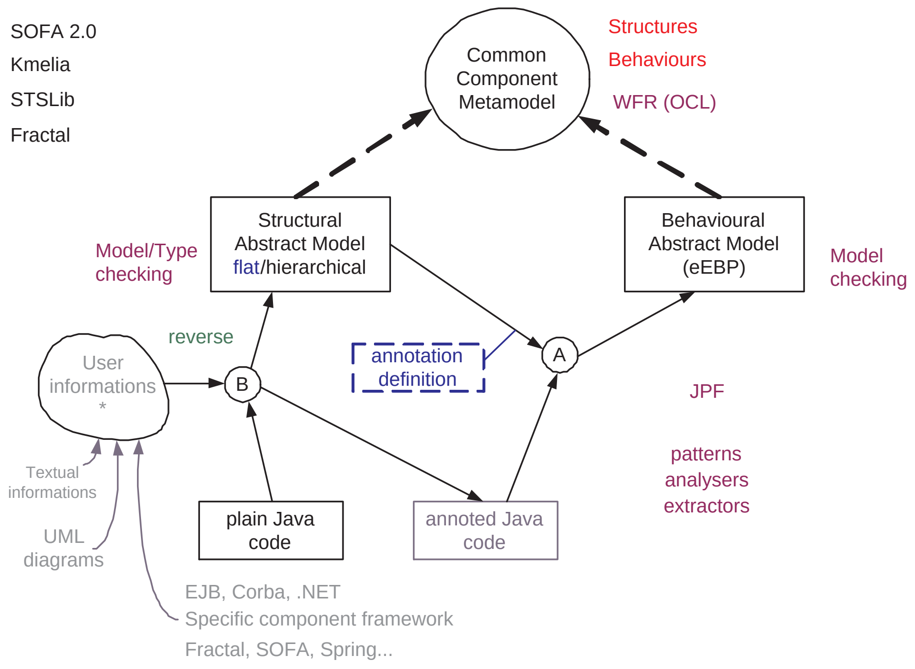


Figure 1.1: Econet Architecture: final version

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

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

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

A first sketch was drawn during the Econet Workshop of Prague in 2007 [ACPR07]. Experimentations were leaded and draft versions have been produced until the Econet Workshop of Nantes in 2008 where a first release has been validated after discussions. This document summarises the result of these works as a **Common Component MetaModel** version 1.1.

The document is organised as follow. In chapter 2 we address methodological issues to specify the metamodel (principles, structuration, asumptions).

The specification is structured using packages to separate concerns (basic elements, component model, instances and model management...) (see chapter 2) . Each main package is detailed in a separate chapter (from chapter 3 to chapter 8) and the specification in based on UML diagrams, informal comments and restrictions by means of Well-Formed-Rules (WFR) written as OCL constraints.

A tool support for modeling, verification and code (API) generation is provided. Chapter 9 describes the main features of tools support and experimentation.

The remaining of this chapter overviews the motivations, the history, the different versions that have been produced or referenced, and the current state of the metamodel.

1.1 Motivations

Roughly speaking the goal of the reengineering processes is to abstract component paradigms from plain or annotated Java code, assuming that the Java code is somewhere a component implementation. The component paradigms are structured in a component model and several component models are targeted including SOFA, Kmelia, KADL but also Fractal, Tracta, Corba... For the sake of simplicity these will be qualified as **concrete** component metamodels.

Of course the above goal locates over these concrete component metamodels and focus on *Generic re-engineering techniques and tools*. This requirement lead to a **common component metamodel (CCMM)** which is model independent and gather a subset of models concepts, postponing specific concepts to concrete model mappings. It must include the relation between component model and implementation code (java annotations). It must be specified in such a way that specification properties can be checked and a Metamodel API can be refined or implemented.

1.2 History and starting points

Having the current version of a model is never sufficient to understand it correctly, we need the motivations, reasons and explanation that lead to it. Therefore we present an historical point of view in this section.

The work on a common metamodel started during the Econet Workshop of Prague in september 2007. Several versions have been produced until march 2008. They are the starting points for the validation process that occurded in the Econet Workshop of Nantes in may 2008. We also recall here other sources of inspiration.

1.2.1 Econet Workshop of Prague in 2007

The participants were quickly convinced of the necessity of having a common component metamodel to handle multiple target models (each component system has its own means for specifying models but most of component systems are similar (black-box component, provided/require services, nesting)...) and also code mapping. Here is a summary of the discussions. ‘

1. Requirements ([ACPR07] p. 27)

    (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).

2. A simplified Sofa metamodel ([ACPR07] p. 31)
   It should correspond to all our component models (with different names).

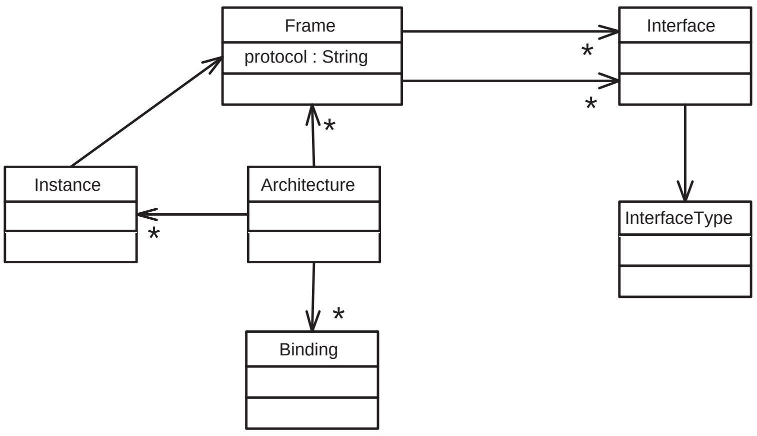


Figure 1.2: Sofa: short metamodel

3. Discussions on mapping concepts ([ACPR07] p. 37)

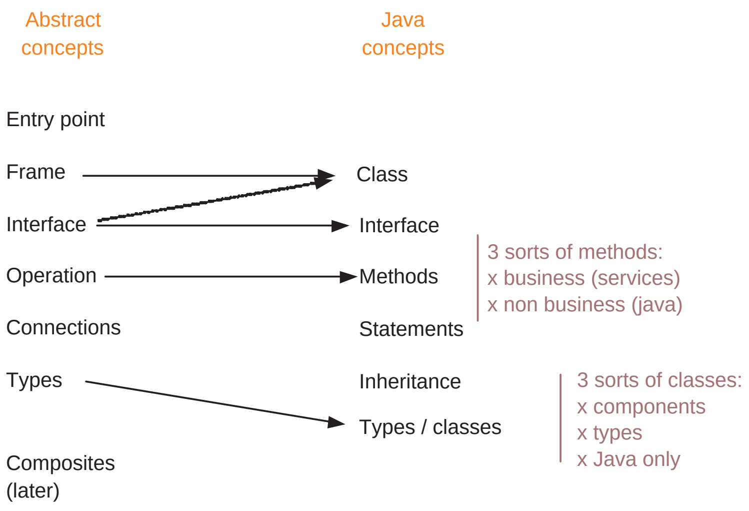


Figure 1.3: Mapping concepts

4. fast comparison of the three abstract models (SOFA, KADL, Kmelia) ([ACPR07] p. 37) 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)

Study of 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)

