
FORMAL FOUNDATIONS FOR REASONING ABOUT EVOLUTION

Maurice Glandrup, Arend Rensink
Department of Computer Science, University of Twente,
P.O. Box 217, 7500 A.E. Enschede, the Netherlands
E-Mail: {glandrup,rensink}@cs.utwente.nl

Abstract:

Designing software systems is difficult. Designing systems that are capable of evolving is even more difficult. Often the system evolves in an unforeseen direction. Practice shows that software systems grow in small evolutionary steps. An evolution usually influences the behavior and the structure of the system. However, it is not desired that the evolution influences one or more modules of the system that are functionally or logically not related to the evolution; the behavior and structure of these modules should be preserved.

In this position paper, we introduce an algebra as a formal foundation for reasoning about evolution. The algebra can be used to express changes in the behavior and structure of the design when it evolves. The aim is to eventually use the algebra to give decision support to the developer during the evolution of a software system.

This position paper can be grouped in: Formalization of AOSD, Design Support for AOSD, Visions and Directions

This work has been funded by the EU-funded IST Project 1999-14191 EASYCOMP

1 INTRODUCTION

Evolution of a system can mean that the system gets new or more (complex) functionality, or that parts of the system are restructured e.g. to make future extensions or derivations easier. Every evolutionary step implies a change to the system. This change must be integrated, which can affect modules of the system that are not evolving. Especially in complex systems, it is not always clear what modules are influenced and what the nature of the influence is when the system evolves. With nature of the influence we mean a behavioral or structural change in the design.

To find changes in a design caused by an evolution, we designed an algebra that expresses these changes. By making the change of the evolution explicit, we can use the algebra to reason about the changes in the behavior and structure of the design. The aim is to eventually use the algebra to give decision support to the developer during the evolution of a software system.

This paper is structured as follows: first we illustrate the problem with an example. Then we discuss the evolution algebra and show how it can be applied on the example. We end with conclusions and a short description of related and future work.

2 PROBLEM DESCRIPTION

Practice shows that software systems grow in small evolutionary steps. In the evolution of a system, not every step is always foreseen. We can say that the step:

- implements concerns of the system that were not or barely recognized before,
- improves not correct functioning concerns.

Usually there are more solutions to let a system evolve. This also accounts for integrating evolution with the existing system; tradeoffs must be made in deciding what the best solution for integrating is. For instance, the new identified concerns can enforce design decisions or design structures that do not fit well into the existing design. In particular, the new concerns can be crosscutting on the behavior and structure of the existing concerns.

With this, we mean that the behavior and/or structure of the existing system are influenced in many different places rather than in a single module.

We categorize evolution of software (or integration of solutions) in the following groups (or combination of groups):

- Introducing new components
There are requirements that introduce new functionality in the existing system. The new functionality is integrated with the existing functionality
- Substituting components
A component is replaced by another component that has the same interface as the component that it replaces. The functionality of the replacing component is improved.
- Removing obsolete components
Removing unused or obsolete software parts can be seen as a form of evolution. The removal can reduce the complexity and risk-of-failure of the system

For every category one or more techniques exist to do the integration.

To summarize the problems:

Recognizing and grouping the evolution of the system is difficult. It is not always clear where to integrate and what techniques are necessary. It is also not clear what the influence of the evolution is on the behavior and structure of the non-evolving parts of the system.

In the following section, we give an example that we use in this paper.

3 EXAMPLE

The example that we use to illustrate the problem is The Examination Assistant (TEA). This system assists examiners in assessing exams. TEA compares the answers of the exam with the answers given by a student. Based on the comparison, the exam made by the student is graded. This is the Phase0 evolution of TEA. In the next phase of the evolution, TEA needs a more advanced grading mechanism. The figure below illustrates the situation:

