Extending a Graphic Modeling Language to Support Partial and Evolutionary Specification

Yong Xia and Martin Glinz
Institut für Informatik der Universität Zürich
Winterthurerstr. 190, CH-8057 Zurich, Switzerland
{xia, glinz}@ifi.unizh.ch

Abstract

The notion of partial and evolutionary specification has gained attention both in research and industry in the last years. While many people regard this just as a process issue, we are convinced that it is as well a language problem. Unfortunately, UML is not expressive enough to deal with evolutionary information in the system. In this paper, we propose an extension to a graphic modeling language called ADORA *which is developed in our research group. We conservatively extend the semantics of some* ADORA *constructs so that intentional incompleteness can be expressed in the language and define a calculus for refining such specifications. With the help of these extensions, evolutionary specifications can be written in a controlled and systematic way. As the language and its extensions are formally defined, the consistency of evolutionary refinements can be checked mechanically by a tool.*

1 Introduction

The notion of partial and evolutionary specification has gained attention both in research and industry in the last years. Software is more and more being developed in an evolutionary, incremental manner. The distinction between development of new systems and evolution of existing ones begins to disappear [5]. While many people regard this just as a process issue, we are convinced that it is as well a language problem.

As mentioned in [5], "change" is a constant in software development. The change of requirements requires that the software system be developed in an evolutionary way. There are two types of changes: one are planned changes, the other are unexpected, unforeseen ones.

For unforeseeable changes in software requirements, only a sophisticated process can help. Nothing can be done at the language level.

However, some changes are predictable. For example, because of the limited resources (lack of manpower, time, etc.), some system requirements cannot be implemented in the first version. Some system functions need to be further augmented in the next version. In such situations, we know the requirements that will change and can evolve the specification in a *planned* way. In this case, extensions on the language level for supporting the incremental development of a specification are useful and necessary: without proper language support, the process will not work well.

Currently, UML [6] is a *de-facto* industry standard for graphic modeling languages specifying software requirements. Use case diagrams, sequence diagrams, and collaboration diagrams in UML are thought of as partial models, which show a part of the system information. Because we cannot at one time specify all aspects of a more and more complex system, whose specification more or less constantly changes, partial models are the means for describing a system in an evolutionary manner. UML has nine different types of diagrams, each of which describing certain aspects of a system and showing only a part of that system. The developers can just concentrate on the parts where the information on certain aspects or certain parts of the system is available in the current development phase. If, for example, the general structure of a subsystem is known, but its behavior is yet unclear, only class diagrams need to be given. The statechart diagrams can be temporarily omitted and added in a later incremental step.

However, there are problems with UML. UML is a set of loosely coupled sublanguages (its nine types of diagrams). There is nearly no enforcement of integrity and consistency checking among the different diagrams at the language level. The UML metamodel provides only weak syntactic interconnections between the concepts expressed in different sublanguages. The usage of OCL in the metamodel is restricted to well-formedness rules within a sublanguage.

UML also does not deal with two problems that arise in the context of partial and evolutionary specifications:

Proceedings of APSEC 2004, the 11th Asia-Pacific Software Engineering Conference, Busan, Korea.

- *Among different aspects*. Incomplete information in one aspect may bring (side) effects to other aspects. For example, an incomplete use case diagram, in which some requirements in current development phase is unknown or some design decision still cannot be made, will affect the construction of statecharts or class diagrams. On the other hand, the already developed model has also effects on the future refinement of that use case.
- *Within one aspect*. Incomplete information brings (side) effects within the same aspect. For instance, a partially specified class affects the specification of the surrounding classes in the class diagram.

In a word, UML does not have sufficient language constructs and mechanism, which

- record *evolutionary information* (e.g. *still be incomplete or partial, will be further evolved*, etc.) of the model elements[1];
- model effects of an incomplete part of a system to other parts;
- support a software development process to make evolutionary specification be developed in a systematical way.

Actually, the characteristic of UML, a set of loosely coupled sublanguages, also brings the difficulty of naturally solving the above problems. In our research group in Zurich we have developed a modeling language for requirements and architecture called ADORA.

In this paper, we focus mainly at the language level and study how a partial and evolutionary specification can be expressed in the ADORA language and where the language has to be modified or extended for this purpose. As consequence, we *syntactically* and *semantically* extend language elements of ADORA so that evolutionary specifications can be written in a controlled way. Sometimes we call our extended language ADORA$^+$ to distinguish it from the original language.

2 A short introduction to ADORA

In this section, we give a brief introduction to ADORA, as we will use the ADORA language as a sample graphic modeling language in the rest of this paper. ADORA is a modeling technique for requirements and software architecture[4][9]. The acronym stands for Analysis and Description of Requirements and Architecture. Figures 1 and 2 (taken from the specification of a distributed heating control system) give an impression how ADORA models look like. At a first glance, ADORA diagrams look similar

[1]In UML, we can use some mechanisms, such as *Notes*, to explicitly record evolutionary information. However, specifying evolutionary specification with those kinds of general purpose language constructs is just at an application level. This makes further system refinement in a controlled and formal way nearly impossible.

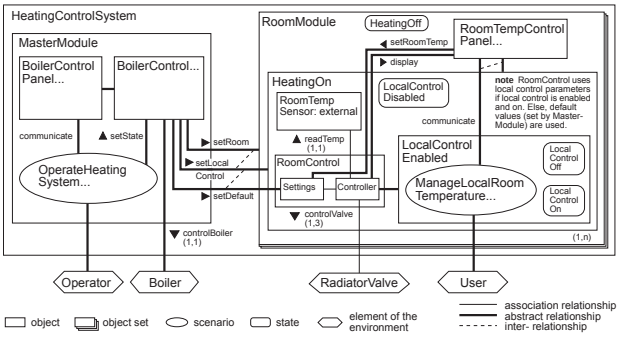

Figure 1. An ADORA view of the heating system: base view + structural view + context view

to UML diagrams. However, there are fundamental differences between ADORA and UML [4].

