Model evolution and system evolution

Claudia Pons

Lifia-UNLP
Calle 50 esq.115 1er.Piso
(1900) La Plata, Argentina

e-mail cpons@info.unlp.edu.ar

Ralf-D. Kutsche

TU Berlin, FB Informatik
Einsteinufer 17
D-10587 Berlin, Germany

email: rkutsche@cs.tu-berlin.de

Abstract

In this paper we define an evolution mechanism with formal semantics using the metamodeling methodology [Geisler et al.98] based on dynamic logic. A remarkable feature of the metamodeling methodology is the ability to define the relation of intentional and extensional entities within one level, allowing not only for the description of structural relations among the modeling entities, but also for a formal definition of structural constraints and dynamic semantics of the modeled entities. While dynamic semantics on the extensional level means run-time behavior, dynamic semantics on intentional level describes model evolution in the system life cycle.

1. Introduction

As global observation in software engineering today, we can state that software development today never starts from scratch, but usually moves into the continuous extension and modification of software infrastructures of increasing size. As a consequence, the classical view of the software engineering discipline has to be revised, recognizing software development as an continuous process over long times. This paradigmatic issue of software and systems development can be subsumed under the term 'continuous software engineering' (CSE, cf. [Mueller, Weber, 1998]). A CSE approach aims to define a methodology for a smooth and consistent process of development facing continuous change (see fig. 1). It is based on classical techniques from software reengineering, including many valuable experiences from:

- classical forward engineering of new systems components

- reverse engineering of existing legacy components, as far as relevant issues for a further development are missing, and

- the re-engineering basics starting from simple maintenance aspects via different levels of change and modification tasks until larger projects of renovation or even replacement of complete components.

CSE, in detail, means a very careful and systematic analysis of:

- primary and further effects of a fwd-/rev-/re-engineering step, i.e. analysis of all induced modifications by an initial modification with each fwd-/rev-/re-engineering step

- invariants for each fwd-/rev-/re-engineering step

- and consistent propagation of all kinds of models with respect to the required induced modifications

as a necessary prerequisite for discussion among CS engineers, customers, managers, and - very important - for a successful tool development, all of them being based on a general understanding of model-based software development as the only way towards a continuous software development.

This general view of software engineering (see [Kutsche and Sünbül 98] and [Kutsche and Sünbül 99] for a more detailed introduction to this topic) gives rise to the subsequent proposal of model-based evolution techniques.

A widely used and generally accepted technique in modern software engineering is the combination of different models (or views) for the description of software systems. The primary benefit of this approach is to model only related aspects (like structure or behavior). For this principle, called

Separation of Concern, different specialized techniques mostly of diagrammatic nature have been developed. Using different models clarifies different important aspects of the system, but it has to be taken into consideration that these models are dependent on each other and they are semantically overlapping. Therefore, it is necessary to state how these models are related. The different views on a system have to be semantically compatible and there are several constraints between them.

Compatibility between views must be also guaranteed during evolution of the system. The dynamic nature of contemporary business requirements forces developers to make their application more flexible and adaptable. Business rules change dynamically, so it is necessary to provide a flexible representation of them. In the last few years the concept of *Dynamic Object Model* has emerged. A system with dynamic object models has explicit object models that it interprets at run-time. If you change a model, then the system changes its behavior, allowing a company to evolve the way it does its business.

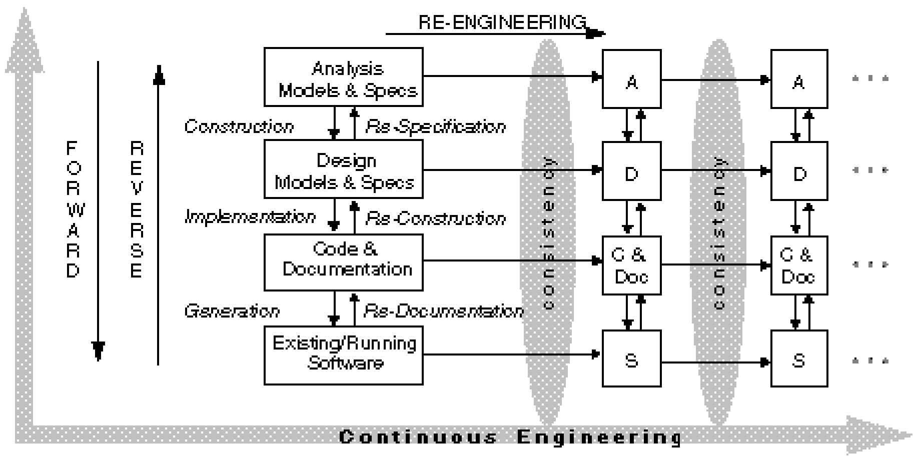

Figure 1: Continuous Engineering

There are many ways in which a model can be adapted. In incremental modification, the modified model is derived from the original model by means of an incremental modification mechanism that specifies which modifications (or adaptations) have been made to the original model. When a model is adapted, unexpected behavior may occur in other models that depend on it. The behavior of the adapted model might have changed, properties of the model that were valid before might not hold anymore, etc. This kind of conflicts is referred as evolution conflicts and has been deeply analyzed by different authors (see for example [Steyaert et al. 96, Lucas 97, Mens et al. 98]).

As a consequence it is necessary to account with evolution mechanisms that guarantee consistency of the software system through evolution. To achieve this requirement evolution mechanisms must provide the following issues:

- identification of primitive modifications.
- identification of evolution patterns (e.g. meaningful groups of primitive modifications that occur together frequently).
- formal definition of evolution operations, including applicability conditions and change propagation.
- identification of conflict situations originated by the application of evolution operations.

In this paper we will use the Metamodeling technique [Geisler et al.98] as an evolution mechanism with formal semantics. In section 2 we describe the metamodeling technique. In section 3 we introduce the dynamic metamodel, in section 4 we describe primitive evolution operations. In section 5 we discuss evolution conflicts and define rules to detect evolution conflicts automatically. Section 6 contains a summary of related works. Finally, in section 7 we present our conclusions.

2. The Metamodeling Technique

Metamodeling is a very promising technique for the definition of multiple view languages like the Unified Modeling Language [UML97]. Using a metamodel, it is possible to determine how these models constitute the whole system. In [Geisler et al.98] we have introduced a metamodeling methodology based on a formal metalanguage. This methodology is based on the four-level approach, see Section 2.1. It allows for the description of all relevant aspects of the entities of the metamodel. The presented approach allows not only for the description of structural relations among the modeling entities, but also for a formal definition of structural constraints and dynamic behavior of the modeled entities.

2.1 The four-level approach

Visual modeling languages are graphic languages for specifying, visualizing, constructing and documenting the artifacts of software systems prior their construction or renovation. Generally, the conceptual framework for modeling notations is based on an architecture with four levels. The different levels of abstraction are illustrated in Figure 2 [Odell95]:

- ***metamodel level.*** A *metamodel* is a model for the information that can be expressed during software modeling. Basically, a metamodel is a model of models. It consists of entities defining the modeling language such as *Class_diagrams, State_machines, Sequence_diagrams*, etc.
- ***model level***. On the other hand, a *model* is an instance of a metamodel. It describes the objects inherent to the application domain. We have different models describing the underlying physical system, e.g. Employer is an instance of Class of the metamodel.
- ***data&process level***. On the data&process level, the entities are run-time objects, i.e. instances of classes running on a concrete system.
- ***meta-metamodel level.*** In order to express these concepts, we need a further level, defining the used language for the metamodel. This level is called meta-metamodel level.

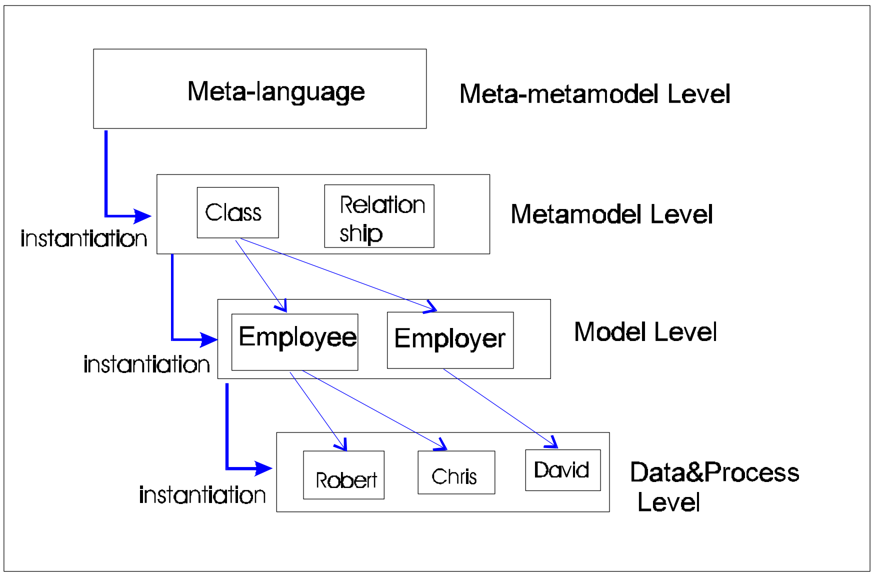

Figure 2: four-level architecture

2.2 Dichotomy of metaentities: intentional vs. extensional entities

An important contribution of the metamodeling technique[Geisler et al.98] is the distinction between:

- intentional metaentities
- extensional metaentities

Intentional entities have a counterpart in the concrete syntax of the modeling language, such as Class, Association or StateMachine. In contrast, extensional entities, such as Object or Link, are used to store necessary run-time information.