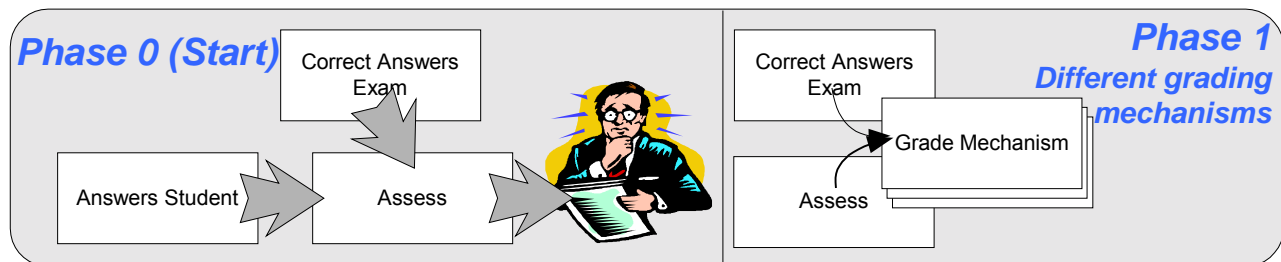


Figure 1. The Examination Assistant

We first discuss the Phase0 of TEA, since this is the starting point for the problem case.

In the problem domain analysis of the Phase0, we recognized the concern *Assessing*. TEA can handle two types of questions, namely multiple choice and open questions. For the Phase0, we started with no explicit requirements for grading; grading was therefore not identified as an important concern¹. The *Assessing* concern is decomposed in three components: Assess, Correct Answers Exam and Answers Student.

The activities in assessing an exam can be denoted as follows²:

Assessment = $\Sigma(\text{compare answers} + \text{grade answer student})^*$

The resulting design is depicted in Figure 2.

¹ For the sake of simplicity, we only discuss the part of the design that involves the grading mechanism, since we want to illustrate design problems concerning the integration of a new grading mechanism in the TEA design.

² Note that we only discuss the grading activities of assessing an exam and not the comparison of the answers.

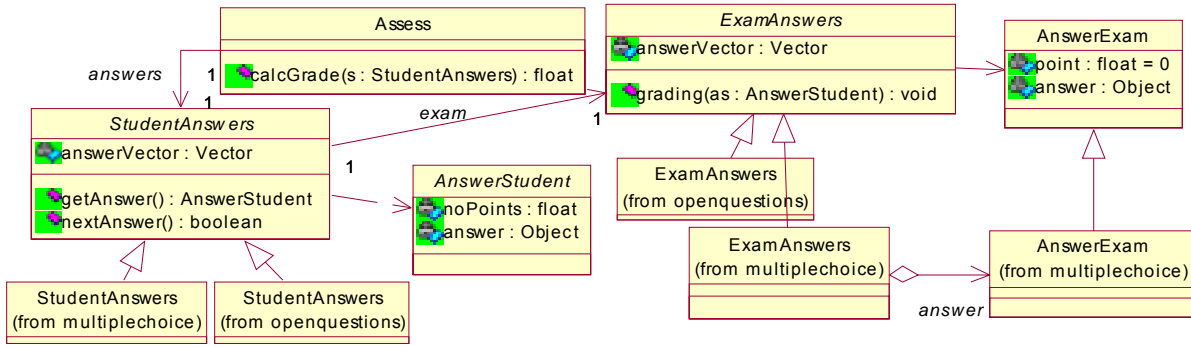


Figure 2. Phase 0: The Examination Assistant

The part of the design that is the scope for this paper operates as follows:

A student already filled in an exam. The *Assess* object is called to calculate the grade of the student. For every answer, a grade is given in terms of a number of points. The *ExamAnswers* objects contain all answers to the questions and the number of points that can be given to it. Every student answer is assigned a number of points. The *Assess.calcGrade()* method adds all points and returns the given grade.

The problem is that grading in Phase0 is interwoven in the design. The grades and their interpretation are woven into the *Answers Exam* and *Assess* components (see right part of Figure 1 & the classes in the design of Phase0 in Figure 2). New requirements such as the capability of handling different grading policies/country and the concept of a correction factor, indicate that grading is a concern of TEA.

Phase1 of TEA is concerned with improving the grading mechanism. In this example, grading is promoted from a less important concern to an important one. A new analysis of problem domain leads to a design model for grading called *GradingMechanism*.

We now integrate the grading mechanism design model with the TEA design model in such a way that the structure and behavior of the Phase0 TEA is not changed fundamentally. In the next section we illustrate the integration while discussing the evolution algebra.

4 EVOLUTION ALGEBRA

To support developers in reasoning about the evolution of their software, we developed an algebra of system evolution. This support starts as soon as the developer knows what improvements have to be made. The components of the algebra are: 1) Algebra for Object-Oriented Design Models, 2) Consistency Rules and 3) Evolution Operators. We discuss the components of the evolution algebra in this section.