2.1 Basic features of ADORA

In this subsection, we summarize the distinguishing features of ADORA.

Using abstract objects (instead of classes) as the basis of the model. Class models are inappropriate when more than one object of a class and/or structural nesting of objects has to be modeled [4]. Therefore, ADORA uses *abstract, prototypical* objects instead of classes as the conceptual core of the language. For example, in the sample heating control system (see Figure 1), there is a single Master Module, but multiple room modules. In ADORA, we model these entities as abstract objects and thus can make these cardinalities immediately visible. Moreover, the Boiler Control Panel in the Master Module and the Room Control Panel in the Room Module may have the same type. Hence, they would not be distinguishable in a class model, while with abstract objects, we can model them separately and place them where they belong.

Integration of all aspects of the system in one coherent model. An ADORA model integrates all modeling aspects (structure, data, behavior, user interaction ...) in one coherent model. This allows us to introduce strong rules for consistency and completeness of models, reduces redundancy, and makes the model construction more systematic.

Using an integrated model does of course not mean that everything is drawn in one single diagram. From the integrated model, we can generate *aspect views* pertaining to a given *aspect*. The so-called *base view* of an ADORA model consists of the hierarchical structure of objects only. Aspect views are generated by combining the base view with all information that is relevant for the selected aspect.

For example, Figure 1 shows the structural view (which shows the static structure of the system by combining the base view with directed relationships between ob-

jects/object sets) and the context view (which shows all actors and objects in the environment of the modeled system and their relationship with the system) of the whole heating control system. Figure 2 shows the behavior view (which shows the dynamic behavior of the system by combining the base view with a statechart-based state machine hierarchy) of the Room Module in our heating control system. Figure 3 shows the details of Scenario *ManageLocalRoomTemperature* using the notation of scenariochart, whose style is derived from Jackson Diagrams. This is a part of the user view of the system.

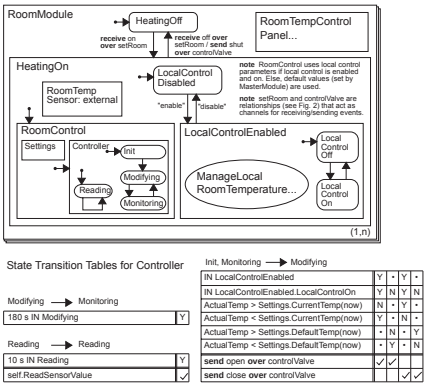

Figure 2. A partial ADORA model of the heating system: base view + behavior view

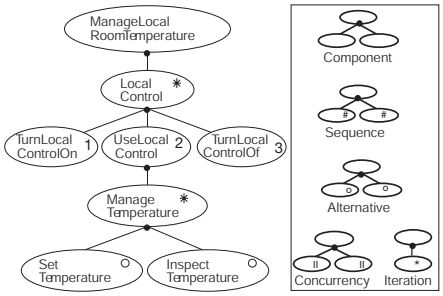

Figure 3. A scenariochart modeling the structure of the *ManageLocalRoomTemperature* scenario.

Hierarchical decomposition. ADORA systematically uses hierarchical decomposition for structuring models. With the use of abstract objects, abstraction and decomposition mechanisms can easily be introduced into the language. We recursively decompose objects into objects (or other elements, like states). So we have the full power of object modeling at all levels of the hierarchy and only vary the degree of abstractness: objects on lower levels of the decomposition model small parts of a system in detail, whereas objects on higher levels model large parts or the whole system on an abstract level.

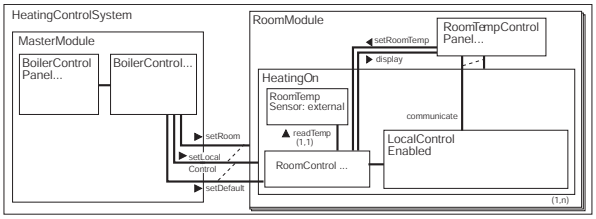

Figure 4. A partial ADORA model with base view and structural view (on a highly abstracted level only with the most fundamental objects and their relationships).

2.2 The view concept

Hierarchical decomposition also is a powerful means for viewing a model at different levels of abstraction ranging from abstract overviews (for example, the view of Figure 4 to detailed views such as the one given in Figure 1.

Having a view concept with both aspect and hierarchical views is crucial for making an integrated model work in practice, because it allows the modeler to generate diagrams that only show what she or he is currently interested in.

A given view is transformed into a more abstract one by hiding model elements from that view. Conversely, a view is transformed into a more concrete and detailed one by displaying model elements that were previously hidden. A view transforming transaction which transforms a syntactically correct view into another syntactically correct view is called a *view transition*.

In an abstract view of an ADORA model, we do not just omit information. Instead, all places where model elements have been hidden are marked in the view. We achieve this using two concepts: the is-partial indicator and the abstract relationship.

Every object that is not shown in full detail in a given view is tagged with an *is-partial indicator* (visualized with three dots appended to the names of these objects).

If the structural aspect is included in a view, the relationships that exist between objects are displayed. Now, if we make a view transition that hides an object which has relationships to other objects, these relationships must also be hidden. In order to indicate where we have such hidden relationships, we generate so-called *abstract relationships* on the next higher level of the decomposition hierarchy.