5. Common component metamodel ([ACPR07] p. 39)
   It 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.

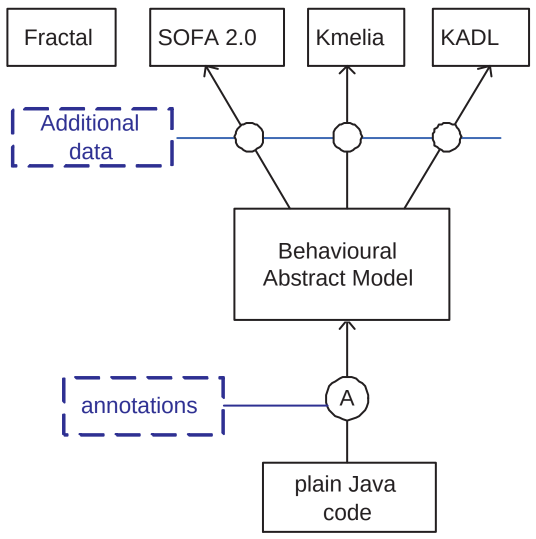


Figure 1.4: Common Component MetaModel

6. Annotations ([ACPR07] p. 40-41, p. 47-48)

7. Metamodel Abstraction Subproject ([ACPR07] p. 52-54) the CoreComponent Metamodel is a simple model grouping the common features of most component-oriented modeling languages [BHP06]. Moreover, in Figure 1.5, only the elements referred in assertions are represented including some constraint definitions for model verification especially 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 1.5), can be expressed by means of the following invariant:

```
(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.

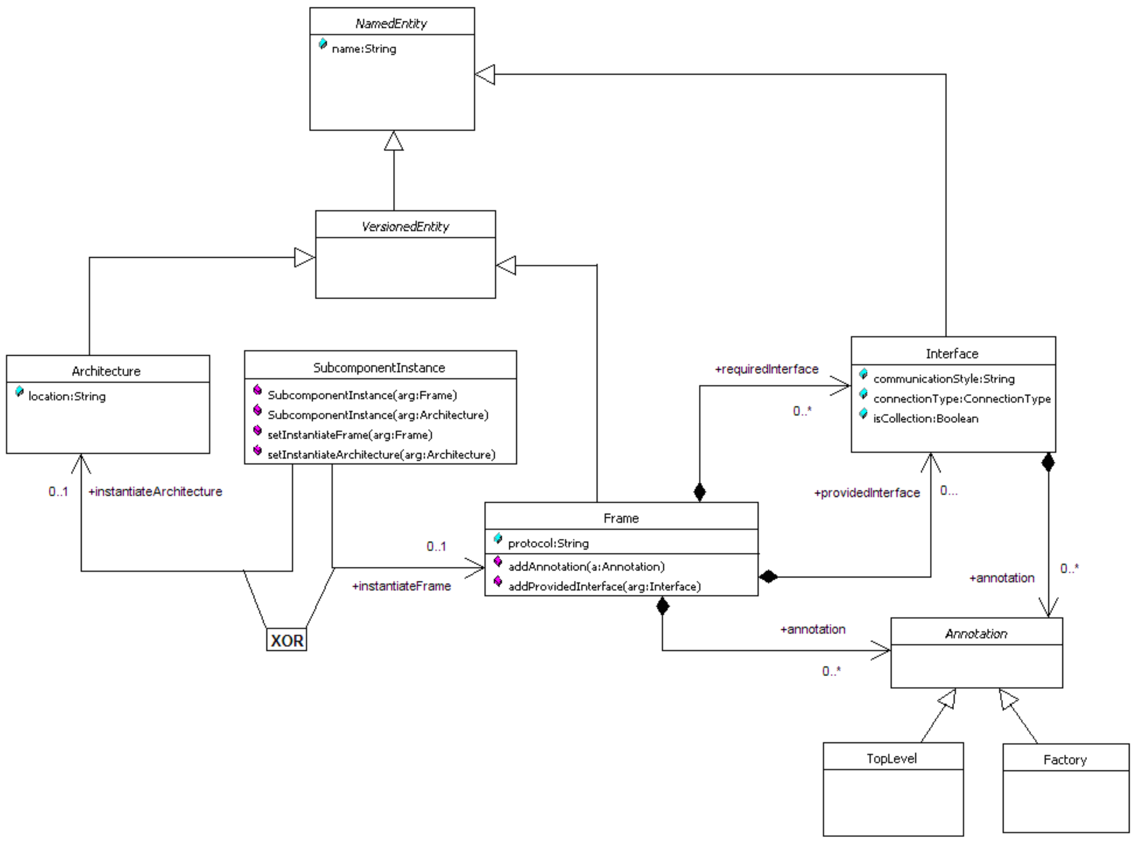


Figure 1.5: A part of the CoreComponent Metamodel

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
```

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

```
(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.

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
```

1.2.2 Annotations

The new definition of Java annotations if provided in appendix A.

1.2.3 Concrete Metamodels

The SOFA metamodel and the Kmelia metamodel can be found on the wiki od SVN repository.

1.2.4 Abstract Metamodels

This first attemps are detailed in section 1.3.

1.2.5 Normative or specific model

We also looked at other and various sources covering two extreme solutions. The Object Management Group families of modeling languages (Meta Object Facility MOF and Unified Modeling Language UML) define normative models covering a wide range of concrete models, they include many concepts and many levels of generalisation (abstraction) for modelling concepts. The Eclipse Modeling Framework Project proposes a restricted model called EMF/Ecore which aims to be more manageable in practice by modelling casetools.

- Ecore from EMF project
  http://eclipse.org/emf
  http://www.eclipsecon.org/2005/presentations/EclipseCon2005_
  .   Tutorial11final.pdf
  http://www.research.ibm.com/journal/sj/453/leroux.html

- OMG UML 2.1 (UML 2.0, UML 1.5)
  UML1.5 http://www.omg.org/docs/formal/05-04-01.pdf
  UML2 infrastructure http://www.omg.org/docs/formal/07-02-06.pdf
  UML2 superstructure http://www.omg.org/docs/formal/07-02-05.pdf
  MOF2.0 http://www.omg.org/docs/formal/06-01-01.pdf
  OCL2.0 http://www.omg.org/docs/formal/06-05-01.pdf

This is a inspiration source for finding the core element and relations and to name them. For example we could define special profiles.

1.3 Versions

In this section we overview several versions of the Common Metamodel.

CMM metamodel PA - november 2007

This metamodel was proposed after the workshop of Prague. Strengths are:

+ There is a clear separation between the core (structure) and behaviour parts.
+ This is a simple model (few concepts).
+ It borrows a usual component terminology.
+ It was inspired by PragueŠs Workshop discussions.
+ It has been used in a prototype for Process B.

Weaknesses points are:

- It is largely inspired from Kmelia.
- Few constraints have been specified.
- No instances and mondel mangaement have been handled.