4.1 OBJECT-ORIENTED DESIGN MODELS

The foundation of the evolution algebra is a formal framework to express object oriented design models. The framework combines structural and behavioral aspects.

- The *structural part* consists of the interfaces in the design, including their signatures. We distinguish sets of *system* interfaces *Sys* (those offered by the system to its environment), *internal* interfaces *Int* (those defined by the system and invisible from the environment) and *environment* interfaces *Env* (those used by the system and assumed to be offered by the environment). The signature, Σ , declares methods with parameter and return types for all of these interfaces: formally,

$$\Sigma: I \rightarrow (\text{Meth} \rightarrow \text{Type}^* \times \text{Type}) \quad (\text{where } I = \text{Sys} \cup \text{Env} \cup \text{Int})$$

- The *behavioral part* consists of classes with their association and inheritance relations. Formally, the classes given by a set *C*; inheritance is expressed by a binary relation *Inh* over *C*; and associations are given by a set *Ass* = *C* × *Role* × *I*; i.e. each $(c,r,i) \in \text{Ass}$ consists of a class name *c*, a *role* name *r* and an interface name *i*. The most important part of the behavior is given by the *declaration* of the classes, which is a function Δ that fixes their methods, with parameter and return types just as for

the interfaces, but also for each method the *roles* it partakes in and the *protocol* it will issue on other classes when itself is invoked. Formally,

$$\Delta: C \rightarrow (\text{Meth} \rightarrow \text{Type}^* \times \text{Type} \times 2^{\text{Role}} \times \text{Prot})$$

We will not elaborate on the precise nature of the protocol Prot.

Structure and behavior together give rise to a model M described by a tuple, as follows:

$$M = \langle \text{Sys}, \text{Env}, \text{Int}, \Sigma, C, \text{Inh}, \text{Ass}, \Delta \rangle$$

The relation between classes and interfaces is implicit: every class $c \in C$ is assumed to have as an interface each $i \in I$ such that $\Sigma(i)$ and $\Delta(c)$ defined the same methods with the same parameter and return types. The algebra is completed with *consistency rules* for the model to guarantee the correct use of (among others) *system* and *internal* interfaces (rule **i**), roles and associations (rule **ii**), protocol (rule **iii**). An example of such a rule is: “for all declared methods, the protocol only uses declared methods”, formally:

$$\forall c \in C: \forall (m, T_p, T_r, A, P) \in \Delta(c): \forall (c', m') \in P: c' \in C, m' \in \text{dom}(\Delta(c'))$$

Continuing with the example, the design model depicted in Figure 2 can be formally denoted as follows:

Note that this diagram does not distinguish classes and interfaces. In the formalisation we will append the names with I (for interface) or C (for class). The structural part is captured by:

Sys: AssessI, StudentAnswersI, StudentAnswersI_{mc}, ...

Int: AnswerExamI, AnswerExamI_{mc}, ...

Env: none

Σ : AssessI \rightarrow calcGrade(StudentAnswersI): float

StudentAnswersI \rightarrow getAnswer(): AnswerStudentI; nextAnswer(): boolean; getExam(): ExamAnswersI

ExamAnswersI \rightarrow grading(AnswerStudentI): void

The behavioral part is given by:

C: AssessC, ExamAnswersC, ExamAnswersC_{mc}, ExamAnswersC_{oq}, AnswerInstanceC, ...

Inh: (StudentAnswersC_{mc}, StudentAnswersC), (StudentAnswersC_{oq}, StudentAnswersC), ...

Ass: (StudentAnswersC, exam, ExamAnswersI), (AssessC, answers, StudentAnswersI),
(ExamAnswersC_{mc}, answerinstance, AnswerInstanceI)

An example of a declaration is:

Δ : AssessC \rightarrow calcGrade(StudentAnswersC): float,

(*roles*) answers, exam

(*protocol*) (StudentAnswerI.nextAnswer, StudentAnswerI.getAnswer, StudentAnswersI.getExam,
ExamAnswerI.grading, AnswerStudentI.getPoints)⁺

The last two lines give the protocol issued by an invocation of calcGrade. This can be understood as follows.

First the answer of the student is obtained. Through the exam association, the number of points for the answer is calculated and set in the AnswerStudent by the grading method. After that the number of points is retrieved from the AnswerStudent and used in the calculation of the final grade. Subsequently the next answer is retrieved; this process continues until nextAnswer returns false.