Figure 5 shows an example. In Figure 5a.1 we have a view showing a model in full detail, while in Figure 5a.2 the interior of object X is abstracted. Hence X is tagged with an is-partial indicator (the three dots) in Figure 5a.2. Abstracting the interior of object X (i.e. hiding objects A, A' and C) implies that the relationships s and t have to be hidden, too. In order to indicate that we have abstracted away some relationships here, an abstract relationship u (drawn

with a thick line) is generated in the view a.2.

2.3 Defining the syntax and static semantics

The syntax of ADORA is defined by an EBNF-based method, which in essence is an extended attributed string grammar: the terminals in the grammar are interpreted as two dimensional objects (i.e. the basic language elements, such as object, association, etc.). The attributes in the grammar express the simple static semantics of the language.

The dynamic constraints in the static semantics (in particular, the view transitions) are specified by a set of *operational rules*, whose logical structure is similar to the rules being used in the definition of operational semantics for textual programming and specification language [8]. Note that our notation looks a little different from the conventional operational semantics such that the rules in our notation are easier to read for humans. The operational rules formally define all possible view transitions, thus guaranteeing that views of ADORA models are always well-formed. In [10], the definition techniques of ADORA syntax and semantics are explained in detail.

3 Extension of the structural view

3.1 Extended semantics

The syntax (and static semantics) of ADORA (see above) is intended to deal with incomplete views of a specification which is (*intentionally*) complete. When a model is shown with all its views in full details, there should be no *is-partial indicators* and *abstract relationships* in the view. Now we extend this notion in a straightforward way: from incomplete views to incomplete models.

Consider the situation in Figure 5 again. Suppose a situation where we are incrementally developing this model and where we have Z, X, Y, and B, but do *not yet know* the details of X. Obviously, drawing a model like Fig 5b) would be an adequate representation of this situation. In this case, the abstract relationship from X to B is deliberately drawn to model the fact that there will be some relationship from objects within X to B (In contrast to that, in Figure 5a2, the abstract relationship is generated as a representation of the hidden relationships t and s.).

This example demonstrates that we can model the structural view of a partial specification by *overloading* the meaning of the *is-partial indicator* (the three dots) and the concept of abstract relationship.

In the case of complete models, they indicate existing information which is hidden from the current view. In the new case of partial/incomplete models, they can either stand for hidden information as before or for information *when details have not yet been modelled*.

The above introduced extension supports *partial* specifications. This in turn is required for supporting the planned evolution of a requirements specification in an incremental software development project.

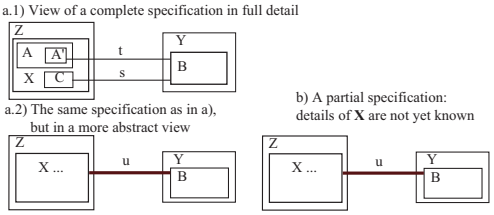

Figure 5. A complete specification vs. an incomplete specification.

Now we explain in detail the semantic extensions on these two language elements: the *is-partial indicator* and the *abstract relationship*.

An object A with an is-partial indictor means:

- some components of X are hidden from the view (the original semantics); *and/or*
- the specification of X is not yet complete and will be refined by adding more components in the further development (the extended semantics, which supports the mechanism of evolutionary specifications).

Note that the first situation usually takes place during the view transitions by applying operational rules. In the second situation, we need to manually add or delete is-partial indicators.

The abstract relationship construct is extended similarly. An abstract relationship u, which connects object X and object B originally implies:

- during the view transition by applying operational rules, an association, which connects
  - a component of X and a component of B; or
  - a component of X and B; or
  - X and a component of B,

  is hidden.

Now it includes a new meaning:

- an association, the details of which are still unclear, will be set up between
  - a component of X and a component of B; or
  - a component of X and B; or
  - X and a component of B

  in the future refinement.

Unlike the original definition, in which the *is-partial indicators* are automatically appended and the *abstract relationships* are automatically generated through the view transitions, we can now also manually add an abstract relationship or make an object partial by appending an is-partial indicator manually to record evolutionary information. As shown in Figure 5b, we manually make Object X partial and add an abstract relationship u, when the details of Object X are still not clear.

The is-partial indicator and the abstract relationship keep

the information of "specification being incomplete". In Figure 5b, they tell requirements analysts/software architects and their team-members that object X is not yet completely specified and some associations between a component of X and B may be set up in the further refinement.

Furthermore, if high-level design decisions are taken based on partial specifications (which is usually the case in incremental software development), the explicit model of partiality provides the designers with information about what to encapsulate in modules and where to design interfaces with special care. For example, the requirements represented by object X in Fig 5b should be realized in a single, encapsulated module with a carefully designed interface to the module(s) that realize(s) B.

We decided to overload the existing constructs for expressing the extended semantics instead of introducing new notations because the original meanings and the extended ones are closely related in ADORA$^+$.

3.1.1 Extended definition of well-formedness

With the extended semantics, we should also extend the definition of *well-formedness* for the extended language (ADORA$^+$). Consider the following example:

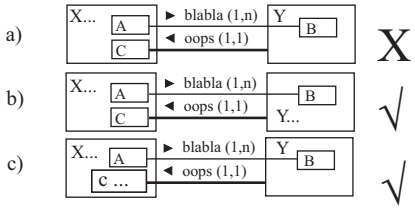

Figure 6. The first diagram is not well-formed; and the second and the third diagram are well-formed.

The three diagrams show some incomplete specifications in the structural view. From Figure 6a, we don't know the details of Object X (e.g. its properties, the number of its embedded objects and the corresponding properties). However, no matter how Object X is completely specified in future, the refined specification can not evolve to Figure 6a using the available operational rules for complete specification [9]. The reason is: both Object C and Object Y don't have any hidden components; however the abstract relationship *oops* implies that at least one object of them has hidden components. This conflict will not be solved no matter how the surrounding objects, such as Object X, are further refined. In a word, Figure 6a can not be a well-formed view, which is transformed from a well-formed complete specification. Before we give a definition of "well-formedness" for an ADORA$^+$ model based on an incomplete speciation, we first define a related concept.