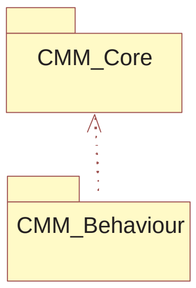


Figure 1.6: Structure of the Component MetaModel 1.0

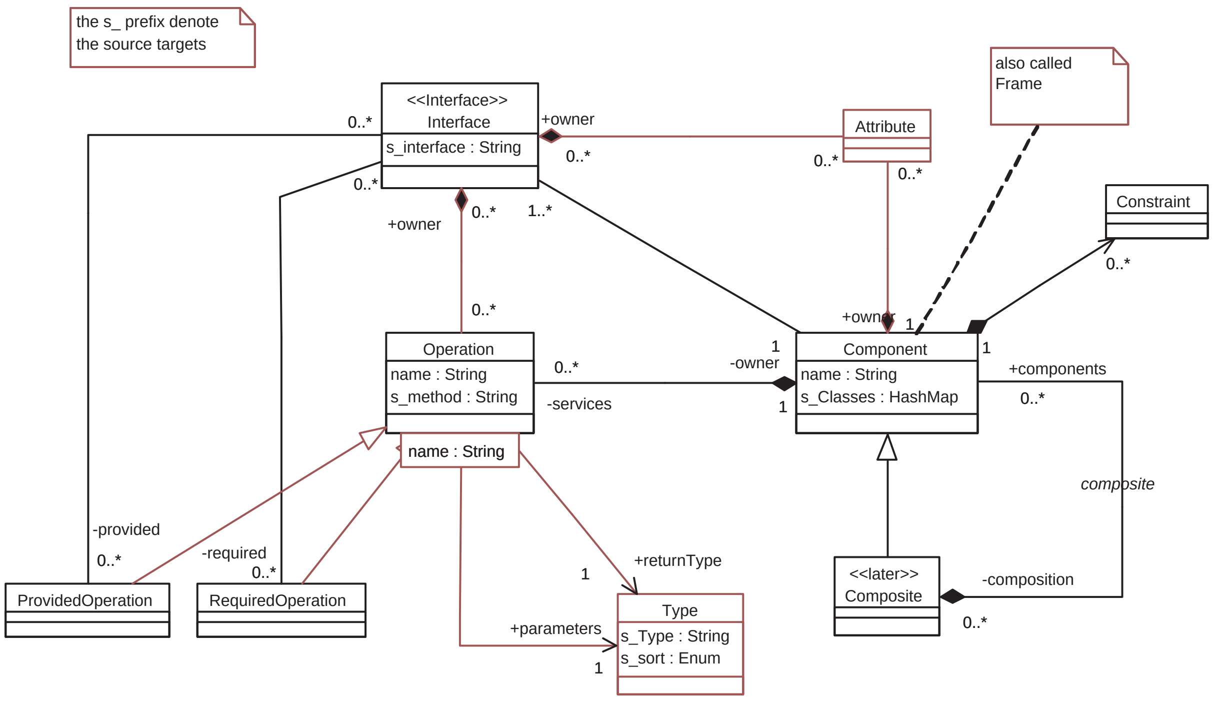


Figure 1.7: Component package of the Component MetaModel 1.0

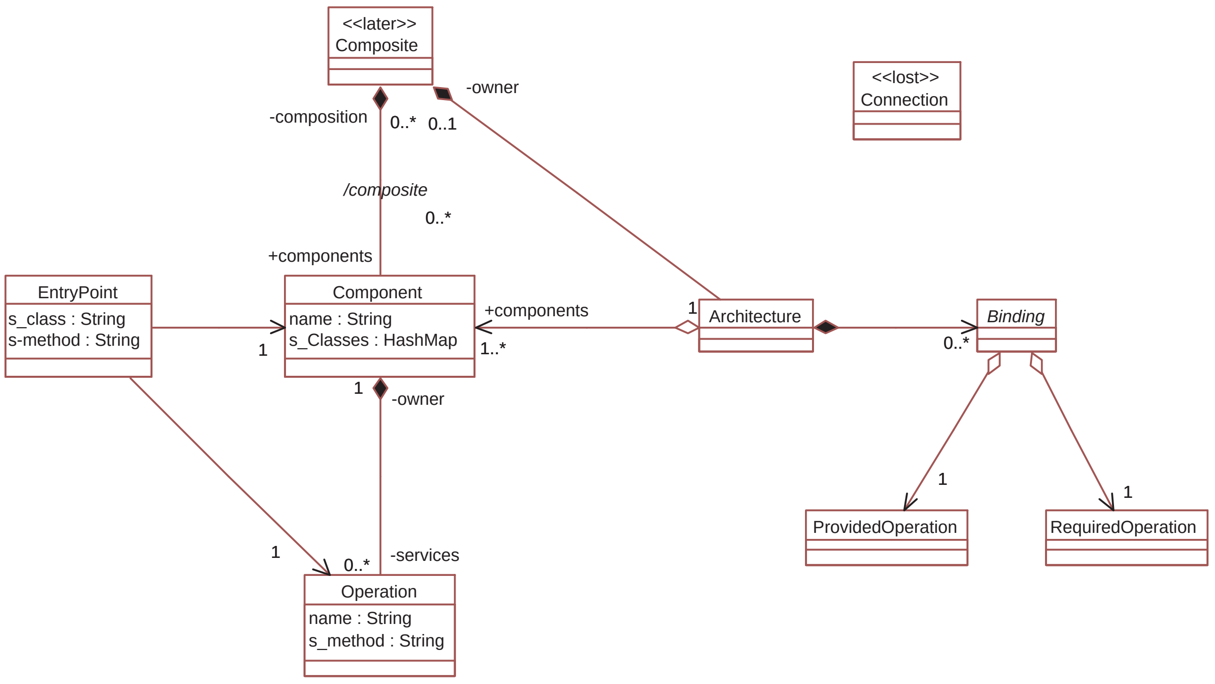


Figure 1.8: Architecture package of the Component MetaModel 1.0

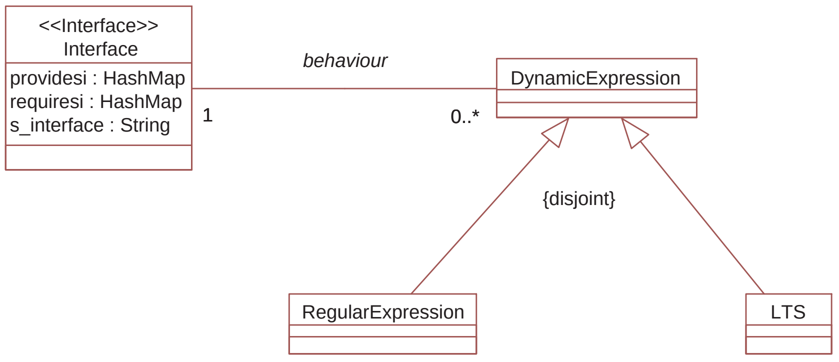


Figure 1.9: Behaviour package of the Component MetaModel 1.0

CCMM_1.0_ecore VP - march 2008

Strengths are:

+ This is a mixin model.

+ It is based on Ecore (instrumentalisable).

Weaknesses points are:

- It is too simple, there are no architectures.

- No constraints have been specified.

- No description is provided.

Another model was produced for the workshop in Nantes, that was inspired from CCMM_1.0 (section 1.3).

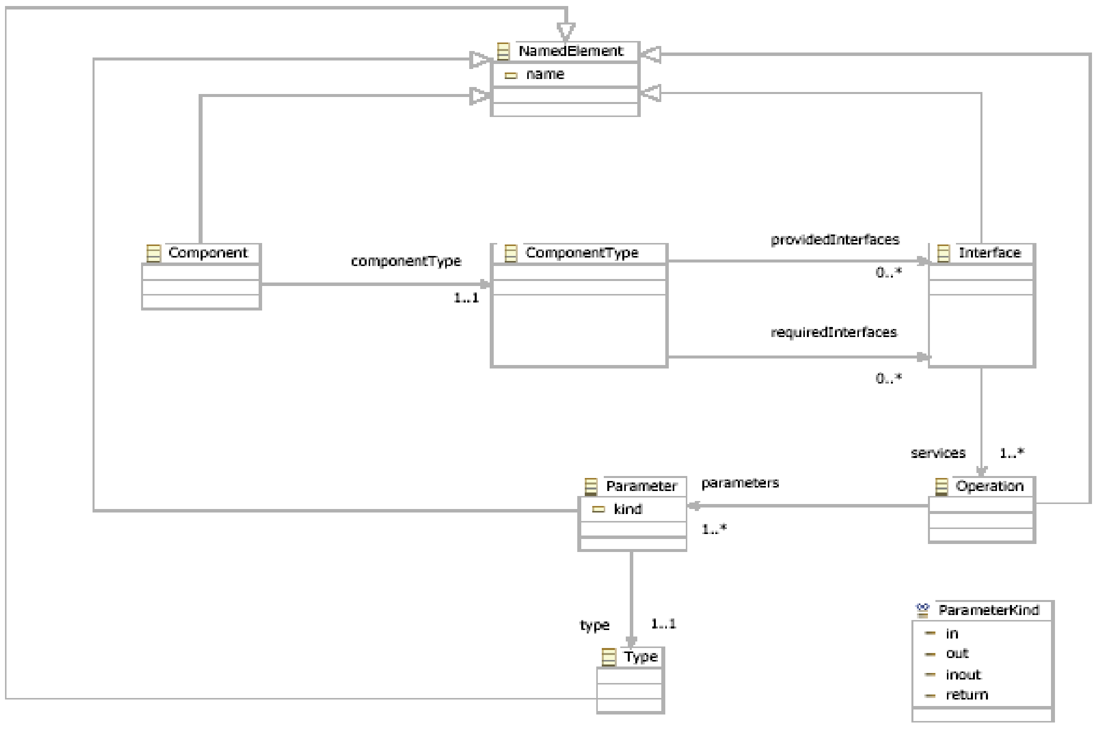


Figure 1.10: Core Component MetaModel 1.0 - Ecore

CMM metamodel PH from another project - march 2008

Strengths are:

+ It is has been validated in an existing project
+ The model is quite simple and complete (minimal).
+ It handles instances and model management.
+ The model is inspired from Sofa and Fractal.

Weaknesses points are:

- It is too specialised.
- It uses a specific component terminology.
- No constraints have been specified.
- No separation of concerns is provided (melted).

CCMM_1.0 PA - march 2008

This is a new version of CMM 1.0 which is more compliant with the various sources of informations. It serves as basis for the discussions and validation. Only its validated version will be presented here.

