Towards Instant Automatic Model Refinement Based on OCL

Hui Song, Yanchun Sun*, Li Zhou, Gang Huang
Institute of Software, School of Electronics Engineering and Computer Science, Peking University, China
Key laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
{songhui06, sunyc, zhouli04, huanggang}@sei.pku.edu.cn

Abstract

Model refinement is a complex task. It is difficult for developers to refine models all by themselves. A good modeling tool should not only do routine refinement work for developers, but also guide them to make decisions during the refinement process. Such refinement work is best done by modeling tools instantly while developers refine models, to provide timely assistance. In this paper, we present a general approach for meta-model designers or experts in different fields to define and describe such automatic refinement work by rules, and these rules can instruct the modeling tool to do the refinement work instantly, whenever the developers modify the model. The automatic refinement rules in our approach are based on OCL, and their semantics and behaviors are formally defined by using Dijkstra's guarded commands, so the rules are compact, unambiguous and reliable to use. We have also implemented the editor and the interpreter for the automatic refinement rules, and integrated them into our own modeling tool to support our approach.

1. Introduction

Modeling is becoming more and more important for developing software systems. A model is an abstraction of the system [7], and also the result of a series of design decisions [11]. Model refinement is the process for developers to make their decisions and record them in the model, refining the model into a desired description of the system. Model refinement is a complex task, and it's difficult for developers to refine models all by themselves. On the one hand, there is much routine work, and doing it mechanically is bothersome for developers, and furthermore, doing such routine work may require the developers to master relevant technical details. On the other hand, model refinement is a complex work and developers can not consider too many things at one time, so there should be some guidance work to lead a stepwise refinement process. So automatic model refinement is important for developers, which alleviates them from dealing with the trivial knowledge or the process, and makes them concentrate on the creative work.

In existing approaches, automatic model refinement is achieved through model transformation [7, 16, 17]. After developers create models without caring about implementation details, model transformation automatically generates refined models based on the knowledge of a certain domain [17].

Model transformation is an effective way to automate model refinement, but it is not adequate because the designers cannot see the result of their decisions instantly. For example, the developers may have made some wrong decisions that will be revealed in the transformation, but the developers cannot notice that until they have designed the full model and have performed the transformation. Furthermore, the wrong decisions may affect the succeeding decisions, leading to a series of mistakes.

In this paper we argue that model refinement should be *instant,* which has the ability to show the effects of design decisions once developers have made them. It is just like the case of programming, a code editor with instant syntax hinting and quick fix is usually more attractive. Egyed has revealed the advantage of instant consistency checking and analysis for modeling [4, 5], and we believe that *instant* automatic model refinement, which means the automatic actions are performed every time when the model is changed, is also useful for model designers.

There are four main challenges to achieve instant automatic model refinement. (1) The tool provider cannot enumerate all refinement work which can be automatically done, because the work depends on the knowledge from different domains or platforms, or even the developers' personal experience. (2) The definition and description of automatic refinement work should not put a burden on meta-model developers or domain experts, and should better reveal

* Corresponding author

their meanings clearly. (3) The refinement work must be done *efficiently*. (4) As the execution of automatic refinement is triggered *every time* when a model is changed, the effect of this execution will also immediately trigger new automatic refinement work. This may introduce non-deterministic behaviors to the execution of the rule set. The first two issues are also important for model transformation approaches, and a simple and clear way to define transformation rules also attracts much attention, but the last two issues are especially for *instant* model refinement.

In this paper, we present an approach for model designers and experts in different fields to define their own automatic refinement work by rules based on Object Constraint Language (OCL) and a group of refinement primitives. These rules can be used by modeling tools to perform the refinement actions *instantly* as developers refine the model.