With the formalization of the protocol that a method can issue, it is possible to search for dependencies with other classes and their methods. If one of the classes and/or methods in the protocol changes, then the protocol of the class also has to change which means that a method of the class that issues the protocol must be changed.

For a complete overview of the algebra and its consistency rules, we refer to respectively Appendix A. and Appendix B.

4.2 EVOLUTION OPERATORS

We already mentioned common forms of evolution in the problem statement. We define them respectively as: *extension*, *substitution* and *extraction*. The formalizations of these operators build on the consistency rules of the algebra to guarantee consistency.

Case 1: Extension (\otimes)

Extension can be used to integrate new functionality with the existing functionality. The notation is as follows:

$$M_{\text{new}} = M_{\text{orig}} \otimes C$$

Case 2: Substitution (⊙)

Substitution is a possible solution to adapt software without altering the existing software. The notation is as follows: $M_{new} = M_{orig} \odot c$

Case 3: Extraction (⊖)

Extraction is the opposite of extension. The notation is: $M_{new} = M_{orig} \ominus c$

We elaborate on *extension*:

Intuitively we define extension as follows: extension means extending the original design with new classes and/or methods. The new classes and methods can not influence the behavior and structure of the original classes and methods. The formal definition for *extension* $M_{new} = M_{orig} \otimes c$ is:

$\Delta_{new}(C_{new})$	The following rules are about new classes
$Env_{new} \supseteq Env_{orig}$	The environment dependency can increase
$Sys_{new} \cup Int_{new} = Sys_{orig} \cup Int_{orig} \cup \{i_{new}\}$	The interface is extended with the extension
$C_{new} = C_{orig} \cup \{C_{new}\} :$	
i) $\forall c \in C_{orig}: \Delta_{new} \supseteq \Delta_{orig}$ $\forall l \in I_{orig}: \Sigma_{new} \supseteq \Sigma_{orig}$	Original declaration & signature can be extended. <i>Note that consistency rule i) of the algebra guarantees that there is also an implementation of the extension</i>
ii) $C_{new}: i_{new}$ $Ass_{new} \upharpoonright C_{orig} = Ass_{orig}$	The associations of the new class do not limit the associations of the existing classes. <i>Note that consistency rule ii) of the algebra guarantees that the role of the association exists</i>

Note that consistency rule iii) of the algebra guarantees that the protocol of the extension only contains method calls that exist.

Continuing with the example:

There are a number of possibilities to integrate the grading mechanism in the Phase0 design of the example. Our choice in realizing the integration is by *extending* the Phase0 design with the grading mechanism and *extracting* the classes and methods from the Phase0 design that have become obsolete.

GradingMechanism is designed as an autonomous component that calculates the grade for a student's exam. GradingMechanism provides its own classes that contain grade information. This also includes a set of grading properties (such as a correction factor and a grading policy) that can be set by –for example– an exam. The algebra can now be used to calculate the behavioral and structural changes for different solutions for extending the Phase0 design.

In this example, we discuss the extension of ExamAnswers with GradingMechanism. The resulting design of the Phase1 TEA is depicted in Figure 3.