This new version of CMM 1.0 was initiated in april 2008. It is a new attempt for a common metamodel (from the above sources) characterised by its originality (not published), widebroad (various source of inspiration), customisable (from a generalised root). But it was a draft version that was to be discussed, updated, completed and validated. Its main features are:

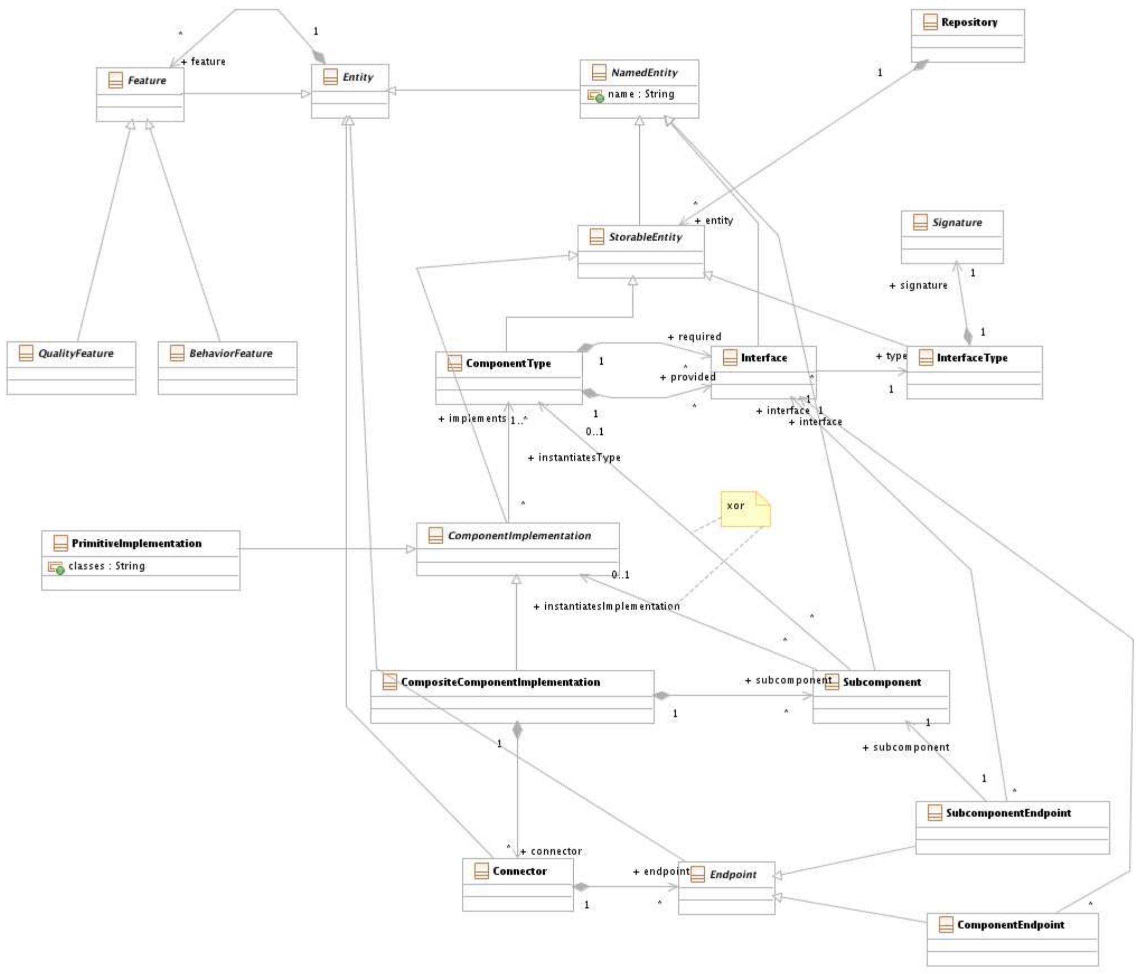


Figure 1.11: Component MetaModel

- Layered Model
  - Abstract the commonalities
  - Core and extensions
- Separation of concerns
  - Core Component Model
  - Behaviour Modelling
  - Instance Management
  - Model Management and Annotations
- Modelling Process
  - Generalisation: various concepts, notations
  - Constraints, comments, examples

Summary

```
Totals:
 7/19 Logical Packages
 79 Classes
```

```
CCMM\_Basic
   Types
   Elements
   Classifiers
   BasicBehaviour
CCMM\_Core
   CCMM\_Components
   CCMM\_Architecture
   Annotations
```

```
CCMM\_Behaviour
   CCMM\_Behaviour\_LTS
   CCMM\_Behaviour\_RE
   CCMM\_Behaviour\_Basic
CCMM\_Instance
CCMM\_CodeMapping
CCMM\_ModelManagement
CCMM\_Addins
```

Questions remained

- Big model ?
- Implementation issues
  - The core part
  - Extend existing frameworks (UML, MOF, EcoreĚ)
  - Reduced model
  - Layered Implementation

That were debated during the workshop.

1.4 Discussions and Validation

The discussions were based on the experimentations leaded and the draft versions produced before the Econet Workshop of Nantes in 2008. Some tracks are:

- Modelling concepts and organisation
- Conflicting concepts
- Modelling issues
- Debugging
  - General/specialised
  - Incompleteness
  - Inconsistency
- Fulfill the draft version
  - Add Constraints
  - API requirements

Several concepts have been qualified, refined or removed ; constraints have been added. A first version has been validated after discussions which is specified in the remaining document.

Chapter 2

Modelling principles and specification structure

In this chapter we address methodological issues to specify the metamodel.

2.1 Modelling Principles

We followed some principles to build the new specification. Concepts are modelling elements.

Principle 2.1.1 (abstraction) *We try to factorise the commonalities in shared concepts (use of the generalisation/specialisation relation).*

We use abstraction when two concepts are different but share similar commonalities, when a concept has different concrete representations.

Principle 2.1.2 (separation of concerns) *We prefer to organise the specification by packages instead of a flat model. Packages can be replaced without changing the overall struture. The main drawback is the dispersal of modelling elements in several views.*

With packages, structural features are separated from behavioural ones, we also separate the core model from model management, component implementation, annotation, optional features.... By abstraction, commonalities are separated from specific features.

Principle 2.1.3 (generality/extensibility) *Based on several sources we tried to be as general as possible in order to include other concrete component models. Moreover the root packages can be grafted in a wider metamodel.*

The generality principle is tied to principles 2.1.2 and 2.1.1. Because generality is obtained by abstraction and extensibility is linked to some separation of concerns. We also use *stereotypes* to qualify concepts.

Principle 2.1.4 (minimality) *We tried to limit the number of concepts.*

The minimality principle should be a compromise with the above generality principle 2.1.3 because we try to avoid unnecessary concepts but stay extensible to include others later.

Principle 2.1.5 (specification) *We separate specification issues from implementation issues. In the latter we reduce the number of concepts, relations and constraints. This allow a general wider specification model and a more restricted implemented metamodel.*

Following these concepts, the CCMM Release 1.1 is as follow.

- A layered model that separate several concerns:
  - Basic layer : common concepts that overlap components (to be connected with usual core metamodels (UML, EMF).

  - Common Component layer (an abstraction of what we find in general component models)
  - Specific Component layer (for concrete models) Many WFR will apply to concrete model layers especially to restrict the element combinations.

- We try to make it complete and consistent.
- It should be original but generalisable and adaptable. We added some basic and core concepts (elements, typesĚ) that we find in most of abstract and concrete models.
- It is enriched by WFR and constraints to enforce some definitions.
- API requirements are taken into account.

2.2 Modelling Issues

This is a short summary of discussion points and answers.

1. Represent Java concepts (like JMI model)
   NO
   Handling a Java model (such as JMI) would be time-cost expensive and should evolve with Java versions (normative or not). Moreover the tool would be programming language dependent. *In fact we modelled a subset of Java concepts in the special `CCMM_CodeMapping` package to illustrate one way to represent the link between the component model and its simplified Java implementation.*
   Java mapping is thus represented by special attributes in the component model. A special prefix is given to the mapping attributes that allow to fix and process them (for example the sole component model is obtained by a model transformation that remove these attributes).

2. Represent model management
   Partial
   Model management is required for model computation but it is not a part of the component metamodel. There is a special package `CCMM_Model_Management` for it. A first detailed version was designed in version 1.0. It has been simplified after the workshop of Nantes, based on the LCI proposal.

3. Represent component instances
   Partial
   There is a special package `CCMM_Instance` for it. It is useful for the CoCoME case study description and verification.

4. Represent annotations

   YES
   There are two way to represent annotations:

   (a) Attributes added to the component concepts definition.
   In order to distinguish the component feature attributes from annotation attributes we prefixed the latter by `s_` (source).