Our approach meets the four challenges mentioned above. First, our approach is generic, and anyone can define their own automatic refinement work by writing rules. Second, as a formal language and an OMG standard to write expressions on models, OCL is easy and clear to read and write [14]. The semantics and behaviors of the rules are also formally defined by using Dijkstra's guarded commands [3], so that the rules can be used compactly and clearly, and the meanings are quite intelligible. Third, we use "instant-scope checking" [5] to guarantee the efficiency of rule execution. Fourth, our refinement primitives are well chosen and *formally proved* to be unambiguous and reliable for users to construct their own actions.

We implement a prototype for editing and executing such rules, and integrate the prototype into our own modeling tool (ABCTool [10]). We define a set of rules to assist our modeling approach to compose EJB components based on a software architecture model. We also demonstrate the generalization of our rules by two simple cases from UML standard.

The rest of the paper is organized as follows. We begin with a motivating example in Section 2, followed by the specification of automatic refinement primitives and rules in Section 3. Then we present implementation details and case studies in Section 4 and Section 5. We compare our work with related approaches in Section 6, and conclude in Section 7 with discussions and future work.

2. Motivating Example

In this section, we take our own modeling tool as an example, for using a simple scenario to reveal the importance of automatic model refinement.

ABCTool is a software architecture modeling and component composition tool [10]. Its goal is to use platform independent software architecture as a blueprint to compose reusable components. For example, provided a developer (tool user) wants to construct a simple 'login' system, he may need a 'Login' and a 'User' component communicated using procedure invocation, as Figure 1(a). This simple model cannot be used directly for composition, because it does not contain adequate information. Supposing the designer makes the first decision to use EJB platform, the tool would introduce extended attributes into the model elements (the 1st star mark in the figure), along with the explanations of these attributes, and guide the designer to decide whether a component is for an operation or an entity (b). It's obvious that "login" is an operation, so the designer assigns the value "Session" to its attribute "BeanType", under the tool's conduction. Then the tool introduces two more attributes (the 2nd star mark), to direct the developer to make further decisions. If the user does not care about these aspects, he can leave these attributes empty, and looks for reusable implementations (c). Supposing he finds a session bean named "SignOn", which provides an appropriate interface, he would establish an implementing association between the EJB interface and the component's player. The tool should automatically establish an association between the two components (the 3rd star mark), and check the validity of this association by comparing their attributes. Then, the tool automatically fulfills the previously empty attributes according to the implementation (the 4th star mark).

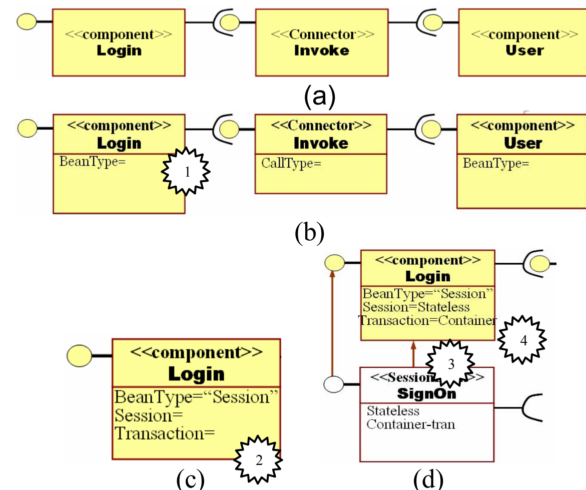

Figure 1: **Composition process**

During this process, the four marked refinement operations should be automatically performed by the tool. These automatic refinements relieve the users of

wasting energy on the routine work and platform details (3 & 4), and more importantly, they guide the users to refine the model step by step (1 & 2). These refinement operations are based on platform knowledge, so if we want to support composition on different platforms, we need a mechanism to introduce the knowledge of the platforms. Furthermore, if an experienced user notices some work which he has to do time after time mechanically, he also needs a way to describe such work, and makes the tool to do it for him during further refinement process.

3. Automatic Refinement Rules

In this section, we discuss how to support automatic refinement rules based on OCL language. We first give a simple formal interpretation of model and model refinement, then we focus on the refinement primitives we provide, and at last, we describe the syntax and semantics of rules, and discuss how to execute them.