A complete structural view is defined as a structural view of an ADORA model without model elements (e.g. object, object set, etc.) being hidden.

Figure 5a.1 is a complete structural view of an ADORA model based on a complete specification; while Figure 5b is a complete structural view of an ADORA model based on an incomplete specification. Note that a complete structural view of an ADORA model based on an incomplete specification should look the same as a partial structural view (in which some model elements are hidden) of an ADORA model based on a complete specification. This is the language design principle of our conservative extension of the ADORA structural view. Now, the formal definition of a well-formed model is given as follows.

An ADORA model M based on an incomplete specification is defined to be *well-formed* when there exists a well-formed ADORA model M' based on a complete specification whose complete structural view can be transformed into the complete structural view of M by view transitions.

The view transitions in turn are defined in the grammar by the operational rules [9] (see also next sub-subsection). Through this definition, we can theoretically determine the well-formedness of an evolutionary specification (c.f. Figure 6).

3.1.2 Conservative extension of operational rules

Our conservative extension should keep the original good features of the language. We extend the operational rules, which guide the view transition in the structural view of ADORA$^+$. As shown in Figure 7, the view transitions in ADORA and ADORA$^+$ look very similar. Here we explain the extension through an example in Figure 7.

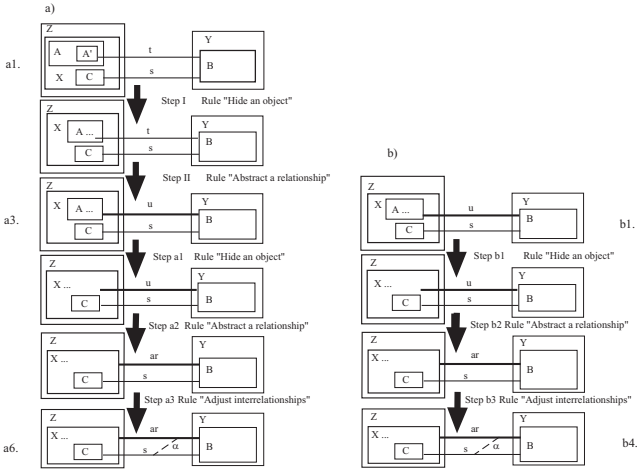

Figure 7. Abstracting an object A: a) a view transition based on a complete specification; b) a view transition on an incomplete specification. The Diagram a1, a3, a6, b1 and b4 are *well-formed*; the other intermediate ones are not.

After the formal definition, the intuitive meaning of the rules and explanation on extension will be given. The semantic functions, operators and notations in the operational rules are explained as follows.

Let M be an ADORA model with X, Y, Z : $\underline{model\_element}$[2], A, B, C, D : $\underline{object}$, r : $\underline{association}$, u : $\underline{abstract\_relationship}$ and let Γ be a partial view of M.

- *Pretrace* is an attribute of a language element (e.g. object, object set, etc), which captures the hierarchical structure. The pretrace of a language element U is an ordered set of model elements that U is embedded in. It is expressed as "*.pre*". In Figure 5, we have $A.pre = \{A, X, Z\}_{ordered}, C.pre = \{C, X, Z\}_{ordered}$, $X.pre = \{X, Z\}_{ordered}$, and $Z = \{Z\}_{ordered}$.
- The meaning of $\supset$ and $\supseteq$ are the usual mathematical superset relations. Due to the definition of pretraces, we have $A.pre \supset X.pre$ and $A.pre \supset Z.pre$. Additionally, we define a special superset relation $\sqsupset$ for pretraces: $U.pre \sqsupset V.pre$, if and only if U is directly embedded in V. In Figure 5, we have $A.pre \sqsupset X.pre$ and $X.pre \sqsupset Z.pre$.
- The meaning of the $\subset$ and $\sqsubset$ operators is extended to relationships as follows.
  - $r(A, B) \subset u(C, D)$ if and only if $A.pre \supset C.pre$ and $B.pre \supset D.pre$.
  - $r \sqsubset u$, if and only if $r \subset u$ and there is no u_1 in Γ, such that $r \subset u_1 \subset u$.
- $visible(X, \Gamma)$ is a boolean function that is true if and only if X is visible in Γ.
  For a set of model_elements, $visible(\{X, Y, Z\}, \Gamma)$ means $visible(X, \Gamma) \wedge visible(Y, \Gamma) \wedge visible(Z, \Gamma)$.
- $hidden(X, \Gamma) = \neg visible(X, \Gamma)$ for all X and Γ.
- $partial(A, \Gamma)$ is a boolean function that is true if and only if the name of A in Γ is followed by an *is-partial indicator* (*trailing dots*).

In ADORA and its extension, the effects of changing a property of a model element to other model elements will be exactly recorded and traced. As the real situation is much more complex than what is shown in the example, the semantics is given by a series of sequential steps, each of which being easily readable and implementable. In the following text, we present the operational rules that describe the processes of making two ADORA views more abstract by hiding their objects. The processes are illustrated in Figure 7.