   (b) Special Package, Code Modelling and Relations.
   The special and optional `CCMM_CodeMapping` package to illustrate that way.

   For implementation reasons and sake of simplicity we decided to choose the first solution. One drawback is that the Abstract Component Model is *polluted* by these special attributes.

5. Represent non functional requirements
   NO.

6. Represent Ecore
   NOT EXACTLY
   but inspired by Ecore and UML2.

7. Comments and cleaning
   This has been mainly done during the workshop.

   - Remove UML qualified associations, aggregation relations
   - Consider EMF composition relations (multiplicities)
   - Check the names and informal semantics.

8. Constraints
   YES. Mandatory to check automatically some properties of the component model using tools.

Additionally to those of UML we define stereotypes to specify modelling elements or packages and introduce concerns. Here is an unlimited list of them.

- «`spec`» indicates that the element is defined in specification step only.
- «`lost`» indicates that the element is not taken into account.
- «`later`» indicates that the element will be introduced later.
- «`concrete`» indicates that the element will be refined in a concrete component model.
- «`primitive`» indicates that the element is considered as primitive for a component model.
- etc.

2.3 Conflicting concepts

In order to solve the conflicts (except noun conflicts), we proposes to draw a specialisation hierarchy.

1. Interface
   Can be a (restricted) *Classifier*, a *NamedElement* (Sofa, KADL) or simply an *Element* (Kmelia). They can be separate between Provided/Required or not. We made a specialisation hierarchy.. In other approaches we have also ports.

2. Operation
   As a behavioural feature denoting some functional computation with or without dynamic features. Can be simply an *Operation* (Sofa, KADL) or a complex entity (Kmelia) We took *NamedElement*.

3. Protocol
   Can be simply associated to a component (Sofa, KADL), an interface or a service (Kmelia)

4. Service
   Can be simply an *Operation* (Sofa, KADL) or a complex entity (Kmelia) We took *Operation*.

5. Constraints/Predicate/Properties
   Can be used to write assertions, classify conceptsĚ They are set in a special and optional package.

6. Pre/post conditions
   Set in operations as optional features.

7. Architecture/Assembly Ű Connectors-Bindings
   We defined an architecture type that denotes patterns of assembling. Connectors are simply bindings. The question is about what we bind : this can be interfaces or services. A CCMM should accept boths. I tried to make it more abstract using EndPoints and specialised endpoints. An endpoint has a target which is either an interface or an operation (service).

2.4 Modelling Constraints

In order to check the metamodel with metamodelling tools, several constraints have been introduced:

- Remove qualified associations and aggregation relations.
- Avoid derived associations and attributes.
- Remove constraints on specialisation relations.
- Replace multiplicity `0..1` by `1` for associations, if possible.
- Ensure reacheability of model elements by giving sufficient composotion relations.
- Avoid deep specialised concepts.
- Unique names.

2.5 Specification Structure

The specification is organised by packages and is described as so using the Rational Rose tool. Each package is detailed by one chapter in the specification document except the (additional) `CCMM_Addins` and `CCMM_CodeMapping` packages, which are defined in the same chapter.

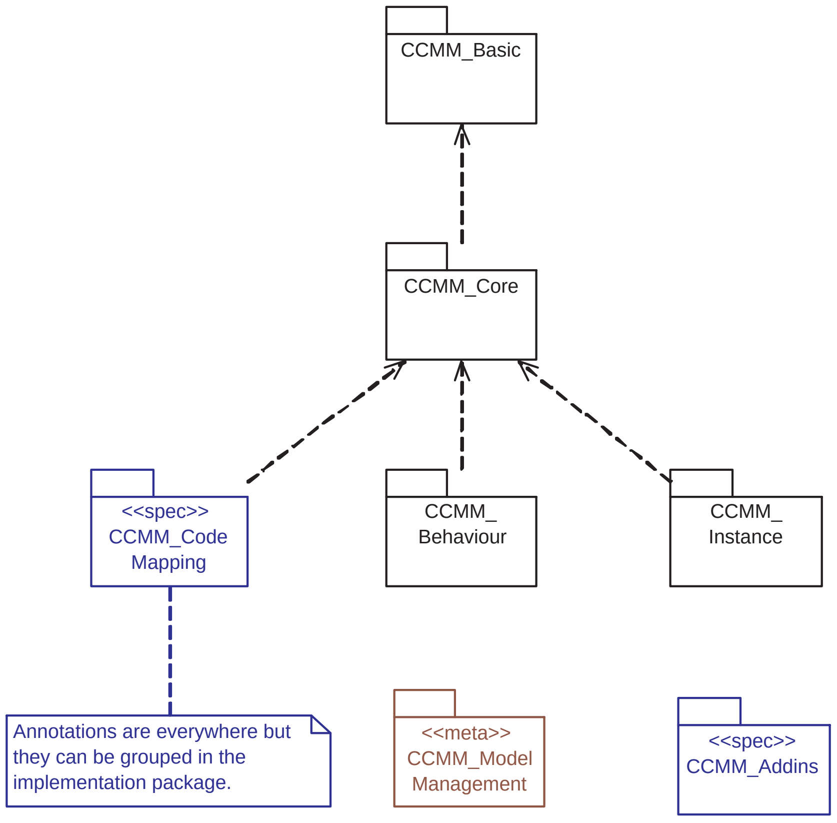

CCMM_Basic
CCMM_Core
<<spec>> CCMM_Code Mapping
CCMM_ Behaviour
CCMM_ Instance
Annotations are everywhere but they can be grouped in the implementation package.
<<meta>> CCMM_Model Management
<<spec>> CCMM_Addins

Figure 2.1: Structure of the Common Component Metamodel

- chapter 3: package `CCMM_Basic` - basic elements of a component model (model elements, types, values...),
- chapter 4: package `CCMM_Core` - components and architectures models and also annotations,
- chapter 5: package `CCMM_Behaviour` - dynamic behavioural features of components and architectures,

- chapter 6: package `CCMM_Instance` - component instance management,
- chapter 7: package `CCMM_ModelManagement` - model handling and repository,
- chapter 8: secondary elements
  - package `CCMM_CodeMapping` - one way to link component (abstract) models and component implementations,
  - package `CCMM_Addins` - additional concepts of the models.

For each package the specification schema is the following:

- Diagrams: overview of the model
- Definitions: definitions of concepts
- Constraints: natural language and OCL expressions
- Examples: illustrations
- Comments: comments on the specification document and process

Chapter 3

CCMM_Basic

In this chapter we specifiy the `CCMM_Basic` package.

`+++ TODO: *VP* +++`

3.1 Overview

3.2 Types SubPackage

3.3 Elements SubPackage

3.4 Classifiers SubPackage

3.5 BasicBehaviour SubPackage

Chapter 4

CCMM_Core

In this chapter we specifiy the `CCMM_Core` package.

`+++ TODO: *VP* +++`

4.1 Overview

4.2 Components SubPackage

4.3 Architecture SubPackage

4.4 Annotations SubPackage

Chapter 5

CCMM_Behaviour

In this chapter we specifiy the `CCMM_Behaviour` package.

5.1 Overview

The `CCMM_Behaviour` package describes the model elements that have a behavioural part. Since there are several approaches to represent behaviours (regular expressions, state machines, logics...) and also different models in each approach we organise it with specialised packages. For instance we choosed the Labelled Transition System (LTS) and the regular expression (RE) formalisms used in the concrete component languages (SOFA, KADL, Kmelia).

Notes that the `CCMM_Behaviour_RE` and `CCMM_Behaviour_LTS` are `«spec»` packages, which means that they are not implemented in the CCMM API.

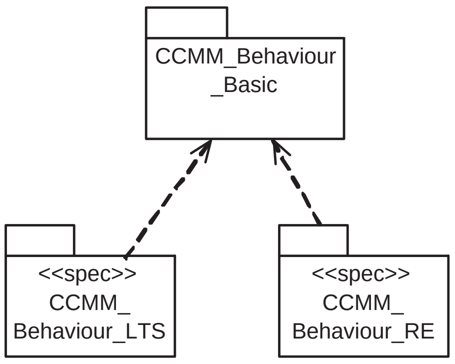


Figure 5.1: Structure of the `CCMM_Behaviour` package

5.2 Behaviour_Basic SubPackage

The `CCMM_Behaviour` package describes the commonalities of behavioural elements.