3.1 Model and Model Refinement

A model is defined with respect to a meta-model $< D, \widetilde{E}, \widetilde{A}, \widetilde{R} >$, with data, element type, attribute and reference names of element types. The state of an editing model can be defined as $\delta = (E, A, R)$, where E is the set of all model elements defined currently; $A \subseteq E \times \widetilde{A} \rightarrow D$ is a labeled function from model elements to data, and $R \subseteq E \times \widetilde{R} \times E$ is a labeled relation on model elements. The attribute and reference labels, $\widetilde{A}$ and $\widetilde{R}$ are not included in the state identifier, for these two sets are defined by the meta-model, and remain stable during model refining.

The effect of a model refinement action is a transition from one state to another, and the process of refinement is a series of these transitions which lead the model to a desired state. During such a process, many states are incorrect, which may be insufficient or inconsistent. How to transform these states depends on the experience and decisions of the model designers. But among these incorrect states, there are also some trivial ones, which have determinate transitions. That is to say, the model designers need not have experience, nor make decisions to deal with these states, but leave it to the modeling tools. In order to automatic model refinement, a rule must specify the relevant trivial states and the refinement actions.

In the following subsections, we first present the basic refinement actions, which we call refinement primitives, for rule writers to compose their own complex refinement operations; then we describe how to express assertions about model states and how to invoke refinement primitives.

3.2 Refinement Primitives

Refinement primitives are a set of operations defined in the meta-model, each of which performs a basic model modification action.

Many meta-models, like OMG's Meta-Object Facility (MOF) [13], have predefined reflection operations, which can be used to modify the models. But these operations are originally defined as an interface for manual model modification, and not quite appropriate for defining automatic refinement actions. On the one hand, these operations are so elementary that users may need many operations to compose a simple action; on the other hand, these operations are so powerful that they may be inappropriately used by rule writers, and lead the model or even the modeling tool to an unexpected status.

We define and implement three refinement primitives as follows.

- $e.xAttr(attr)$ is used to add extended attribute **attr** to the model element **e.**
- $e.xSet(attr, value)$ is used to set value to a previously empty attribute.
- $e.xAdd(ref, e')$ is used to add new value to a reference set, but the reference would not contain duplicate values after this action.

We limit the ability of "xSet" operation by only allowing assigning value to a previously empty attribute, since the value of an attribute is actually a design decision made previously by the model designer, and if some refinement rule changes the attribute automatically, this decision will be vaporized. We also limit the ability of "xAdd" operations and do not provide a way to create new model elements, for the following reasons: 1) For our current work, these primitives do provide enough power, and other refinement actions should be performed with human participation; 2) These limitations guarantee that the refinement rules composed are unambiguous and reliable, as proved in the following subsection.

3.2.1 Formal Semantics of Refinement Primitives. As stated above, a refinement action is a transition from one state to another, so let Σ be the set of all potential states, and the semantics of a refinement primitive can be formally defined as a map on Σ

$\text{Act}[\![prim]\!] : \Sigma \rightarrow \Sigma$.

The semantics of the three refinement primitives are defined in Figure 2.

$$\mathrm{Act}[\![e.xAttr(attr)]\!](\delta=(E,A,R))=$$
$$\begin{cases}(E, A\cup(e,attr,\perp),R), & \forall d\in D_\perp.(e,attr,d)\notin A\\ \delta, & otherwise\end{cases}$$
$$\mathrm{Act}[\![e.xSet(attr,value)]\!](\delta=(E,A,R))=$$
$$\begin{cases}(\mathrm{Act}[\![e.xSet(attr,value)]\!]\circ\mathrm{Act}[\![e.xAttr(attr)]\!])(\delta) & \forall d\in D_\perp.(e,attr,d)\notin A\\ (E,(A-\{(e,attr,\perp)\})\cup\{(e,attr,value)\},R) & (e,attr,\perp)\in A\\ \delta, & otherwise\end{cases}$$
$$\mathrm{Act}[\![e.xAdd(ref,e')]\!](\delta=(E,A,R))=$$
$$\begin{cases}(E,A,R\cup\{(e,ref,e')\}), & (e,ref,e')\notin\delta(A)\\ \delta, & otherwise\end{cases}$$