There is different kind of relationships between these entities (see Figure 3):

- *Intra-model relationships:* Within the metamodel the modeling entities itself but also the relationships between these entities are described, e.g. an Association consists of AssociationEnds. Furthermore, entities of different models might be also related. Consider for example the association between Class and Behavior which establishes the relation between structural (Class) and behavioral elements (StateMachine).
- *Intra-system relationships*: for example the relationship named 'slot' between Object and AttributeLink, denotes the connection between an Object and the values of its attributes.
- *Inter-level relationships:* The integration of model level (intentional entities) and system level (extensional entities) within the metamodel is important in order to state the relationship between entities belonging to these two different levels. There is a special relationship among some modeled entities with its corresponding modeling entity, This relationship denotes 'instantiation', for example an Object is an instance of a Class, whereas Links are instances of Associations.

A remarkable feature of this approach is the ability to define the relation of intentional and extensional entities within one level.

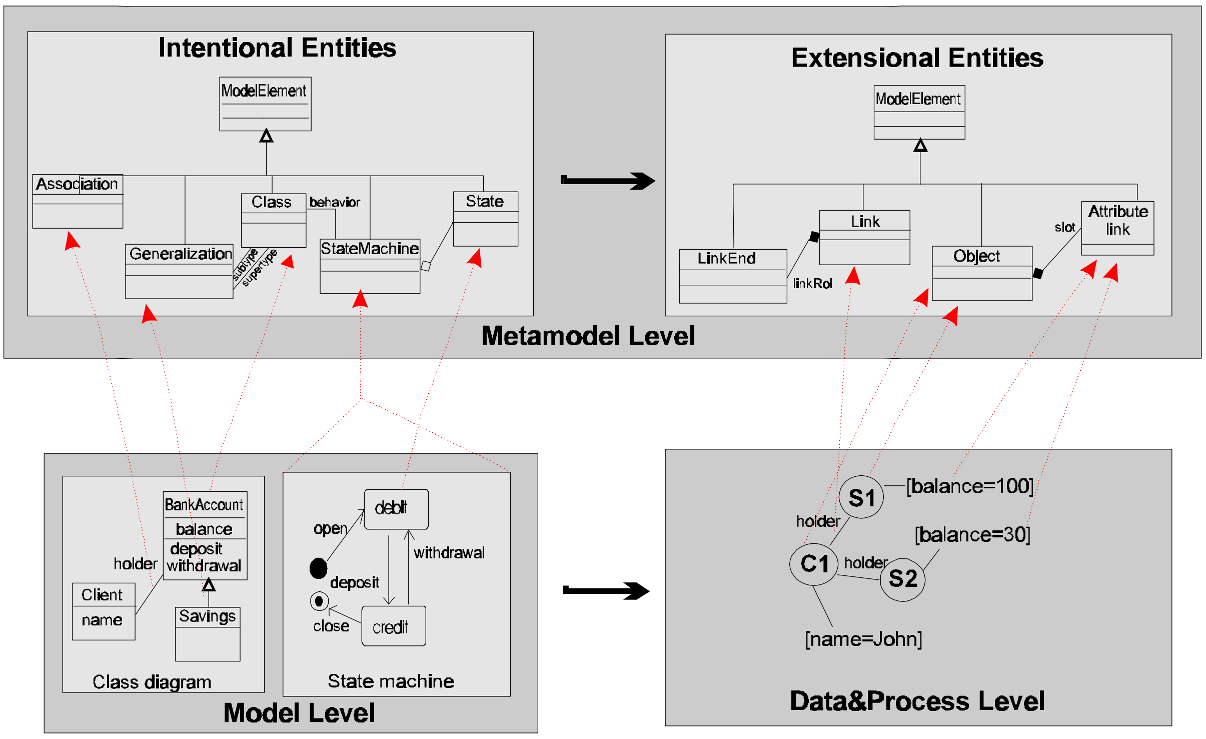

Figure 3: Dichotomy of metaentities

3. The Dynamic Metamodel

In this section we introduce a dynamic metamodel for the UML [UML 97]. The basic idea behind this formalization is using the metamodel for the integration of both levels of the architecture of modeling notations (intentional and extensional level). The integration of both levels allows us to express both static and dynamic aspects of either the model or the modeled system into a first order formal framework.

3.1 Static and Dynamic aspects of metaentities

The metamodel consists of metaclasses. Within a metaclass, we are able to express all aspects relevant to a metaentity:

- **Abstract Syntax**: An abstract description of the entities that form a model of the respective language.
- **Static Semantics**: Well-formedness conditions between the syntactic entities.
- **Dynamic Semantics**: The (operational) behavior of the entities of the specification, such as I/O, reaction to stimuli, effect of executing an operation, etc.

In Figure 4 the different roles of abstract syntax, static and dynamic semantics for intentional and extensional entities is summarized. In Figure 5 examples of this roles are shown. The most remarkable difference is observable for the dynamic semantics. While dynamic semantics on the extensional level means run-time behavior, dynamic semantics on intentional level describes model evolution in the development process.

	Intentional	**Extensional**
Abstract Syntax	specification in the modeling language(s)	state of a program at run-time
Static Semantics	well-formed specification	possible/consistent system states of a program
Dynamic Semantics	evolution of the specification during the system life cycle.	dynamic behavior of the program at run-time

Figure 4: Intentional vs. Extensional Entities

	Intentional	**Extensional**
Abstract Syntax	Class, Association	Object, Message
Static Aspects	No attributes may have the same name within a Classifier.	The values of attributes have to match the declarations in the Classifier.
Dynamic Aspects	meaning of adding an Attribute in a Class.	How an object reacts to a message.

Figure 5*: Examples*

3.2 Formalization of evolution actions

Evolution actions are formalized as actions in Dynamic Step Logic [Wieringa and Broersen 98]. That is to say, each evolution action is a term of sort Action. This allow us to predicate on actions, for example: **$\forall\alpha$:ActionP(α)**. Other important feature of Dynamic Step Logic is that it is an order-sorted logic. As a consequence of this fact it is possible to define a hierarchy of actions by defining sub-sorts of the sort Action. The sub classification of actions allows us to specify each kind of evolution action in a hierarchical way, for example: The statement ModelEvolution$\leq$Action implies that if $\forall\alpha$:ActionP(α) then $\forall\alpha$: ModelEvolution P(α). But it is possible to define properties (e.g. $\forall\alpha$: ModelEvolution Q(α)) that hold for ModelEvolution actions in particular but do not hold for Actions in general.

3.2.1 Granularity of evolution actions

Evolution actions are classified in two categories (see Figure 6):

- PrimitiveActions
- CompositeActions

Primitive actions represent atomic modifications whereas composite actions represent groups of modifications that are applied together. A composite action represents an uninterruptable transaction. Consistency of the system must be guaranteed after the execution of actions, but it may not be guaranteed during the execution of a composite action.

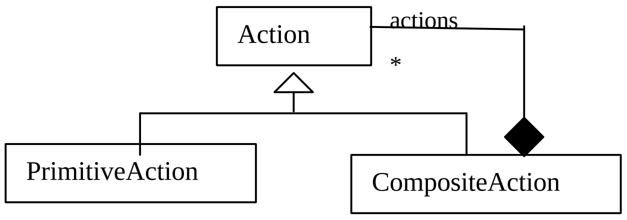

Figure 6: Action Hierarchy

3.2.2 Classification of primitive evolution actions

Entities in the metamodel are classified in:

♦ ***intentional entities*** and ♦ ***extensional entities***.