5.2.1 Diagrams

The UML diagram for the basic behavioural elements is given in figure 5.2.

5.2.2 Definitions

Every component *dynamic element* is associated to a *dynamic expression*.

- A `DynamicElement` is a concept of the component model.

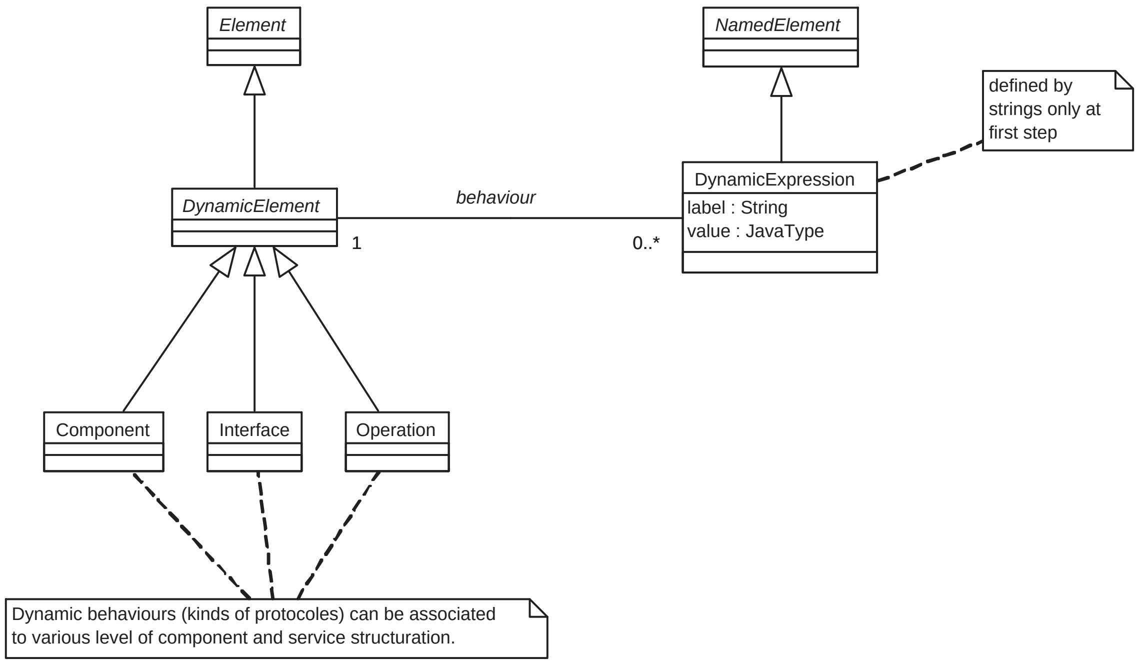

Element
NamedElement
defined by strings only at first step
DynamicElement
behaviour
1
0..*
DynamicExpression
label : String
value : JavaType
Component
Interface
Operation
Dynamic behaviours (kinds of protocoles) can be associated to various level of component and service structuration.

Figure 5.2: `CCMM_Behaviour_Basic` package

- `DynamicExpression` is given at first stage by a string (the `label`) which can be analysed later in a concrete behavioural formalism.

5.2.3 Constraints

Constraint 5.2.1 (behaviouralUniformity) *For a given component model, the dynamic expressions are described in the same language.*

It means that

- the subclasses of `DynamicExpression` are disjoint specialisations (`{disjoint}` specialisation constraint).
- for each dynamic element of a component model, the dynamic expressions have the same type (subclasses of `DynamicExpression`).

A component model is assumed to be an architecture defined in the repository.

```
context Repository
  inv behaviouralUniformity :
  self.architectureTypes.contains.allDynamicElement.behaviour.oclType->size() = 1
```

We assumed a navigation `allDynamicElement` on architectural elements that provides the dynamic elements. We also assumed that the subclasses of the abstract class `DynamicExpression` have only one deep-height level.

5.2.4 Examples

The contents of the label is be processed to get any dynamic description. This is an "abstract" level it is refined in the subpackages.

5.2.5 Comments

The subpackages are given only for information and the behavioural language is delegated to the concrete component model formalism.

5.3 Behaviour_RE SubPackage

In the category of regular expression formalism we retain only the SOFA `BehaviorProtocol` language. No special metamodel is provided and the correctness of the string `label` if delegated to the BP analyser and type checker.

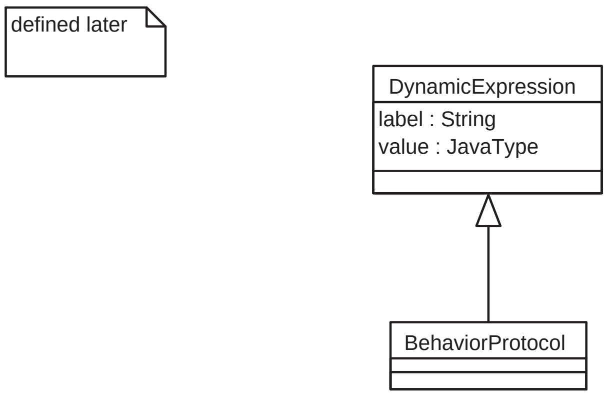


Figure 5.3: `CCMM_Behaviour_RE` package

5.4 Behaviour_LTS SubPackage

The core LTS concepts are common to all LTS languages. Here is an informal and simplified description.

- `LTS` is the container (graph) of state and transitions.
- `State` is a vertex in the graph defined by a name.
- `Transition` is an oriented edge in the graph defined by a source state, a target state and a label.
- `Label` is a textual description.

The LTS languages vary on the representation of these concepts and merely on the additional features:

- A label can be a simple name or an expresssion including
  - a functional computation or an action (in some action language)
  - a guarded expression,
  - events or communications (messages for examples)
- A transition can be a simple transitions or complexe ones
  - multiple sources or target
  - defined by a subgraph (composition)
- A state can be a simple state or a complex entity with
  - a functional computation or action (in some action language)
  - ports or access,
  - hierarchical nested subgraphs (one or more)...
- A LTS can
  - distinguish classes of states (initial, final, error, hierarchical...),
  - distinguish classes of transitions (initial, final, error, communications, hierarchical...),

  - be associate to some context (*e.g.* qualifiers, states, transitions),
  - handle time constraints,
  - subtyping...

In this category of LTS expression formalism we mainly capture the concepts of the Kmelia metamodel and tried to define them in a general schema. This model has be restructured to capture the KADL language of dynamics.

5.4.1 Diagrams

The UML diagrams for the LTS behavioural elements are given in figure 5.4 and figure 5.5.

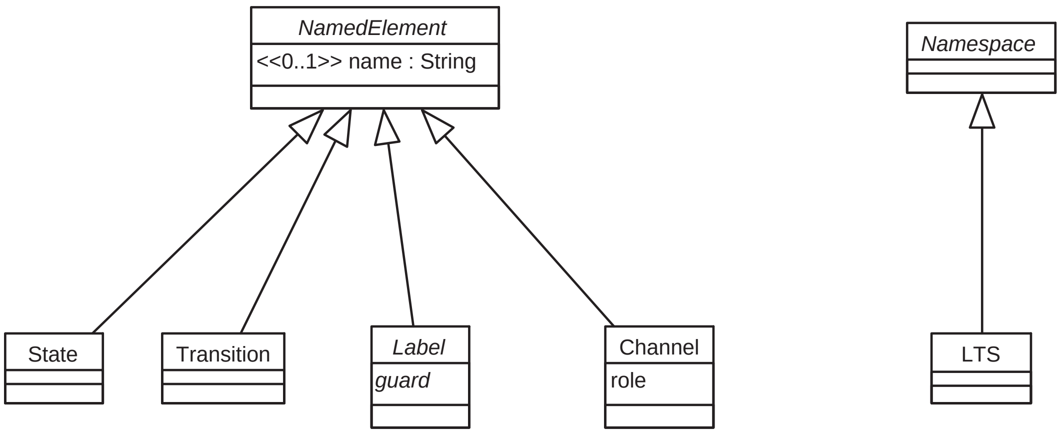


Figure 5.4: `CCMM_Behaviour_LTS` package 1/2

The parts specific to Kmelia are dark red colored in the diagram of figure 5.5).

5.4.2 Definitions

Every component *dynamic element* is associated to a *dynamic expression*.