Figure 2. **Semantics of refinement primitives**

Here, we lift the set of data from D to $D_\perp = D\cup\{\perp\}$, and the attribute set in the state triple from $A = E\times\widetilde{A}\to D$ to $A_\perp = E\times\widetilde{A}\to D_\perp$. So $a=(e,attr,\perp)\in A_\perp$ here means that there is an attribute label **attr** for model element e, but no value has been appointed.

3.2.2 Correctness Proof of Refinement Primitives. The refinement primitives are automatically invoked by the tool, and the execution of them may change the model to another state, which may be recognized by the tool as another unnecessary state. So once the tool invokes one primitive, there will be a sequence of primitives followed, and this introduces non-deterministic behaviors. As tool providers can't make assumptions about how the rule writers use these primitives, we have to make sure that any execution trace of primitives satisfies the following properties:

- **Unambiguousness.** It means that the final result will remain the same after exchanging any two primitives in the trace.
- **Recursiveness.** It means that any execution trace will finally reach a stable state which does not automatically trigger any other execution.

In the rest of this section, we give a partition of the state set Σ, and define a partial order relation on each of the subsets. Then we use them to prove that the refinement primitives satisfy the two properties above.

DEFINITION 1. The equivalence class of a state set δ, denoted as $[\delta]$, is a subset of Σ and $\delta'\in[\delta]\Leftrightarrow E(\delta)=E(\delta')$, i.e., a set of states with same model elements. It's obvious that these primitives all satisfy such a property: $\mathrm{Act}[\![prim]\!](\delta)\in[\delta]$.

DEFINITION 2. $\leq$ is a relation on a set $[\delta]$, and $\delta_1\leq\delta_2\Leftrightarrow A(\delta_1)\preceq A(\delta_2)\vee(A(\delta_1)=A(\delta_2)\wedge R(\delta_1)\subseteq R(\delta_2))$

.The denotation $\preceq$ here means the elements in the left hand have less attributes or attribute values than those in the right hand, that is,

$$A_1\preceq A_2\Leftrightarrow A_1\subseteq A_2\vee\exists e,a,d.(A_1-\{(e,a,\perp)\}=A_2-\{(e,a,d)\})$$

It is obvious that the relation $\leq$ is a partial order, which expresses the increment of model information. It is also obvious that for all primitives, $\delta\leq\mathrm{Act}[\![prim]\!](\delta)$.

THEOREM 1. Any execution trace of primitives is unambiguous.

PROOF: Any primitive satisfies the expression $\delta\leq\mathrm{Act}[\![prim]\!](\delta)$, so changing the order of primitives in an execution trace will not change the increment trend. We can prove that the primitives are abelian, i.e. $\mathrm{Act}[\![prim_1]\!]\circ\mathrm{Act}[\![prim_2]\!]=\mathrm{Act}[\![prim_2]\!]\circ\mathrm{Act}[\![prim_1]\!]$ So we can use induction to prove that any execution trace has its determined results.

THEOREM 2. Any execution trace of primitives is recursive.

PROOF*:* For an execution trace $p_1,\ldots,p_n$, a state sequence $\delta_0,\delta_1,\ldots,\delta_n$ presents the result states of each primitive execution, and it's obvious that $\delta_0\leq\delta_1\leq\ldots\leq\delta_n$ (as defined in Definition 2). As a sequence $\delta_0=\delta_1=\ldots=\delta_m$ is insignificant, we just need to prove that the sub sequence $\delta_0<\delta_1<\ldots<\delta_m$ of the previous state sequence is finite. According to Definition 1, all these result states are in the same equivalence class, that is, $\delta_0,\delta_1,\ldots,\delta_m\in[\delta_0]$. There is a "max" state $\bar{\delta}=(E,E\times\tilde{A}\to D,E\times\tilde{R}\times E)$ exists in such a set and $\forall\delta\in[\delta].\ \delta\leq\bar{\delta}$. As $\tilde{A}$, $\tilde{R}$ and E are all remained stable, $m<=|E|*|\tilde{A}|*|E|*|\tilde{R}|*|E|$, which is sure finite.