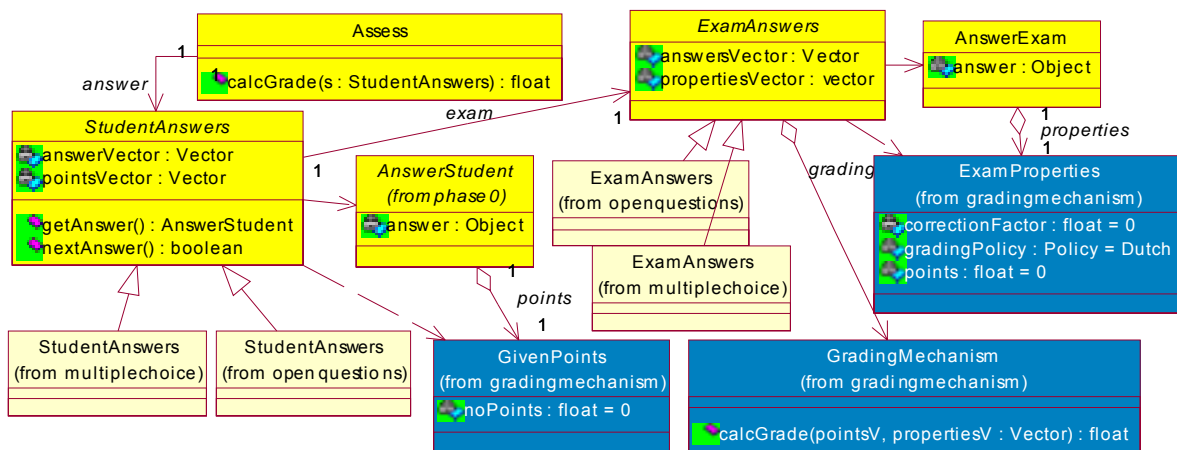


Figure 3. Phase 1: The Examination Assistant: extraction & extension

In this figure, the shaded classes are the classes that change, the “inverted” classes are the extension

We discuss the change in the classes `Assess` and `ExamAnswers` since we consider the change in these classes the most important. The classes `StudentAnswers`, `AnswerStudent` and `AnswerExam` are relatively simple extended by `ExamProperties` and `GivenPoints`. These extensions exist because they are inflicted by the `GradingMechanism` so it can operate correctly; we can say that the `GradingMechanism` dictates its use in these cases.

Class `ExamAnswers`:

This class is extended with `GradingMechanism`, so this class obviously changes. The consistency rule for extension guarantees that the behavioral part and the structural part of the extension are met. The `GradingMechanism` replaces the method `grading`, which can be extracted from the class.

Structural changes:

The extraction influences the signature of `ExamAnswers`. `grading()` is removed, but `getGrading()` is introduced. The attribute `propertiesVector` is introduced, this attribute can be retrieved. Formally the Phase1 Σ is:
 Σ : `ExamAnswersI` \rightarrow `getGrading(StudentAnswersI)`: float ; `getPropertiesVector()` : Vector

Behavioral Changes:

The class `C` is extended with a new association `Ass`. The formal notation is left out since it is relatively straightforward (see the notation of `C & Ass` in section 4.1 for an indication).

Class `Assess`:

Structural changes:

There are no structural changes since this class is not extended.

Behavioral Changes:

There are behavioral changes. These changes occur because of the use of `GradingMechanism`. The protocol of `Assess.calcGrade` uses the `ExamAnswers.grading` method. Since `ExamAnswers.grading` is replaced by the `GradingMechanism`, this method changes. The Phase1 declaration of `Assess` looks now as follows:

Δ : `AssessC` \rightarrow `calcGrade(StudentAnswersC)`: float,
(roles) `answers, exam, grading`
(protocol) `StudentAnswerI.getExam, StudentAnswerI.getPointsVector, ExamAnswerI.getGrading,`
 `ExamAnswerI.getPropertiesVector, GradingMechanism.calcGrade`

Calculation of the grade is now completely handled by the grading mechanism. `Assess.calcGrade()` is only used as an access method to start the calculation of the grade. `Assess.calcGrade()` is not removed from the model because `AssessI` is a system interface and the `calcgrade()` method can be used by other components to start the grading process.

5 CONCLUSIONS AND RELATED & FUTURE WORK

In this position paper, we introduced an algebra to express changes in a design. The algebra has behavioral and structural elements and can be used to reason about evolution. To guarantee the correct use of a model, we defined consistency rules for the algebra. To express the evolution of a model we formalized evolution operators. The formalization of an operator is based upon the consistency rules of the algebra.

We illustrated how the algebra can be used when extending an existing design. We showed that the extension (or the evolution) could inflict a number of design decisions when integrating the evolution.

With the algebra, we can formalize the evolution of a design. Usually there are more possibilities to integrate an evolution. The algebra can be used to find the most efficient integration in terms of behavioral and structural changes. We can also formalize decision heuristics to find the most efficient solution, however this is future work.

Related & Future work:

There is related work in the field of behavioral preservation (Refactoring [Opdyke 92]) and structure preservation (Design Patterns [Gamma et al. 95]). In future work we want to combine the results of these fields in our evolution algebra.

Integrating the evolution in the original design, and the heuristics to do it, need a lot of work, but we feel that the formalizations and expressing heuristics using the algebra look promising. We feel that the evolution algebra can be used to develop formalizations and/or heuristics that help in recognizing crosscuts.