- A `LTSExpression` is defined by a LTS that provides its `label` expression and value.
- A `LTS` is a `Namespace` defined by an non-empty set of states (in this set there is one initial state and at least one final state) and a set of transitions.
- States, transitions and labels and channels are named elements (`NamedElement`).
- Labels are guarded actions and can be defined by composition of other labels (*e.g.* sequential, parallele composition).
- Actions can be elementatry (say functional computations) or communication actions.
- Communication actions use some channel to send or receive messages. Pairwise communication are between a sender and a receiver. Broadcast actions occur with one sender and multiple receiver.
- Channels are named elements (`NamedElement`) with some role for the communications (channels can be process identifiers, ports, ...).

Some Kmelia specific concepts have been represented.

- A `Service` is a kind of operation whom dynamic is defined by a LTS.
- Kmelia states can be annoted by services (optional service calls).
- Kmelia transitions can be annoted by services (mandatory service calls).
- Kmelia communication actions include service calls or result. Service call can be synchronous.

5.4.3 Constraints

Constraint 5.4.1 (mandatoryServiceCall) *A transition is labelled with a mandatory service call or (exclusively) with plain label.*

```
context Transition
  inv mandatoryServiceCall:
  self.mandatoryAnnotation->isEmpty() xor label self.->isEmpty()
```

There are other constraints.
`+++ to write +++`

5.4.4 Examples

See Kmelia [AAA06, AAA07] or KADL [PNPR05].

5.4.5 Comments

This is only a partial representation of the languages.

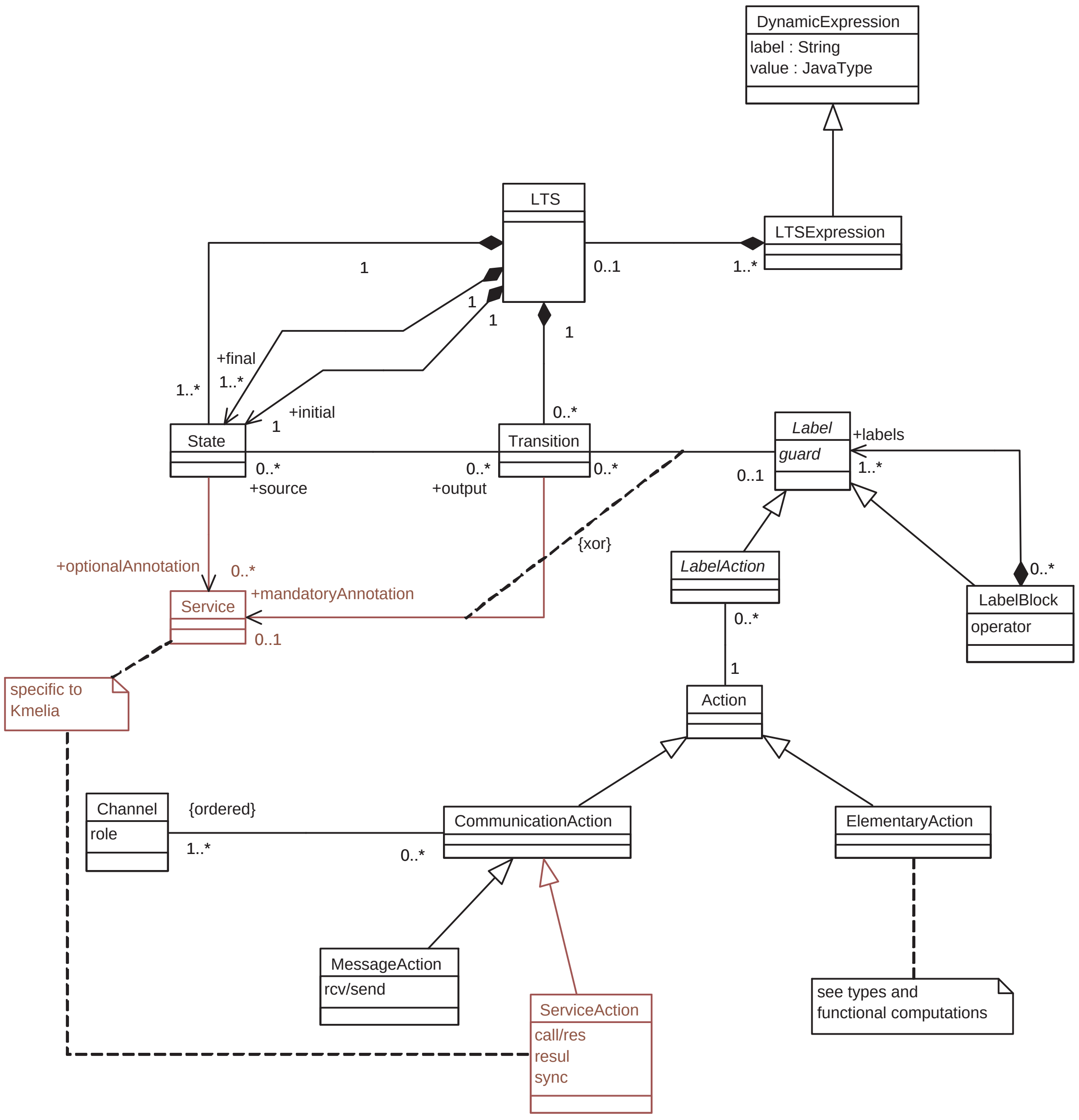


Figure 5.5: `CCMM_Behaviour_LTS` package 2/2

Chapter 6

CCMM_Instance

In this chapter we specifiy the `CCMM_Instance` package.

`+++ TODO:` *`VP`* `+++`

6.1 Overview

6.2 Types SubPackage

6.3 Elements SubPackage

6.4 Classifiers SubPackage

6.5 BasicBehaviour SubPackage

Chapter 7

CCMM_ModelManagement

In this chapter we specifiy the `CCMM_ModelManagement` package.
`+++ TODO: *VP* +++`

7.1 Diagram

7.2 Definition

7.3 Constraints

7.4 Examples

7.5 Comments

Shall we add a `topLevel` architecture(s) ?
for example to model some constraints on it.

7.6 CCMM_ModelManagementOld SubPackage

This is a «`lost`» package. It includes a more complete but too complex description of model management. Also it has not been validated.

Chapter 8

CCMM_Others

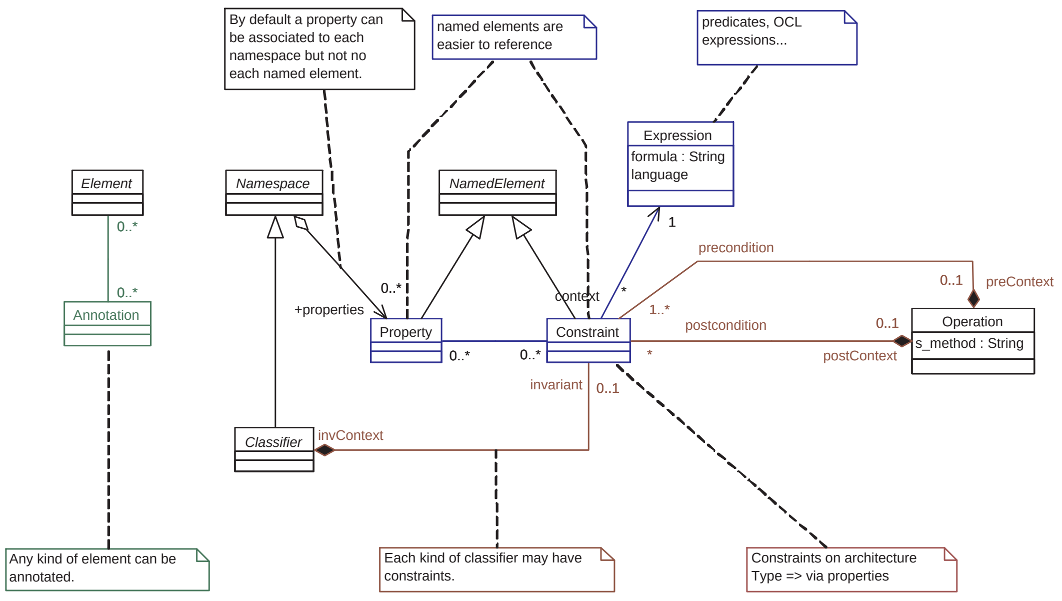
In this chapter we specifiy the `CCMM_Addins` and `CCMM_CodeMapping` package.

These packages were included having in mind a possible evolution of the metamodel.