Hide an object

Rule	Hide an object (A)
Given:	$M; \Gamma;\ X : \underline{object\_i}; A : \underline{object}$
Condition:	$(A.pre \sqsupset \overline{X.pre}) \wedge \overline{visible(\{A, X\}, \Gamma)} \wedge$ $(\neg\exists Y : \underline{object} \bullet (Y.pre \supset A.pre \wedge visible(Y, \Gamma)))$
Assertion:	$\overline{hidden(A, \Gamma)} \wedge partial(X, \Gamma)$
Next Rule:	Abstract a relationship (A)

[2]Note that the definition of operational rules (and static semantics) is based on our EBNF syntax definition. In the following rules and their explanation, the names with underlines, such as object, object_i, association, abstract_relationship, are actually the terminal and non-terminal symbols in the grammar, which correspond to the graphical language elements or the categories of the language elements.

. . . . . .

Abstract a relationship

Rule	Abstract a relationship (A)
Given:	$M; \Gamma;\ X : \underline{object\_i}; A, B : \underline{object}; r(A, B) : \underline{association}$
Condition:	$(A.pre \sqsupset \overline{X.pre})$ $\wedge hidden(A, \Gamma) \wedge visible(\{B, X, r(A, B)\}, \Gamma)$
Assertion:	$\exists ar : \underline{abstract\_relationship} \bullet (ar(X, B) \in \Gamma) \wedge$ $hidden(r(A, B)) \wedge (\forall z : \underline{abstract\_relationship}$ $\neg\exists \alpha : \underline{interrelationship} \bullet \alpha(r, z) \in \Gamma) \wedge visible(ar, \Gamma)$
Next Rule:	interrelationships (ar)

Rule	Abstract a relationship (A)
Given:	$M; \Gamma;\ X : \underline{object\_i}; A, B : \underline{object};$ $u(A, B) : \underline{abstract\_relationship}$
Condition:	$(A.pre \sqsupset \overline{X.pre}) \wedge hidden(A, \Gamma) \wedge$ $visible(\{B, X\}, \Gamma) \wedge u(A, B) \notin M \wedge u(A, B) \in \Gamma$
Assertion:	$\exists ar : \underline{abstract\_relationship} \bullet (ar(X, B) \in \Gamma) \wedge$ $u(A, B) \notin \Gamma \wedge visible(ar, \Gamma) \wedge \forall z : \underline{abstract\_relationship}$ $\neg\exists \alpha : \underline{interrelationship} \bullet \alpha(u, z) \in \Gamma$
Next Rule:	Adjust interrelationships (ar)

Rule	Abstract a relationship **: incomplete specification** (A)
Given:	$M; \Gamma;\ X : \underline{object\_i}; A, B : \underline{object};$ $u(A, B) : \underline{abstract\_relationship}$
Condition:	$(A.pre \sqsupset \overline{X.pre}) \wedge hidden(A, \Gamma) \wedge$ $visible(\{B, X\}, \Gamma) \wedge u(A, B) \in M \wedge u(A, B) \in \Gamma$
Assertion:	$\exists ar : \underline{abstract\_relationship} \bullet (ar(X, B) \in \Gamma) \wedge$ $u(A, B) \notin \Gamma \wedge visible(ar, \Gamma) \wedge \forall z : \underline{abstract\_relationship}$ $\neg\exists \alpha : \underline{interrelationship} \bullet \alpha(u, z) \in \Gamma$
Next Rule:	Adjust interrelationships (ar)

. . . . . .

Adjust interrelationships

Rule	Adjust interrelationships (ar)
Given:	$M; \Gamma;\ s : \underline{association}; ar : \underline{abstract\_relationship}$
Condition:	$(s \sqsupset ar) \wedge visible(s, \Gamma) \wedge$ $\neg\exists \alpha : \underline{interrelationship} \bullet \alpha(s, ar) \in \Gamma$
Assertion:	$\exists \alpha : \underline{interrelationship} \bullet \alpha(s, ar) \in \Gamma$
Next Rule:	Adjust interrelationships (ar)

. . . . . .

As showed above, every operational rule has the following format.

Rule	$rule\text{-}name\ (parameters)$
Given:	$M; \Gamma; model\ elements$
Condition:	$predicate\_pre$
Assertion:	$predicate\_post$
Next Rule:	$rule\text{-}name\ (parameters)$

Rule names are used to divide the whole set of rules into three groups: *Hide an object*, *Abstract a relationship* and *Adjust interrelationship*. As each rule has one of the above three names, we may have more than one rule with the same name but with different conditions. A rule is interpreted as follows: for any ADORA model M which contains the given *model elements* and has a view Γ such that *predicate_pre* is true, the application of the rule modifies Γ so that *predicate_post* becomes true. The application of the rule does *not* modify anything that is *not* specified in *predicate_post*. If the **Next Rule** field contains a name, the rule(s) matching this name must be applied next in order to transform a well-formed view Γ eventually into a new view Γ_1. Rule execution stops when the **Next Rule** field is empty or when the

conditions (*predicate_pre*) of all matching rules are false. Parameters may be used to transfer information from a rule to the next one.

In fact, most of the original operational rules can be applied for the incomplete specification without any changes. But there are still some differences between two sets of the operational rules.

In the structural view of an ADORA model, three model elements (abstract_relationship, interrelationship and is-partial indicator) appear in the view, only because some objects and relationships are hidden. In the model *resp.* the complete structural view of the model, they don't exist at all. On the contrary, in the structural view of an ADORA$^+$ model, the *abstract relationship* and the *is-partial indicator*, which record evolutionary information, may exist also in the model and the complete structural view. Therefore, some rules must be extended or added to cope with more general situations.

Let us look at the second group of rules in the example more closely. The first rule in Group "Abstract a relationship" specifies the following three points: (i) after object A is hidden, any association connecting with A should also be hidden in the view; and (ii) an abstract_association ar connecting with X, in which A is directly embedded, should be automatically generated in the view (only in the view, not in the model); and (iii) any interrelationship should be deleted from the view. This rule is applied in *Step II* in Figure 7a. The second rule in Group "Abstract a relationship" specifies also three points. The last two points are the same as the first rule. The first one is different, and it says: after object A is hidden, any abstract_relationship, which connects with A and does not belong to the model (i.e. it *was* automatically generated.), should also be deleted from the view. It is applied in *Step a2* in Figure 7a.