As a consequence of this fact, we propose a hierarchy of actions considering two groups of primitive evolution actions:

- **model evolution** (evolution of intentional entities)
- **system evolution** (evolution of extensional entities)

On the other hand, from the point of view of the kind of modification, action may be classified in tree groups:

- **Creation Actions** insert a new element into either the model or the system.
- **Deletion Actions** delete an existing element from either the model or the system.
- **Modification Actions** modify an existing element from either the model or the system.

Figure 7 contains the hierarchy of primitive actions. Notice that six subclasses of PrimitiveAction can be obtained from the combination of both hierarchies: ModelEvolution-Creation, ModelEvolution-Deletion, ModelEvolution-Modification, SystemEvolution-Creation, SystemEvolution-Deletion, SystemEvolution-Modification.

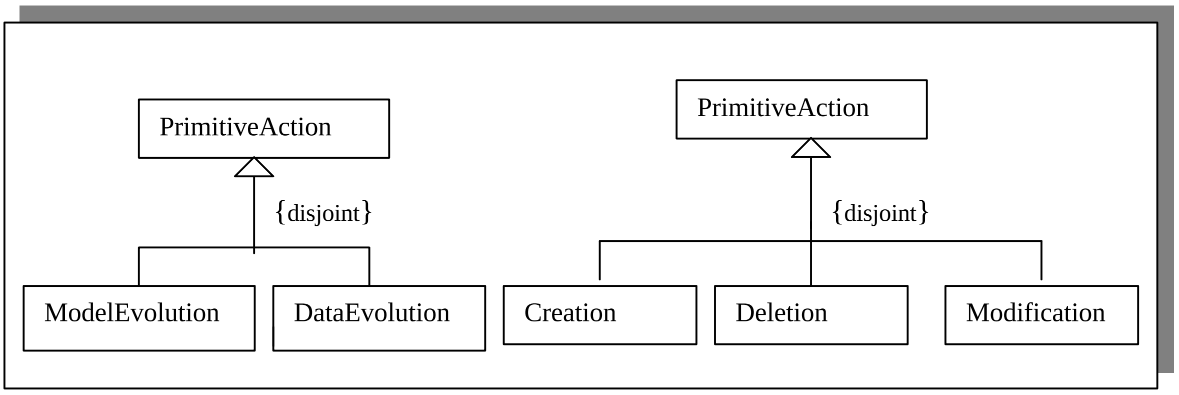

Figure 7: hierarchy of primitive evolution actions

It is important to highlight the differences between a primitive action applied on an intentional entity from a primitive action applied on an extensional entity. For example:

- while a creation on the extensional level (SystemEvolution-Creation) means the creation of a new data object (e.g. Peter) as an instance of some Intentional entity (e.g. Employee), a creation on the intentional level (ModelEvolution-Creation) means the definition of a new view of the system (e.g. a new Class) as an instance of an entity in the metamodel (e.g. the metaclass Class).

- Updating (i.e. modifications) of an intentional object means to manipulate the model, while updating the state of an extensional object contributes to the dynamic behavior of the particular modeled system.

Since the rank of Primitive Actions is very wide, It is useful to refine their classification. From the point of view of the kind of model element that is being modified, Model evolution action can be sub-classified in three groups:

- **Structural Actions** represent evolution of structural model elements, such as Class, or relationship(e.g. association or generalization).
- **Behavioral Actions** represent evolution of behavioral model elements, such as state Machine. Behavioral actions modify the specification of a group of operations belonging to a Classifier.
- **Constraint Actions** describe evolution of the constraints in the model.

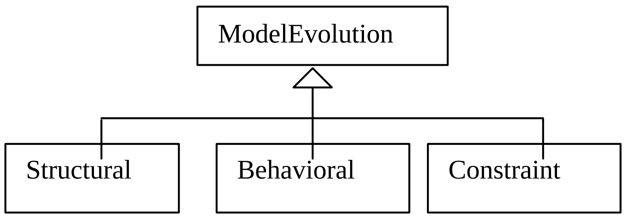

Figure 8: Hierarchy of Model Evolution Action

Classification of model evolution actions is shown in figure 8. No further subclassification of SystemEvolution actions is introduced. As a consequence, twelve concrete subclasses of actions can be obtained from the current subclassification of intentional and extensional actions. Table below contains these classification.

Structural-Creation	Structural-Deletion	Structural-Modification
Behavioral- Creation	Behavioral-Deletion	Behavioral-Modification
Constraint- Creation	Constraint-Deletion	Constraint-Modification
System-Creation	System-Deletion	System-Modification

In Appendix 1 we describe all the actions included in each group.

3.2.3 Overview

In section 3.3 we introduce the formal description of intentional level. In section 3.4 we present the formal description of extensional level. Section 3.5 contains considerations about the integration of both levels. Finally, in section 3.6 we describe the semantics function that maps the UML constructions to elements in the metamodel.

3.3 Intentional level

In the UML, class diagrams model the structural aspects of the system. Classes and relationships between them, such as generalizations, aggregations and associations constitute class diagrams. On the other hand, the dynamic part of the system is modeled by collaboration diagrams that describe the behavior of a group of instances in terms of message sending, and by state machines that show the intra-object dynamics in terms of state transitions.

Models evolve over their life cycle of for a variety of reasons. One of the most common forms of evolution involves structural changes such as the extension of an existing specification by addition of new classes of objects or the addition of attributes to the original classes of objects. At the other extreme, evolution at this level might reflect not only structural changes but also behavioral changes of the specified objects. Behavioral changes are reflected for example in the modification of collaboration diagrams or state machines.

The formal specification of the intentional level consists of a dynamic logic signature Σ_{UML}**= ((S**$_{UML}$**, ≤), F**$_{UML}$**, P**$_{UML}$**)** and a formula Φ_{UML} over Σ_{UML}, where the sorts in the signature correspond to model elements (such as classes, relationships, state machines) and the action symbols represent modifications on the specification of the system, for example adding a new class, modifying an existing class, etc.

3.4 Extensional level

The elements in the extensional level are basically instances (data values and objects) and messages. On the extensional level a system is viewed as a set of objects collaborating concurrently. Objects communicate each other through messages that are stored in semi-public places called mailboxes. Each object has a mailbox where other objects can leave messages. There exist privacy requirements to make sure that for all object o, only o receives messages destined to o.

The formal specification of the elements in the Extensional level consists of a dynamic logic signature Σ_{SYS}**=** ((**S**$_{SYS}$,≤) **F**$_{SYS}$**, P**$_{SYS}$**)** and a formula Φ_{SYS} over Σ_{SYS}, where the sorts in the signature correspond to system elements (such as objects, values and messages) and the action symbols represent system evolution.

3.5 Integration of both levels: the M&S-theory

The M&S-theory (Model&System theory) is a first-order dynamic logic theory, expressing the integration of the model level with the system level. It consists in a signature (defining the language of the theory) and a set of axioms:

$$\text{M\&S-theory}=(\Sigma_{M\&S}, \phi_{M\&S})$$

The signature of the theory, $\Sigma_{M\&S}$=((**S**, ≤), **F, P**), is a first-order dynamic logic signature that includes both the signature Σ_{UML} and the signature Σ_{SYS} .

On the other hand, the theory includes three different kind of axioms. That is to say, $\phi_{M\&S} = \phi_{UML} \wedge \phi_{SYS} \wedge \phi_{JOINT}$. Firstly, Φ_{UML} is the formula defining the intentional entities. Secondly, Φ_{SYS} is a formula describing the extensional entities (semantics of objects and messages). Thirdly, Φ_{JOINT} is constructed over the extended *M&S* language and thus it can express at the same time system properties (e.g. behavioral properties of objects), model properties (e.g. properties about the specification of the system) and properties relating both aspects.

In appendix 2 we present a part of the M&S-theory, the complete definition is in [Pons 99] (for an overview of the theory and its applications see [Pons et al.99]).

3.6 Semantics interpretation of the M&S-theory

The semantics domain where the M&S-theory is interpreted is a set of transition systems (for details about semantics of dynamic logic specifications, see [Wieringa and Broersen 98]). A transition system is a set of states with a set of transition relations on states. The domain for states is an algebra whose elements are both intentional and extensional elements. The set of transition relations is partitioned into two disjoint sets:

- a set of transitions representing modifications on the specification of the system (i.e. evolution at the intentional level), and
- a set of transitions representing modifications on the system state (i.e. evolution at the extensional level).

Figure 9 shows an example of evolution in both directions. Note that as a consequence of an evolution in the specification (i.e. the modification of transition t2 adding a new effect: to send the message *notify* to the holder) the behavior of the object *o* has changed.