8.1 Addins Package

The `CCMM_Addins` package include special features. For instance these features are related to constraints, assertions, properties and annotations.

8.1.1 Diagrams



By default a property can be associated to each namespace but not no each named element.
named elements are easier to reference
predicates, OCL expressions...
Expression
formula : String
language
Element
Namespace
NamedElement
0..*
0..*
Annotation
+properties
0..*
Property
Constraint
context
*
1
1..*
precondition
0..1
preContext
postcondition
0..1
Operation
s_method : String
postContext
*
0..*
0..*
invariant
0..1
Classifier
invContext
Any kind of element can be annotated.
Each kind of classifier may have constraints.
Constraints on architecture Type => via properties

Figure 8.1: `CCMM_Addins` package

8.1.2 Definitions

+++ to write +++

- `Constraints` are predicates or boolean expression.

- `Assertions` are a special usage for constraints (invariants of classifiers and pre/post conditions of operations).
- `Properties` are a way to qualify model elements using some optional constraints. *In some way a stereotype is a property but in the metamodel.*
- `Annotations` are a way to enrich model elements.

8.1.3 Constraints

Only multiplicity constraints apply for instance.

8.1.4 Examples

Kmelia uses both assertions and properties. Assertions enforces the definition of functional properties for components and architectures. Properties define special kind of concepts. For example a Kmelia *behaviour* in is a special kind of Kmelia *service*.

8.1.5 Comments

The assertions are defined for classifiers and operation in their usual meaning as in OCL, Eiffel, Z, VDM or B...

8.2 CodeMapping Package

The `CCMM_CodeMapping` package specifies the mapping between a component model and a component implementation realised in Java.

8.2.1 Diagrams

A simplified Java metamodel is proposed (dark red colored in the diagram of figure 8.2). The mapping is represented by (blue colored in the diagram of figure 8.2) associations.

8.2.2 Definitions

The (abstract) component model is the one defined in the previous chapters.

The program model is a small subset of Java concepts including:

- structuring: Java packages
- types : Java types, classes and interfaces
- features: attributes, operations and signatures
- relations: extends and implements

with their usual meanings.

8.2.3 Constraints

No matter for instance.

8.2.4 Examples

The CoCoME benchmark establishes such a mapping of concepts.

8.2.5 Comments

The mapping is defined in its usual menaing in set theory (a set of couple elements) rather than a model transformation. This mapping is defined from the annotation definitions (appendix A).

A mapping to another programming language can be done in a similar way.

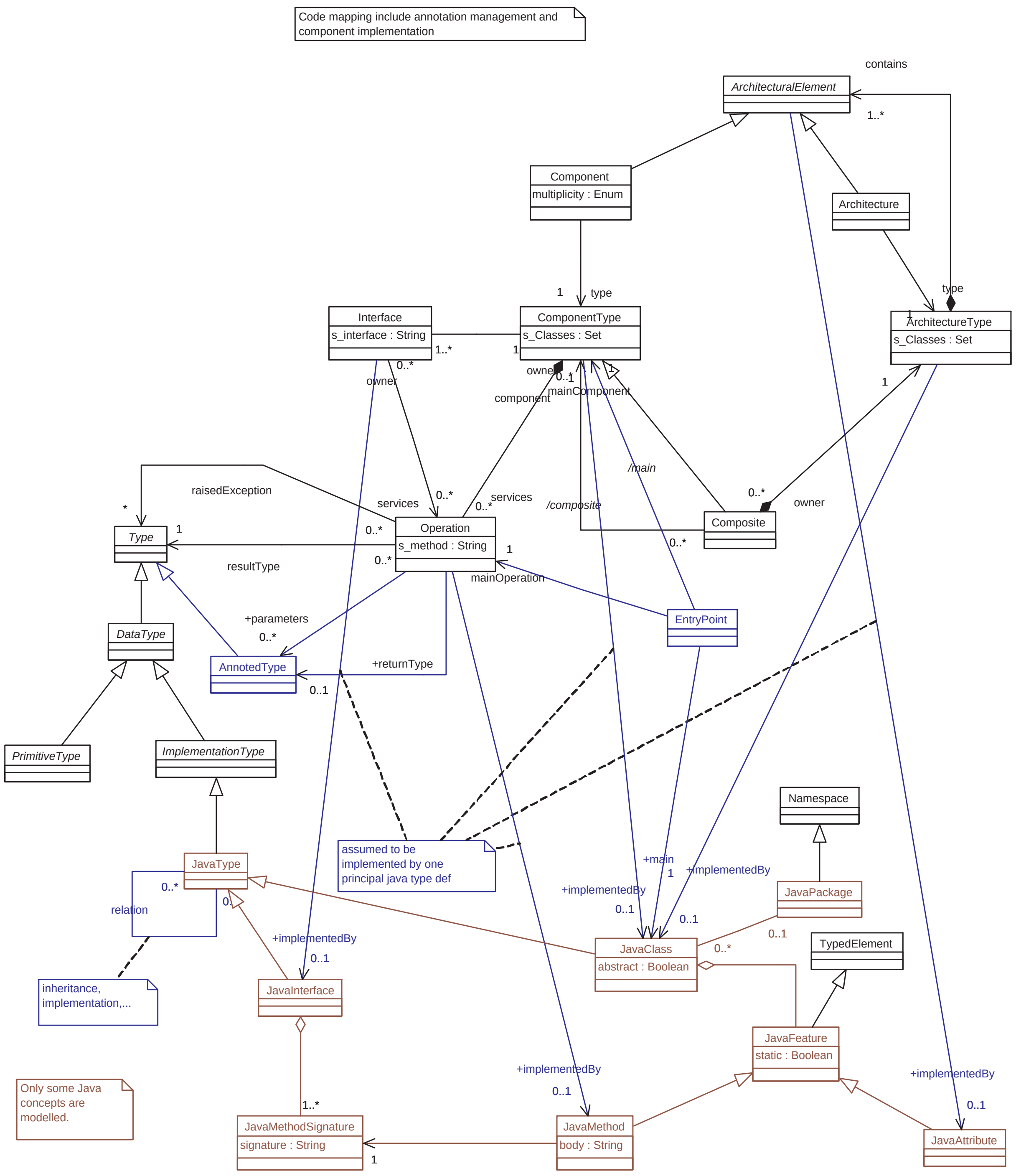


Figure 8.2: `CCMM_CodeMapping` package

Chapter 9

Tool Support and Experimentation

`+++ TODO: VP +++`

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

Section 9.1 we overview the existing tool support for modeling, verification and code (API) generation. Section 9.2 relates the exementations we led before the workshop and the ongoing ones.

9.1 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:

- OCLE `http://lci.cs.ubbcluj.ro/ocle/`
- EMF `http://www.eclipse.org/modeling/emf/`
- oAW `http://www.openarchitectureware.org/`
- 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

9.2 Experimentations

`+++ to write +++`

9.2.1 Experimentations with OCLE

9.2.2 Experimentations with EMF

9.2.3 Experimentations with oAW

9.2.4 Experimentations with ATL

9.2.5 Ongoing Experimentations

Chapter 10

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.

Bibliography

[AAA06] Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking Component Composability. In *5th International Symposium on Software Composition, SC'06*, volume 4089 of *LNCS*. Springer, 2006.

[AAA07] P. André, G. Ardourel, and C. Attiogbé. Defining Component Protocols with Service Composition: Illustration with the Kmelia Model. In *6th International Symposium on Software Composition, SC'07*, volume 4829 of *LNCS*. Springer, 2007.

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

[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.

[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.

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

[PNPR05] Sebastian Pavel, Jacques Noye, Pascal Poizat, and Jean-Claude Royer. Java Implementation of a Component Model with Explicit Symbolic Protocols. In *4th International Symposium on Software Composition, SC'05*, volume 3628 of *LNCS*. Springer, 2005.

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

Appendix A

Annotations

In this appendix chapter we provide the Java definition of the annotations.

A.1 Java Annotations

Component - Class Relation

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

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

Entry points

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

  /**
```

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

  String[] componentName();
}

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

  /**
   * @return the array (one entry per annotation source) containing component
   *   Names for which the annotated method provides the initialization.
   *
   */
  String[] componentName();
}
```

Interfaces

Provided

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

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

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

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

   String[] modelIfaceName();

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

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

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

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

Required

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

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

Business elements

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

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

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

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

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

List of Figures