The above two rules are the original ones for the complete specification. However, they are not adequate to deal with the situation of an incomplete specification (c.f. *Step b2* in Figure 7b). Therefore, a third rule, to whose name a syntax sugar "**incomplete specification**" is appended, needs to be added in Group "Abstract a relationship". The first point of this rule specifies: after Object A is hidden, any abstract relationship, which connects with A and belongs to the model (i.e. it is manually generated from users), should also be hidden in the view (note that it should not be deleted.). The remaining two points are the same as those in the first two rules.

The rules in the other groups are extended similarly. From the above explanation, we see some differences between the extended rules and the original ones:

- In a model based on a complete specification, an abstract relationship does not exist in the model, but may exist in a structural view. Therefore, if an abstract relationship should disappear from the current view during the view transitions, it will just be deleted. Even if the two objects, with which it connects, become visible in future, it is not necessary to be generated.
- In a model based on an incomplete specification, an abstract relationship *does* exist in the model. During the view transitions, if it should disappear in the current view, it will not be deleted from the model, but only from the current *view*. When the two objects, with which it connects, become again visible, it must also be visible.

Here, we only show the part of rules, which specify the view transition in Figure 7. Actually, the view transitions in the structural view are rather complex, in which more than 30 operational rules are used to generalize all the situations for the complete specification (e.g. abstract an object, concretize an object, etc.). In order to get a semantics applicable in the extended ADORA, about 10 rules need to be modified or added to the original rules. The strict formal definition, their execution sequence, and detailed explanations on those rules and the corresponding examples can be found in [10]. The formal definition serves two purposes: (i) it provides a sound base for the language and avoids semantic conflicts and inconsistencies in the language; (ii) it is also a formal specification for the ADORA tool, in which the view transitions can be automatically carried out.

3.2 A refinement calculus

Our extended language fits the typical "top-down" approach of software development and system refinement. As ADORA$^+$ is compositional, the objects which specify the general structure of (sub-)systems, can be first defined and be viewed as a composition of several black box components. Then they will be filled in with the relationships and objects describing more details. Again those newly defined relationships and objects need to be refined to the full detail level. *Is-partial indicators* and *abstract relationships* will be used in all situations where an object or a relationship cannot be defined in full detail yet.

In order to control this process and preserve the integrity of the model during evolutionary refinement, we define a *refinement calculus* which is composed of a set of logic constraints. Applying the calculus during refinement makes sure that the evolving model is always well-formed and thus preserves the integrity of the model. In essence, together with the extended operational rules in the last subsubsection, the refinement calculus can also be seen as a *constructive definition* of well-formed model *resp.* views.

Now we give a natural language description of the calculus, which is applied in the following four situations.

making an object partial When system developers think that an object should be further defined, but still cannot decide how this object should be refined, they can just add an *is-partial indicator* after the name of that

object. In this situation there is no special constraint.

adding an abstract relationship When system analysts and software architects think that Object A and Object B or their components should have some relationships, but the details can still not be decided, they can add an abstract relationship between them. The constraints are:

1. either A or B should be partial in the model;
2. if there exists already an abstract relationship between A and B, no new abstract relationship should be added;
3. after the abstract relationship is newly added, some interrelationship should be adjusted. The principle of this $adjustment$ is the same as that of the operational rules in the last sub-subsection.

deleting the is-partial indicator of an object After an object A, which was made "partial" before, is fully specified, it can be made "not partial" again. I.e. the previous manually added *is-partial indicator* of A can be deleted. There is one constraint, which prevents an object from being "not partial": if there exists an abstract relationship connecting A and another object B in the model, and B is not partial in the model, A must remain to be "partial". In this case, the manually added abstract relationship should be deleted first.

deleting an abstract relationship After two objects A and B are fully specified, and no associations will be added between A and B or their components, the previous added *abstract relationship* can be deleted. There are two constraints in this case: (1) the abstract relationship cannot just be deleted in the view, unless no association between the components of A and B is hidden in the view; (2) if the abstract relationship is deleted, some interrelationship should be adjusted.

The formal definition of the refinement calculus has a similar notation as that of operational rules. We give only one example on the situation of "adding an abstract relationship". The formal definition of the complete refinement calculus can be found in [10].

Rule	adding an abstract relationship (ar)
Given:	$M; \Gamma; A, B : \underline{object}; ar : \underline{abstract\_relationship}$
Condition:	$visible(\{A, B\}, \Gamma)$
Assertion:	$man\_g\_absrel(ar(A, B), \Gamma, M) \rightarrow$ $(Constraint_1 \wedge Constraint_2 \wedge Constraint_3)$

where

Constraint$_1$ $\equiv partial(A, M) \vee partial(B, M)$

Constraint$_2$ $\equiv (\forall ar_1 : \underline{abstract\_relationship} \bullet ar_1(A, B) \in \Gamma \rightarrow$ $(ar_1 = ar)) \wedge ar(A, B) \in M$

Constraint$_3$ $\equiv (ar \in \Gamma) \wedge (\forall \alpha : \underline{interrelationship},$ $as : \underline{abstract\_relationship}, r : \underline{association} \bullet$ $(ar \sqsupset as \leftrightarrow \alpha(ar, as) \in \Gamma) \wedge (as \sqsupset ar \leftrightarrow \alpha(as, ar) \in \Gamma) \wedge$ $(r \sqsupset ar \leftrightarrow \alpha(r, ar) \in \Gamma) \wedge (\neg(r \sqsupset as) \rightarrow \alpha(r, as) \notin \Gamma))$

Note that

- The rule structure is nearly the same as that of the operational rules. As the constraints here are much easier, they don't need to be written as a sequence of rules to enhance the understandability. Therefore, the item of "**Next Rule**" is unnecessary.
- $man\_g\_absrel(ar(A, B), \Gamma, M)$ is a boolean function, which is true when the action of "adding an abstract relationship" happens and an abstract_relationship is *manually generated*.
- The three formally specified constraints one to one correspond to the explanations in the natural language.