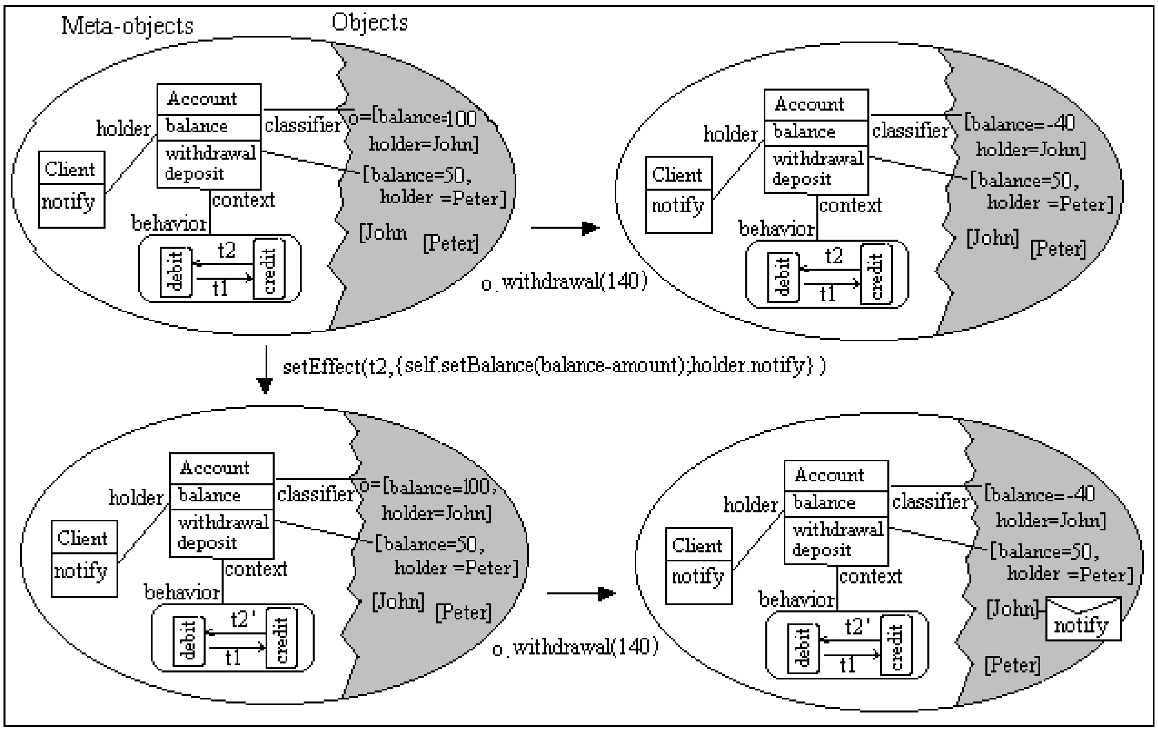

Figure 9: two-level evolution

4. Specification of Primitive Evolution Actions

In the M&S-theory, each evolution action is defined by means of two formulas:

- **Necessary preconditions** to describe the applicability conditions of operations. The formula ÿ(⟨*op*⟩*true* →*cond*) states that the operation *op* is applicable only if the condition *cond* is true.
- **Sufficient postconditions** to describe the effect (direct effect and change propagation) of the operations. The formula ÿ([*op*]*cond*) states that after the application of the operation *op* the condition *cond* is true.

These formulas may contain either intentional entities, or extensional entities or both. This feature allows us to define:

- **Intra-level change propagation**:
  How a modification over an extensional entity impacts on other extensional entities.
  How a modification over an intentional entity impacts on other intentional entities.
- **inter-level change propagation**:
  How a modification over an extensional entity impacts on intentional entities.
  How a modification over an intentional entity impacts on extensional entities.

The specification of each evolution action consists in four sections:

1- ***Action*** act
2– ***Precondition*** τ
3– ***Effect*** γ
4- ***Propagation*** δ

The schema above represents the following dynamic formula: (⟨act⟩*true* →τ) $\wedge$ ([act] ($\gamma \wedge \delta$))

Preconditions are applicability conditions, that is to say conditions under which an evolution action is semantically correct. The clause effect specifies the direct impact of the action, whereas the clause propagation specifies the side effects of the action on other related entities.

As an example, we introduce the specification of the following primitive actions: addClassifier, addAssociation, addGeneralization, deleteFeature, setBehavior and destroy.

Examples:

Action addClassifier(p,c)
Precondition
[1]The source Package exists and the new Classifier does not exist (as a consequence, $c \notin$ allContents(p))
Exists(p) $\wedge \neg$Exists(c)
[2] in a Package the Classifier names are unique
$\forall c_1$: Classifier ($c_1 \in$ contents(p) $\rightarrow$ name(c_1) $\neq$ name(c))
[3] the new Classifier does not participate in any relationship.
AssociationEnds(c)=$\emptyset$ $\wedge$ generalizations(c)=$\emptyset$ $\wedge$ specializations(c)=$\emptyset$
[4] The type of the attributes must be included in the Package.
$\forall f \in$ allAttributes(c) type(f)$\in$allContents(p)
[5] The type of the Parameters must be included in the Package .
$\forall f \in$ allOperations(c) $\forall m \in$ parameters(f) type(m)$\in$allContents(p)
Effect
Exists(c) $\wedge c \in$ ownedElement (p) $\wedge$ package(c)=p
Propagation
[1]Life dependency
$\forall f \in$ features(c)Exists(f)

Action addAssociation(p,a)
Precondition
[1]The source Package exists and the new Relationship does not exist (as a consequence, $a \notin$ allContents(p) y $\forall c \in$ allConnectedElements(a) $a \notin$ allAssociations(c))
Exists(p) $\wedge \neg$Exists(a)
[2] all elements connected by the new relationship must be included in the Package.
$\forall c \in$ allConnectedElements(a) $c \in$ allContents(p)
[3] in a Package the association names are unique
$\forall a_1$: Association ($a_1 \in$ contents(p) $\rightarrow$name(a_1) $\neq$ name(a))
[4]No opposite AssociationEnds may have the same rol-name within a Classifier
$\forall c \in$ allConnectedElements(a) ($\forall e1 \in$ allOppositeAssociationEnds(c) $\forall e2 \in$ connections(a) name(e1) $\neq$name(e2))
Effect
Exists(a) $\wedge a \in$ ownedElement(p) $\wedge$ package(a)=p
Propagation
[1]Life-dependency.
$\forall e \in$ connections(a) Exists(e)
[2]The association is connected to the classifiers
$\forall e \in$ connections(a) $e \in$ associationEnds(type(e))

Action addGeneralization(p,g)
Precondition
[1] The source Package exists and the new Relationship does not exist (as a consequence, $g \notin$ allContents(p))
Exists(p) $\wedge \neg$Exists(g)
[2] all elements connected by the new relationship must be included in the Package.
supertype(g)$\in$ allContents(p) $\wedge$ subtype(g)$\in$ allContents(p)
[3] A root cannot have any Generalizations.
$\neg$isRoot(subtype(g))
[4] No GeneralizableElement which is a leaf can have a subtype
$\neg$isLeaf(supertype(g))
[5] Circular inheritance.
IsA(supertype(g), subtype(g)) $\rightarrow$ supertype(g) = subtype(g)
[6] multiple inheritance.
$\forall$c:Classifier (*IsA*(subtype(g),c) $\rightarrow$
    $\forall$f,g:Feature(($f \in$ allFeatures(supertype(g)) $\wedge$ $g \in$ allFeatures(c) $\wedge$ name(f)=name(g)) $\rightarrow$ f=g))
[7] Constraint consistency
consistent(allConstraints(subtype(g)) $\cup$allConstraints(supertype(g))) $\wedge$

$\forall c \in$ subtypes(subtype(g)) consistent(allConstraints(c) $\cup$allConstraints(supertype(g)))
[8] Behavioral consistency
refinement(behavior(subtype(g)) , behavior(supertype(g)))
Effect
Exists(g) $\wedge$g$\in$ ownedElement(p) $\wedge$ package(g)=p
Propagation
[1] The new generalization is linked to the generalizable elements
g$\in$ specialization(supertype(g)) $\wedge$g$\in$ generalizations(subtype(g))