Since a number of aspect and composition languages and technologies exist, future work is to provide the formal foundation to compare and choose the technology to implement the evolution operators.

6 REFERENCES

[Gamma et al. 95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, ISBN: 0-201-63361-2, 1995

[Gill 76] Arthur Gill, *Applied Algebra for the Computer Sciences*, Prentice-Hall, Series in Automatic Computation, 1976

[Opdyke 92] William Opdyke, *Refactoring Object-Oriented Frameworks*, Ph.D thesis, University of Illinois at Urbana-Champaign, 1992

APPENDIX A. ALGEBRA

Bas	the set of basic types
Iface	the universe of all interface names
Class	the universe of all class names
Meth	the universe of all method names
Role	the universe of all role names
$C \subseteq \text{Class}$	
$I \subseteq \text{Iface}$	
$A \subseteq \text{Ass}$	
$R \subseteq \text{Role}$	
$\text{Type} = \text{Iface} \cup \text{Bas}$	
$\text{Sig} = \text{Iface} \rightarrow (\text{Meth} \rightarrow \text{Type}_p^* \times \text{Type}_r)$	
$\text{Decl} = \text{Class} \rightarrow (\text{Meth} \rightarrow \text{Type}_p^* \times \text{Type}_r \times 2^{\text{Role}} \times \text{Prot})$	The declaration definition defines the protocol in terms of the set of activities that a model, class, method can issue upon another model, class, method.
$\text{Prot} = 2^{(\text{Iface} \times \text{Meth})^*}$	A protocol is a set of sequences of (interface , method)-pairs. A protocol describes an activity.
$P \subseteq \text{Prot}$	
$\Sigma \in \text{Sig}$	
$\Delta \in \text{Decl}$	
$\text{Ass} \subseteq C \times \text{Role} \times I$	Associations and aggregations
$\text{Inh} = C_{\text{super}} \times C$	Inheritance: overloading is not allowed. This is specified in Decl and the consistency rules for M
Sys, Env, Int: disjoint subsets of Iface	System (what is provided), Environment (what is needed), Internal (what is encapsulated)
$M = \langle \text{Sys}, \text{Env}, \text{Int}, C, \Sigma, \text{Ass}, \text{Inh}, \Delta \rangle$	Model M
$\text{Dom}(\Sigma) = \text{Sys} \cup \text{Env} \cup \text{Int}$	
$\text{Dom}(\Delta) = C$	

APPENDIX B. CONSISTENCY RULES

- i) For all system and internal interfaces there is at least 1 implementation
 $c \in C$ is an implementation of $l \in \text{Iface}$ if
 $\forall (m, T_p^*, T_r) \in \Sigma(l) \exists (m, T_p^*, T_r, A, P) \in \Delta(c)$
 $\stackrel{\text{def}}{=} \text{notation: } c:l$
for all $i \in \text{Sys} \cup \text{Int}$
there is a $c \in C$
such that $c:l$
And also
for all $c \in C$
 $\exists i \in I$
such that $c:l$
- ii) for all declared methods, the relevant interface & roles exist
 $\forall c \in C:$
 $\forall (m, T_p^*, T_r, A, P) \in \Delta(c):$
 $T_p^* \in (\text{Bas} \cup \text{Sys} \cup \text{Env} \cup \text{Int})^*$
 $T_r \in (\text{Bas} \cup \text{Sys} \cup \text{Env} \cup \text{Int})$
 $\forall r \in A : \exists (c, r, c') \in A$
- iii) for all declared methods, the protocol only uses declared methods
 $\forall c \in C:$
 $\forall (m, T_p^*, T_r, A, P) \in \Delta(c):$
 $\forall (c', m') \in P$
 $c' \in C$
 $m' \in \text{dom}(\Delta(c'))$
- iv) for all declared methods, the declarations in the super, super-super class, etc. of the class c is in the domain of class c
 $\forall (C_{\text{super}}, C_{\text{sub}}) \in \text{Inh}^+$
 $\forall m \in \text{dom}(\Delta(C_{\text{sub}})):$
 $\exists (m, T_p^*, T_r, A, P) \in \Delta(C_{\text{super}})$
 $\Delta(C_{\text{sub}})(m) = (T_p^*, T_r, A_{\text{sub}}, P_{\text{sub}})$

The associations and protocol of the subclass are not defined. This is future work