4 Extension of other aspect views

Because of the limit of the space, here we can only talk about the extension of the behavioral view and the user view. The basic ideas on the extension of other *aspect views* are very similar to what we introduce in this and previous sections.

4.1 The behavioral view

The behavioral view in ADORA models the system behavior by combining the base view with a statechart-like state machine hierarchy. In essence, the behavior view can be seen as a special statechart, in which a state can also be replaced by an object. It is called *Generic Statechart* in ADORA. As explained in [3], a (generic) statechart without inter-level transitions is compositional. Therefore, the extension method used in the structural view can be also used in the behavioral view: for the generic statechart without inter-level transitions, we use the *is-partial indicator* in a stateobject [3] D', which means (i) either during the view transitions, the details in D' are temporarily hidden; or (ii) some (sub-) generic statecharts may be added into D' in a further refinement.

By the way, a refinement calculus in the behavior view is also defined similarly. A formal and complete description of the extension is given in [10].

4.2 The user view

The user view combines the base view with the actors in the system environment, which the scenarios interact with, and all those abstract relationships, which model interactions between scenarios and objects. In the integrated ADORA model, the scenariocharts are usually embedded in the generic statechart or the object hierarchy.

In this section, the effects of incomplete specifications from one aspect to another aspect will be particularly discussed.

4.2.1 The conservative extension

The basic idea on the extension of scenariocharts is nearly the same as those in the structural and behavioral views. The is-partial indicator is appended after the name of a scenario A to mean that (i) either some sub-scenarios of A are

[3] stateobject SO denotes that SO is either a state or an object.

hidden from the view; or (ii) the specification of A is not yet complete and will be refined by adding more sub-scenarios in the further development.

With this extension, evolutionary information of scenarios is recorded. For example, an incomplete scenario S implies that special attention needs to be paid to designing of the scenarios around S, and more importantly, the objects containing or connected to S. For details, see [10].

4.2.2 Integration semantics

As we mentioned before, one advantage of ADORA over UML is that ADORA provides the mechanism to check the consistency among different types of the aspect views. The integrity checking is not only at the application level (model level), but also at the language level (metamodel level). Here we only show two example of the integrity checking.

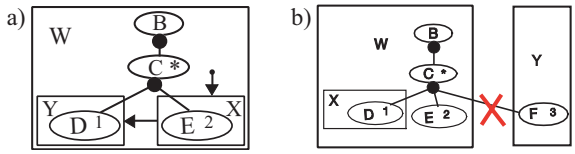

Figure 8. a) integration of a scenariochart and a generic statechart b) integration of a scenariochart into an object hierarchy.

- Conflicts between two aspect views. In Figure 8a, Scenario C can be decomposed into two sequentially executed sub-scenarios: first D then E. At the same time, in Object W there is a generic statechart, where Object X precedes Object Y. This brings a conflict on the temporal order of Scenarios D and E, which should not be allowed.
- Violation of the modularity. Good modularity guarantees the principle of *Information Hiding*. Suppose that Scenarios S is embedded in Object O. a sub-scenario S' of S (S' describes part of the scenario of S) should and must be embedded in Object O or a component of O. In Figure 8b, Scenario F in Object Y, which is not a component of W, destroys the principle of a good modularity.

Usually, the semantics is formally defined in a form, in which the meanings of language constructs are easy to understand. Then a refinement calculus is derived from that formal definition. Derivation process makes the refinement calculus repeat parts or the whole information of the semantics definition. A refinement calculus is defined in a form, in which logic and algebraic inferences are easy to carry out. When the semantics is not difficult to understand, they can also be directly defined in the form of refinement calculus.

4.2.3 A refinement calculus in the user view

When we further refine a *currently not fully specified* scenario with is-partial indicator, the newly specified sub-scenarios should conform to the constraints mentioned in last sub-subsection. The constraints are defined by a set of rules, which compose a refinement calculus.

Here we only show only a part of the refinement calculus, which specifies the constraint in Figure 8b.

Refinement in the User View: ensuring the modularity

Rule	Refinement Scenario(D)
Given:	$M; \Gamma; A, B : \underline{object}; C, D : \underline{scenario}$
Condition:	$part\_of(C, D)$
Assertion:	$((C.pre \sqsupset A.pre) \wedge (D.pre \sqsupset B.pre)) \rightarrow (A.pre \sqsupseteq B.pre)$

... ...

where

- The rule structure is the same as that of the operational rules.
- $sequence\_sce(C, D)$ is a boolean function that is true iff Scenario C is a sub-scenario of D, and the relation between C and D is *sequence*.
- The function $part\_of(C, D)$ means C is a sub-scenario of D. The relation between C and D can be one of the five types specified in the scenario-chart (c.f. Figure 3).

In essence, while the refinement calculi in the previous sections concern the effects of a partially specified part to others in the same aspect view, the rules here, which guarantee the integrity and consistency of the integrated ADORA model, specify the effects of a partial specification in one aspect view to other aspect views. A complete formal definition on the above refinement calculus can be found in [10].