Action deleteFeature(c,f1)
Precondition
[1] The Feature must exist in the Classifier.
f1$\in$ attributes(c)
[2] The deleted Feature cannot be referenced from other elements in the package.
$\forall$m$\in$ allContents(p) f1$\notin$ referencedElements(m)
Effect
$\neg$Exists(f)$\wedge$f$\notin$ features(c)
Propagation
[1] The corresponding slot must be deleted from all the existing instances of c.
$\forall$i:Instance ((Exists(i) $\wedge$classifier(i)=c) $\rightarrow$($\exists$l:AttributeLink (atribute(l)=f1 $\wedge$ slots(i)=slots(i)-{l})))

Action setBehavior (c,h)
Precondition
[1] The source Classifier exists and the new StateMachine does not exist.
Exists(c) $\wedge$ $\neg$Exists(h)
[2] They are syntactically compatible, that is to say only features of the Classifier are referenced in the state Machine
syntactic-compatible(c,h)
[3] Behavioral correctness: the state machine satisfies the pre and post conditions of the corresponding operations. [behavior(c):=h] means term substitution in the context of the formula.
($\forall$o$\in$ instances(c) $\forall$<op,s,o,p>:Message
eval(precondition(op)[self:=o, parameters:=p])=true
$\rightarrow$[r.<op,s,o,p>]eval(postcondition(op)[self:=o, parameters:=p])=true) [behavior(c):=h]
[4] Behavioral correctness: all the operations specified by the StateMachine should preserve the constraints of c.
($\forall$i$\in$ allConstraints(c) $\forall$o$\in$ instances(c) $\forall$m:Message
eval(i[self:=o])=true $\rightarrow$[o.m]eval(i[self:=o])=true) [behavior(c):=h]
Effect
Exists(h) $\wedge$ package(h)=package(c) $\wedge$ h$\in$ ownedElements(package(c)) $\wedge$ context(h)=c $\wedge$ behavior(c)=h
Propagation
There is no structural propagation on instances of c, but their behavior is affected indirectly.

Action destroy(o)
Precondition
[1] The object exists
Exists(o)
Effect
[1]$\neg$Exists(o)
[2] The associations of the deleted object are modified
$\forall$l$\in$ linkEnds(o) instance(l)=nullElement
Propagation
[1] all the parts of a composite objects are destroyed
$\forall$i$\in$ allParts(o) $\neg$Exists(i)
[2] the association of the deleted objects are modified.
$\forall$i$\in$ allParts(o) ($\forall$l$\in$ linkEnds(i) instance(l)=nullElement)

5. Evolution Conflicts

Every evolution mechanism must guarantee consistency of the software system through evolution. To achieve this requirement evolution mechanisms must be based on formal understanding of change propagation to guarantee the systematic detection of inconsistencies (i.e. conflicts). We show how the proposed formal evolution mechanism allows us to identify conflict situations originated by the application of evolution operations. Using the M&S-theory we can define a set of rules to identify conflict situations originated by the application of evolution actions. We have to consider two arbitrary modifications that do not cause problems when they are applied exclusively, but that can arise conflicts when they are integrated (i.e. they are applied together). According to the hierarchies of model evolution actions and system evolution actions there are 144 (i.e. 12^2) different combinations of actions. Matrix in figure 11 shows this combination of actions. The matrix is symmetric (i.e. T[i,j]=T[j,i]). The presence or absence of conflict is independent of the order in which evolution actions are applied, although the specific kind of conflict could be different depending on the order of actions.

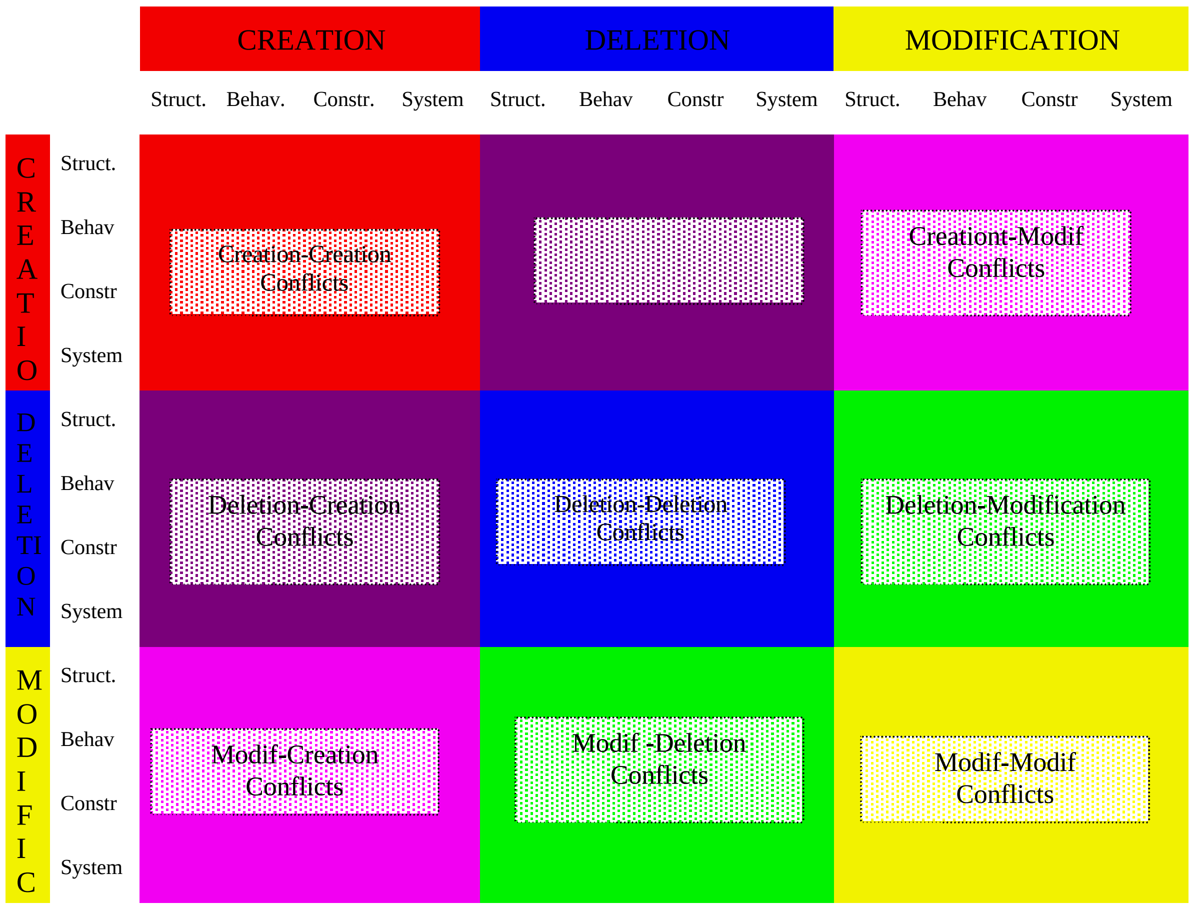

Figure 11: Conflict Matrix

<u>Example:</u>

As an example we show part of the analysis of the Creation vs. Creation Conflict sub-matrix.This matrix shows the conflicts that arise from the combination of two Creation actions that do not cause problems when they are applied exclusively. The matrix is symmetric (i.e. T[i,j]=T[j,i]). The kind of conflict is the same, independently of the order in which Creation actions are applied.

creation \ creation	Structural	Behavioral	Constraint	System
Structural	1	2	3	4
Behavioral	2	5	6	7
Constraint	3	6	8	9
System	4	7	9	10

In the following sections we analyze some of the conflicts separately:

1 Structural Creation vs. Structural Creation

	addSubpackage(p1,q1)	addClassifier(p1,c1)	addAssociation(p1,r1)	addGeneralization(p1,r1)	addFeature(c1,f1)	addConnection(a1,e1)
addSubpackage(p,q)	conflict 1	no conflict	no conflict	no conflict	no conflict	no conflict
addClassifier(p,c)		conflict 2	no conflict	no conflict	no conflict	no conflict
addAssociation(p,r)			conflict 3	no conflict	no conflict	no conflict
addGeneralization(p,r)				conflict 4	conflict 5	conflict 6
addFeature(c,f)					conflict 7	conflict 8
addConnection(a,e)						conflict 9

There is not conflict if the first action creates either a new Classifier, Package, Association, Feature or AssociationEnd because the new created element cannot be referenced from the second action. There is possibility of conflicts only if both actions create two ModelElements of the same kind and with the same name:

[Conflict 1] in a Package the Package names are unique
Conflict(addSubpackage (p,q), addSubpackage (p1,q1))$\leftrightarrow$ p=p1 $\wedge$ name(q)=name(q1) $\wedge$ q$\neq$q1
[Conflict 2] in a Package the Classifier names are unique
Conflict(addClassifier(p,c), addClassifier(p1,c1))$\leftrightarrow$ p=p1 $\wedge$ name(c)=name(c1) $\wedge$ c$\neq$c1
[Conflict 3] in a Package the association names are unique
Conflict(addAssociation(p,r), addAssociation(p1,r1))$\leftrightarrow$ p=p1 $\wedge$ name(r)=name(r1) $\wedge$ r$\neq$r1
[Conflict 7] No Features may have the same name within a Classifier
Conflict(addFeature(c,f), addFeature(c1,f1))$\leftrightarrow$ c=c1 $\wedge$ name(f)=name(f1) $\wedge$ f$\neq$f1
[Conflict 8] The name of an Attribute cannot be the same as the name of an opposite AssociationEnd
Conflict(addFeature(c,f), addConnection(a,e))$\leftrightarrow$ a$\in$allConnections(c) $\wedge$ name(f)=name(e)
[Conflict 9] (a) The AssociationEnds must have a unique name within the Association. (b)At most one AssociationEnd may be an aggregation or composition. (c)No opposite AssociationEnds may have the same rol-name within a Classifier.
Conflict(addConnection(a,e), addConnection(a1,e1))$\leftrightarrow$
(a) (a=a1 $\wedge$ name(e)=name(e1) $\wedge$ e$\neq$e1)
(b) $\vee$ (a=a1 $\wedge$ aggregation(e)<>#none$\wedge$aggregation(e1)<>#none $\wedge$ e$\neq$e1)
(c) $\vee$ $\exists$c:Classifier (a$\in$allConnections(c)$\wedge$a1$\in$allConnections(c)$\wedge$a$\neq$a1$\wedge$name(e)=name(e1))

[Conflict 4]
Let r and r1 be the generalizations that are added into the model by the evolution actions. Let super1=supertype(r), super2=supertype(r1), sub1=subtype(r), sub2=subtype(r1)

Conflict(addGeneralization(p,r), addGeneralization(p1,r1)) $\leftrightarrow$
(a) IsA(super2,sub1) $\wedge$ IsA(super1,sub2) $\wedge$ $\neg$(super1=sub1=super2=sub2)
(b) $\vee$ (IsA(super1,sub2) $\wedge$ $\neg$consistent(allConstraints(sub1)$\cup$ allConstraints(super2)))
$\vee$ (IsA(super2,sub1) $\wedge$ $\neg$consistent(allConstraints(sub2)$\cup$allConstraints(super1)))
(c) $\vee$ (IsA(super1,sub2) $\wedge$ $\neg$refinement(behavior(sub1), behavior(super2)))
$\vee$ (IsA(super2,sub1) $\wedge$$\neg$refinement(behavior(sub2), behavior(super1)))
(d) $\vee$ (sub1=sub2 $\wedge$ super1$\neq$super2 $\wedge$
$\exists$f,g:Feature(f$\in$allFeatures(super1)$\wedge$ g$\in$allFeatures(super2) $\wedge$ f$\neq$g $\wedge$ name(f)=name(g)))

The formulas above define:
(a) Cyclic generalization. (b) Inconsistent constraint refinement. (c) Inconsistent behavioral refinement.
(d) Name conflict by multiple inheritance.

[Conflict 5] Action a adds feature f1 to Classifier C. After applying action b, C inherits feature f1. There is a feature redefinition. Since feature redefinition is allowed, this situation is not a conflict, but a warning. A similar situation occurs when the inserted feature is inherited by a Class that already had a feature with the same name.

$\neg$Conflict(addGeneralization(p,r), addFeature(c,f))

Warning(addGeneralization(p,r), addFeature(c,f)) $\leftrightarrow$

subtype(r)=c $\wedge$ ($\exists$f1$\in$ allFeatures(supertype(r)) name(f1)=name(f))

$\vee$ (supertype(r)=c $\wedge$ ($\exists$f1$\in$ allFeatures(subtype(r)) name(f1)=name(f)))

[Conflict 6] This conflict is similar to the one described above.

$\neg$Conflict(addGeneralization(p,r), addConnection(a,e))

Warning(addGeneralization(p,r), addConnection(a,e)) $\leftrightarrow$

a$\in$ allConnections(subtype(r)) $\wedge$ ($\exists$e1$\in$ allOppositeAssociationEnds(supertype(r)) name(e1)=name(e))

$\vee$ (a$\in$ allConnections(supertype(r))$\wedge$($\exists$e1$\in$ allOppositeAssociationEnds(subtype(r))name(e1)=name(e)))

2 Structural Creation vs. Behavioral Creation

	setBehavior(c1,h)
addSubpackage(p,q)	no conflict
addClassifier(p,c)	no conflict
addAssociation(p,r)	no conflict
addGeneralization(p,r)	conflict 1
addFeature(c,f)	no conflict
addConnection(a,e)	no conflict

In general, there is not conflict between Structural Creation and Behavioral creation actions, because the elements that are inserted by the structural action cannot be referenced from the behavioral action, since they are new elements (i.e. they do not exist in the model on which behavioral action is applied). For example, if the structural action adds a feature f to a Classifier c and the behavioral action attaches a State Machine h to c, the feature F is not used by State Machine h because f did not exist in c before applying the structural creation.

[Conflict1] The only source of conflict is the creation of a Generalization that arise inconsistency of the inherited behavior. There may be conflict if the behavioral action a attaches a state machine h to a Classifier c and after the creation of the new generalization r, c inherits other state machine h'. As a consequence of the combination of both actions there is a refinement of behavior and the attached state machines must satisfy the refinement relationship. A similar situation occurs when the inserted state machine is inherited by a Class that already had a state machine.

Conflict(addGeneralization(p,r), setBehavior(c,h)) $\leftrightarrow$

isA(c,subtype(r)) $\wedge$ $\neg$refinement(h , behavior(supertype(r)))

$\vee$ isA(supertype(r),c) $\wedge$ $\neg$refinement(behavior(subtype(r)), h)

4 Structural Creation vs. System Creation

	newObject(c1,o)	newLink(a1,k)
addSubpackage(p,q)	no conflict	no conflict
addClassifier(p,c)	no conflict	no conflict
addAssociation(p,r)	no conflict	no conflict
addGeneralization(p,r)	conflict 1	conflict 2
addFeature(c,f)	conflict 3	no conflict
addConnection(a,e)	no conflict	conflict 4

$\neg$Conflict(addSubpackage(p,q) newObject(c1,o))

$\neg$Conflict(addClassifier(p,c) , newObject(c1,o))

$\neg$Conflict(addAssociation(p,r) , newObject(c1,o))

$\neg$Conflict(addSubpackage(p,q) newLink(a1,k))

$\neg$Conflict(addClassifier(p,c) , newLink(a1,k))

$\neg$Conflict(addAssociation(p,r) , newLink(a1,k))

$\neg$Conflict(addConnection(a,e), newObject(c1,o))

$\neg$Conflict(addFeature(c,f) , newLink(a1,k))

[Conflict 1] As a consequence of the generalization the Classifier c1 may inherit new attributes that will not have the corresponding slots in the instance o.
Conflict (addGeneralization(p,r) , newObject(c1,o)) $\leftrightarrow$ IsA(c1, subtype(r))
[Conflict 2] As a consequence of the generalization the Association a1 may inherit new LinkEnds that will not have the corresponding linkRoles in the link k.
Conflict(addGeneralization(p,r) , newLink(a1,k)) $\leftrightarrow$ IsA(a1, subtype(r))
[Conflict 3] The instance o does not have the corresponding slot for the new Attribute f. If the feature is an Operation, there is not structural conflict, but the behavior of the new instance may be different from the expected behavior.
Conflict(addFeature(c,f) , newObject(c1,o)) $\leftrightarrow$ c=c1
[Conflict 4] The Link k does not have the corresponding linkRole for the new LinkEnds e.
Conflict(addConnection(a,e), newLink(a1,k)) $\leftrightarrow$ a=a1

5 Behavioral Creation vs. Behavioral Creation

	setBehavior(c1,h1)
setBehavior(c,h)	conflict

Conflict(setBehavior(c,h), setBehavior(c1,h1)) $\leftrightarrow$
[1] (c=c1 $\wedge$ h$\neq$h1)
[2] $\vee$ (c$\neq$c1 $\wedge$ IsA (c,c1) $\wedge$ $\neg$refinement(h, h1))
[3] $\vee$ (c$\neq$c1 $\wedge$ IsA (c1,c) $\wedge$ $\neg$refinement(h1, h))

[Conflict 1] There is conflict if both actions are attaching a different StateMachine to the same Classifier.

[Conflicts 2 and 3] There is conflict if both classes are related by generalization. The attached StateMachines must satisfy the refinement relationship.