5 Conclusion

A model should record system information, which includes evolutionary information. As evolutionary information is so closely coupled with system information, it should not be separated from system information and classified only into the field of software process. Therefore, the concept of evolutionary specification being supported at the language level is useful and necessary. This paper is a continuation of previous work [9], in which the syntax and static semantics of ADORA are formally defined. Here, we extend the semantics of ADORA so that evolutionary specifications can be written in a controlled and systematic way, while the original good properties of ADORA are perfectly kept. In order to make this paper more understandable, we have tried to avoid too many formal notations. Interested readers find all the formal definitions and procedures in [10].

Discussion of achievements. Just introducing some new constructs to support evolutionary specification is not too difficult. However, every potential extension should be carefully evaluated whether it is really necessary and useful. Here, we achieve a smallest extension: instead of introducing new constructs, we extend the semantics of the original constructs of ADORA to support evolutionary specification. We achieve a conservative extension. That is to say, our

extension satisfies the following two criteria: (i) for the extended operational rules, if we throw away the parts relating to the evolutionary specification, they are the same as, or at least totally consistent with the original rules; (ii) the result of view transitions of applying the extended rules to a specification without any extended constructs is the same as that of applying the original rules. This makes sure that the extension causes no syntactic or semantic conflicts with the original language and that the original features and semantics are maximally kept (e.g. the mechanism of hierarchical decomposition).

Important evolutionary information is formally specified and documented in the extended ADORA models, which can be exploited in the further refinement of the models by applying our refinement calculus.

We give a formal semantics definition and a refinement calculus in a constructive way. This helps particularly for the development of a tool, which automatically checks the syntax and semantics of an ADORA model and mechanically controls the evolutionary development of a software system.

Related Work. Just as what we do for a UML model, we can build an ADORA model view by view. Therefore, building an integrated ADORA model is no more complicated than building a UML model. On the contrary, the mechanism of the integrated model and its corresponding refinement calculus makes the model refinement easier.

Surely our extension approaches and language definition methods for syntax, semantics and refinement calculus can also be applied to other graphical modeling languages with some adjustments. For example, we can extend UML to a "stereotyped UML" with part of the above mentioned features of ADORA. We can also easily translate our semantics and the refinement calculus from a normal first order predicate logic into OCL. However, this extension of UML makes the target model unnecessarily complicated, and brings no more value than ADORA does. Research on a small language can often inspire some new ideas, which can not be achieved by restricting only to a big language such as UML. And keeping great variety in our research community is valuable, even for the further development of UML.

[7] shares a basic idea with ADORA: using a single model. Like most of other works on consistency checking, they mainly concentrate on the refinement process from an architecture model to a detailed design model or coding (BON/Eiffel).

There are lots of work on the definition of semantics and refinement calculus for each individual sublanguage of UML in literature (e.g. [1], etc.). [2] proposes a common formalism relating different models. Some correctness rules based on that formalism are provided to validate the whole model. However, without a clear integration semantics, consistency checking is difficult to be carried out in a language with a set of loosely coupled sublanguages, such as UML.

In our approach, we mainly focus on specification of *partial and evolutionary* information in the *requirement* and *architecture* model. In particular, we study the effects of the partial and evolutionary information on other model elements in the same or other views, and define a refinement calculus for the integrity and consistency checking.

Limitations. As we know, our work on supporting evolutionary specification at the language level is the first try in this field. It has to be done in a very careful way, and the extension is really "conservative". Whether new constructs needed to be introduced is not clear in the current research phase. A radical extension (e.g. introduction of new constructs recording evolutionary information, etc.) needs to be further validated.

Future Work. We will validate the other extension possibilities for ADORA. A process for ADORA should also be proposed correspondingly to better support the evolutionary development of a software system. What is more, the formal definition of the extended ADORA on syntax, static semantics (well-formedness rules) and the refinement calculus provides a sound base to develop a tool, which implements the above mentioned rules and checks the well-formedness of the ADORA model dynamically. A prototype ADORA tool is now being developed in our research group.

References

[1] Back, R-J., Petre, L., Porres, I.:Formalising UML Use Cases in the Refinement Calculus. Turku Centre for Computer Science, Technical Report No 279 (1999)

[2] Fradet, P., Metayer, D., Perin, M.: Consistency checking for multiple view software architectures. Proc. ESEC/FSE99, LNCS series, Springer (1999)

[3] Glinz, M.: Statecharts for Requirements Specification - As Simple As Possible, As Rich As Needed. Proc. of the ICSE2002 Workshop on Scenarios and State Machines: Models, Algorithms and Tools (2002)

[4] Glinz, M., Berner, S., Joos, S.: Object-oriented Modeling with ADORA. In: Information Systems, 27, 6 (2002)

[5] Lehman, M.: Software's Future: Managing Evolution. In: IEEE Software, 15, 1 (1998)

[6] OMG: Unified Modeling Language Specification (Version 1.5). OMG document (2003)

[7] Paige, R., Ostroff, J.: The Single Model Principle. In: Jounal of Object Technology, Vol. 1, No. 5 (2002)

[8] Xia, Y., George, C.: An Operational Semantics for Timed RAISE. In: Proc. of the World Congress on Formal Methods, LNCS Vol. 1709. Springer (1999)

[9] Xia, Y., Glinz, M.: Rigorous EBNF-based Definition for a Graphic Modeling Language. In: Proc. of 10th Asia-Pacific Software Engineering Conference, IEEE Computer Society Press (2003)

[10] Xia, Y.: A Language Definition Method for Visual Specification Languages, Ph.D. thesis, Institut für Informatik, University of Zurich (2004) www.ifi.unizh.ch/req/ftp/phdthesis/xia.pdf