[Conflict 4] If the intersection between specified and referenced elements is not empty, there is a capture conflict, because an action is using an operation (for example into some effect clause of the introduced state machine) while the other action is modifying the expected behavior of such operation.
Warning(setBehavior(c,h), setBehavior(c1,h1)) $\leftrightarrow$ c$\neq$c1 $\wedge$
(specifiedOperations(h)$\cap$referencedElements(h1)$\neq\emptyset$ $\vee$ specifiedOperations(h1)$\cap$referencedElements(h)$\neq\emptyset$)

7 Behavioral Creation vs. System Creation

	newObject(c1,o)	newLink(a,k)
setBehavior(c,h)	conflict	no conflict

There is not conflict between these actions, but the behavior of the new instance may be different from the expected behavior:
$\neg$Conflict(setBehavior(c,h), newObject(c1,o))
$\neg$Conflict(setBehavior(c,h), newLink(a,k))
Warning(setBehavior(c,h), newObject(c1,o))$\leftrightarrow$ c=c1

6. Related Work

Most work in evolution of the system specification address the problem of structural evolution (e.g. change of the inheritance hierarchy, adding a new class) for example the works of [Bergstein 97, Kesim and Sergot 96, Bertino et al. 98], but do not deal with behavioral evolution (e.g. changing the way an object reacts to a message).

About the problem of consistency of class libraries and frameworks in evolution, Mira Mezini in [Mezini 97], divides these problems in vertical and horizontal evolution conflicts. Horizontal conflicts occur when changes to a base class invalidate inheritors, while vertical conflicts occur when the base class is extended by an inheritor in a way that was not anticipated by the base class designer. She suggests an automatic consistency maintenance system, where designers are enables to formulate properties of the base module to be propagated to the inheritors during composition.

The mechanism of "Reuse Contracts" [Steyaert et al. 96, Lucas 97] provides interface descriptions that partially document the internal structure of components. A reuse contract is a set of interacting participants. Reuse contracts can only be adapted by means of reuse operators. In [Mens et al. 98], the authors try to translate the idea of reuse contracts in order to cope with reuse and evolution of UML models. They provide a precise semantics to reuse in UML, which allows one to detect a number of reuse conflicts automatically. The reuse contract approach deal with structural evolution but covers only a few cases of behavioral evolution.

There exist a number of works addressing the problem of providing formal semantics for the UML. *Although* abstract syntax and static semantics of the language are properly covered by most of these works, dynamic semantics is not treated in a precise way. For example

The description of the UML [UML97] gives a precise notion of what the abstract syntax of the language is. However, it does not cope with semantics because it does not consider dynamic semantics of model evolution and express dynamic semantics of the model (the meaning of the UML constructs) by natural language.

Lano and Biccaregui in [Lano and Biccaregui 98] propose an axiomatic semantics of UML notation, using structured theories in temporal logic. Transformations on UML models, such as adding a class or association are represented as theory extensions in the formalism.

The Precise UML group [Evans et al. 98] aims at a precise semantic model for UML diagrams. They describe abstract syntax, semantic domain and a mapping from syntax to semantics in the specification language Z. But no precise semantics of model evolution is described.

7. Conclusion

Metamodeling is a very promising technique for the definition of multiple view languages like the Unified Modeling Language UML. Using a metamodel, it is possible to determine how all the different models are related in order to constitute the whole system.

In this paper we have defined an evolution mechanism with formal semantics using the metamodeling methodology [Geisler et al.98] based on dynamic logic. A remarkable feature of the metamodeling methodology is the ability to define the relation of intentional and extensional entities within one level, allowing not only for the description of structural relations among the modeling entities, but also for a formal definition of structural constraints and dynamic semantics of the modeled entities. While dynamic semantics on the extensional level means run-time behavior, dynamic semantics on intentional level describes model evolution in the system life cycle.

By animating the transition system defined by the M&S-theory it is possible to simulate the behavior of the specified system and also it is possible to analyze the behavior of the system after evolution of its specification (either structural evolution or behavioral evolution or both). It is possible to express consistency rules between different UML diagrams, and to validate these rules after evolution. In this way consistency of the software system through evolution is guaranteed.

The main contributions of the proposed evolution mechanism are:

-Primitive evolution actions are identified and classified.

-It is possible to define applicability conditions, that is to say conditions under which an evolution action is semantically correct.

-The formal definition of the evolution actions allows us to understand and specify change propagation. The impact of a change in one entity on other entities can be specified.

-It is possible to identify and detect conflict situations originated by the combination of evolution actions.

-Semantics preserving model transformation can be defined: transformation of models is naturally expressed by intentional-object-manipulating methods

References

[Bertino et al.98] E.Bertino, E.Ferrari, G.Guerrini and I.Merlo, Extending the ODMG Object Model with time, proceedings of ECOOP'98, Lecture Notes in Computer Science 1445, July 1998.

[Bergstein 97] Paul Bergstein, Maintenance of object-oriented systems during structural evolution, Theory and Practice of Object Systems, V.3,N.3, John Wiley &Sons, Inc, 1997.

[Evans et al 98] Andy Evans, Robert France, Kevin Lano and Bernhard Rumpe, Developing the UML as a formal modeling notation. In Muller and Bezivin editors, UML'98 Beyond the notation, International workshop, France, 1998.

[Geisler et al 98] Robert Geisler, Marcus Klar and Claudia Pons, Dimensions and Dichotomy in Metamodeling, proc. of Third BCS-FACS Northern Formal Methods Workshop, Ikley, UK, September 1998.

[Kesim and Sergot 96] F.Kesim and M.Sergot. A logic programming framework for modeling temporal objects, IEEE Transactions on knowledge and data engineering, vol.8,no.5, October 1996.

[Kutsche and Sünbül 99] R.-D. Kutsche and A. Sünbül. A Meta-Data Based Development Strategy for Heterogeneous, Distributed Information Systems. In Proc. 3rd IEEE Metadata Conference, Bethesda, Maryland, April 6-7 1999. IEEE Computer Society.

[Kutsche and Sünbül 98] R.-D. Kutsche and A. Sünbül.Metadata Support for Evolutionary Software Systems. In Proc. XVIII International Conference of the SCCC, pages 84-90, Antofagasta, Chile, Nov 9-14 1998, IEEE Computer Society.

[Lano and Biccaregui 98] Formalizing the UML in Structured Temporal Theories, Kevin Lano, Jean Biccaregui, Second ECOOP Workshop on Precise Behavioral Semantics, TUM-I9813, Technische Universitat Munchen.

[Lucas 97] Carine Lucas, "Documenting Reuse and evolution with reuse contracts", PhD Dissertation, Programming Technology Lab, Vrije Universiteit Brussel, September 1997.

[Mezini 97] M.Mezini, Maintaining the consistency of class libraries during their evolution. In proceedings OOPSLA'97, ACM Sigplan notices, pag.1-21, October 1997.

[Mens et al.98] T.Mens, C.Lucas and P.Steyaert, Giving presice semantics to evolution in the UML, In PSMT Workshop on Precise Semantics for Software Modeling Techniques, Ed: M.Broy, D.Coleman, T.Maibaum, B.Rumpe, Technische Universitat Munchen, Report TUM-I9803, April 1998.

[Steyaert et al. 96] P.Steyaert, C.Lucas, K.Mens and T.D'Hondt. Reuse Contracts: Managing the evolution of reusable assets. In proceedings of OOPSLA'96, New York, Oct 1996.

[Odell95] James Odell. Meta-modeling. In OOPSLA'95 Workshop on Metamodeling in OO, October 1995.

[Pons et al. 99] C.Pons, G.Baum, M.Felder, Foundations of Object-oriented modeling notations in a dynamic logic framework, In Fundamentals of Information Systems, Chapter 1, T.Polle,T.Ripke,K.Schewe Editors, Kluwer Academic Publisher, 1999.

[Pons 99] C.Pons, The M&S-theory. Technical report, Lifia, Computer Science Department, Universidad Nacional de la Plata. April 1999.

[UML 97] The Unified Modeling Language (UML) Specification – Version 1.1, September 1997. Joint Submission to the Object Management Group (OMG), ver http://www.omg.org.

[UML 97 (a)] UML Notation Guide, Version 1.1, September 1997. Part of [UML 97]

[UML 97 (b)] UML Semantics, Version 1.1, September 1997. Part of [UML 97].

[Wieringa and Broersen 98] R.Wieringa and J.Broersen, Minimal Transition System Semantics for Lightweight Class and Behavior Diagrams, In PSMT Workshop on Precise Semantics for Software Modeling Techniques, Ed: M.Broy, D.Coleman, T.Maibaum, B.Rumpe, Technische Universitat Munchen, Report TUM-I9803, April 1998.

Appendix 2: The M&S-Theory

Specification of Association	
Sorts	Association
Taxonomy	Association $\leq$GeneralizableElement Association $\leq$ Relationship
Updatable functions	
	connections: Association $\rightarrow$ Seq of AssociationEnd *Additional functions* allConnections:Association $\rightarrow$ Seq of AssociationEnd
Updatable predicates	
Actions	
	addConnection: Association x AssociationEnd $\rightarrow$Action deleteConnection: Association x AssociationEnd $\rightarrow$Action
Axioms	$\forall$a: Association $\forall$e,e1,e2:AssociationEnd
Static axioms	
	axioms for additional functions: [1] allConnections returns the set of all AssociationEnds of the Association itself and all its inherited AssociationEnds. $\text{allConnections}(a) = \text{connections}(a) \cup (\bigcup s\in \text{supertypes}(a)\ \text{allConnections}(s))$ [2]connectedElements (a) returns a sequence containing all the classifiers connected by the association. connectedElements(a) = map type connections(a) .... *well-formedness axioms:* [1] The AssociationEnds must have a unique name within the Association. $\forall e1,e2\in \text{allConnections}(a)\ \text{name}(e1)=\text{name}(e2) \rightarrow e1=e2$ [2] At most one AssociationEnd may be an aggregation or composition. $\forall e1,e2\in \text{allConnections}(a)$ $\text{aggregation}(e1)<>\#\text{none}\wedge\text{aggregation}(e2)<>\#\text{none}\rightarrow e1=e2$ [3] If an Association has 3 or more AssociationEnds then no AssociationEnd may be an aggregation or composition. $\text{size}(\text{allConnections}(a))>2\rightarrow \forall e\in \text{allConnections}(a)\text{aggregation}(e)=\#\text{none}$ ....
Dynamic axioms	
	$\langle\text{addConnection}(a,e)\rangle \mathit{true} \rightarrow e\notin \text{allConnections}(a)$ $[\text{addConnection}(a,e)]\ \text{Exists}(e) \wedge e=\text{last}(\text{connections}(a)) \wedge\text{association}(e)=a \wedge e\in \text{associationEnds}(\text{type}(e))$ $\langle\text{deleteConnection}(a,e)\rangle \mathit{true}\rightarrow e\in \text{connections}(a)$ $[\text{deleteConnection}(a,e)]\ \neg\text{Exists}(e)\wedge e\notin \text{connections}(a)\ )\wedge e\notin \text{associationEnds}(\text{type}(e))$
End specification of Association	

Specification of Classifier	
Sorts	Classifier
Taxonomy	Classifier$\leq$GeneralizableElement
Updatable functions	
	features: Classifier$\rightarrow$Seq of Feature associationEnds: Classifier$\rightarrow$Set of AssociationEnd *Additional functions* associations: Classifier$\rightarrow$Set of Association oppositeAssociationEnds: Classifier$\rightarrow$Set of AssociationEnd ....
Updatable predicates	
	additional predicates *DirectPartOf*: Classifier x Classifier *PartOf*: Classifier x Classifier
Actions	

	addFeature: Classifier x Feature → Action deleteFeature: Classifier x Feature → Action
Axioms	∀c:Classifier ∀f,f1,f2:Feature ∀e:AssociationEnd
Static axioms	
	axioms for additional predicates [1] the *DirectPartOf* predicate: *DirectPartOf*(c1,c2) ↔ ∃e:AssociationEnd (e∈allOppositeAssociationEnds(c1) ∧ aggregation(e)≠#none ∧ type(e)=c2) [2] the *PartOf* predicate: *PartOf=DirectPartOf** *Well-formedness axioms* [1]No Attributes may have the same name within a Classifier ∀f,g∈attributes(c) name(f) = name(g) → f=g [2] No Operations may have the same signature in a Classifier. ∀f,g∈operations(c) hasSameSignature(f,g)) →f = g [3]No opposite AssociationEnds may have the same rol-name within a Classifier ∀f,g∈oppositeAssociationEnds(c) name(f) = name(g) → f=g [4] The name of an Attribute cannot be the same as the name of an opposite AssociationEnd. ∀f∈oppositeAssociationEnds(c) ∀g∈allAttributes(c) name(f) ≠ name(g)
Dynamic axioms	
	⟨addFeature(c,f)⟩*true* →f∉allFeatures(c) [addFeature(c,f)] Exists(f) ∧ f∈features(c) ∧ owner(f)=c ⟨deleteFeature(c,f)⟩*true*→ f∈features(c) [deleteFeature(c,f)] ¬Exists(f)∧f∉features(c)
End specification of Classiffier	

Specification of Instance	
Sorts	Instance
Taxonomy	Instance≤DataElement
Updatable functions	
	slots: Instance → Set of AttributeLink linkEnds: Instance → Set of LinkEnd classifier: Instance → Classifier ....
Axioms	∀i:Instance
Static axioms	
	well-formedness axioms: [1] the AttributeLinks matches the declarations in the Classifier. ∀l:AttributeLink(l∈slots(i) ↔attribute(l)∈allAttributes(classifier(i))) [2] the links matches the declarations in the Classifier. ∀l:Link(l∈allLinks(i) →association(l)∈allAssociations(classifier(i))) [3] An Instance may not belong by composition to more than one composite Instance. ∃e1,e2∈oppositeLinkEnds(i) ((aggregation(associationEnd(e1))=#composite ∧ aggregation(associationEnd(e2))=#composite) → e1=e2) [4] Satisfaction of Constraints. Constraints always evaluate true. ∀c∈allConstraints(classifier(i)) (eval(c)[self:=i] = true) [5] symmetry l∈linkEnds(i)↔ instance(l)=i
End specification of Instance	

Appendix 1: Classification of primitive evolution actions

	Creation	Deletion	Modification
STRUCTURAL	-adding a new Subpackage, Classifier or Relationship to a Package addSubpackage: Package x Package addClassifier: Package x Classifier addAssociation: Package x Association add Generalization: Package x Generalization -adding new features to a Classifier, addFeature: Classifier x Feature -adding new associacionEnds to an association, addConnection: Association x AssociationEnd	deleting an existing Subpackage, Classifier or Relationship from a Package deleteSubpackage: Package x Package deleteClassifier: Package x Classifier delete Association: Package x Association delete Generalization: Package x Generalization -deleting an existing feature from a Classifier, deleteFeature: Classifier x Feature -deleting an existing associacionEnds from an association, deleteConnection: Association x AssociationEnd	-Renaming a model Element setName: ModelElement x Name -modifying the characteristics of an existing association. setAggregation: AssociationEnd x AggregationKind setChangeable: AssociationEnd x ChangeableKind setMultiplicity: AssociationEnd x Multiplicity ..etc... -modify an existing feature of a classifier setOwnerScope: Feature x ScopeKind setVisibility: Feature x VisibilityKind -modify an existing attribute of a classifier setInitialValue: Attribute x Expression setChangeable: Attribute x ChangeableKind ...etc... -modify the signature of an existing operation. addParameter: BehavioralFeature x Parameter deleteParameter: BehavioralFeature x Parameter - add or delete a referenced element into a Package addReferencedElement: Package x ModelElement deleteReferencedElement: Package x ModelElement
BEHAVIORAL	-Attaching a new StateMachine to a Classifier setBehavior: Class x StateMachine	-Deleting a StateMachine form the model cancellBehavior: Class	-Modifying the behavioral characteristics of an operation. setPrecondition: Operation x BooleanExpression setPostcondition: Operation x BooleanExpression setImplementation: Operation x ProcedureExpression -Modifying an existing StateMachine from the model addSubState: CompositeState x State deleteSubState: CompositeState x State addInternalTransition: CompositeState x Transition deleteInternalTransition: CompositeState x Transition - modifying transitions setTrigger: Transition x Operation setGuard: Transition x Guard setEffect: Transition x Seq of Act
CONSTRAINT	Attaching a new constraint to a model element addConstraint: ModelElement x Constraint	Detaching an existing constraint from a model element deleteConstraint: ModelElement x Constraint	Modifying an existing constraint from the model setBody: Constraint x BooleanExpression
SYSTEM	Creation of objects and links newObject:Classifier x Object newLink : Association x Link	Destruction of objects and links destroy: Object destroy:Link	Modification of objects and links: local invocations and call actions update: Object x Name x Instance -.-: Object, Message