3.3 Automatic Refinement Rules on OCL

The next issue is how to describe the unnecessary states and how to collect enough information to invoke the refinement primitives. As a constraint and query language for models, OCL is appropriate for this job.

3.3.1 OCL overview. Object Constraint Language is a formal language used to describe expressions on MOF models and meta-models, including UML [14, 15]. Rooted in syntropy method, the original goal of OCL is to provide a set of modeling techniques that would allow precise specification. Now OCL is being widely used in different fields. Used as an "Object Query Language", it has been demonstrated that OCL has the expressive power of Relational Algebra [1].

In the automatic refinement rules, we do not only use OCL as a constraint language for rule writers to make assertions on the unnecessary states, but also use

it as a query language to collect information for invoking primitives.

3.3.2 The Syntax of Refinement Rules. Figure 3 describes the syntax of refinement rule, where *bexpr, dexpr,* or *eexpr* means an OCL expression returns a Boolean value, a data value, or a model element. *const* means a constant value, which may be a description or identifier for attribute or reference. We only provide "sequence" control statement, because as a kind of state-action rules, users can easily compose switch or loop controls.

$$
\begin{array}{lcl}
rule & ::= & bexpr \rightarrow action \\
action & ::= & prim \mid action \,;\, prim \\
prim & ::= & eexpr.xAttr(const) \\
 & \mid & eexpr.xSet(const, dexpr) \\
 & \mid & eexpr.xAdd(const, eexpr)
\end{array}
$$

Figure 3**. Syntax of refinement rule**

.

$$
\begin{array}{l}
Exe[\![rule]\!] : \Sigma \rightarrow \Sigma \\
Exe[\![bexpr \rightarrow action]\!](\delta) = \\
\quad \begin{cases} Act[\![action]\!](\delta), & Query[\![bexpr]\!](\delta) = \text{true} \\ \delta, & Query[\![bexpr]\!](\delta) = \text{false} \end{cases} \\
Act[\![action]\!] : \Sigma \rightarrow \Sigma \\
Act[\![action; prim]\!] = \\
\quad Act[\![prim]\!] \circ Act[\![action]\!]
\end{array}
$$

Figure 4. **Semantics of refinement rules**

3.3.3 The Semantics of Refinement Rules. Figure 4 describes the effect of an execution of one single rule. We borrow the notation of Dijkstra's guarded commands [3], if the guard expression returns true, then the tool executes the action, otherwise, it does nothing. Let Rule be the set of all available refinement rules which a tool employs, the behavior of the tool can be described as follows.

$$\mathbf{do}\ rule_1 \parallel rule_2 \parallel \ldots \parallel rule_n\ \mathbf{od} \quad (rule_1, \ldots, rule_n \in Rule)$$

It means that any rule in the set **Rule** can be executed at any moment, with the state being changed or not. After one time of execution, the control flow goes back to the "**do**" keyword, and any rule may be executed again. So, any time the model reaches a trivial state, there will be an execution trace of refinement primitives automatically formed, which will lead the model to a stable state. As proved in the last subsection, any such trace is unambiguous and recursive, so, although we introduce non-deterministic choice, the rule writers can still be confident that any trivial state has a determined destination.

3.3.4 Rule Execution. According to the behavior description above, the tool must keep on executing rules ceaselessly, which is not acceptable for a real modeling tool. We improve the efficiency of rule execution based on two assumptions: (1) Only modifications (manual or automatic) can cause the model to enter an unstable (unnecessary) state, so the execution of rules can be triggered by the change notification. (2) A single model modification can't influence the truth values of all state assertions, so we just have to execute some of the rules on a certain change notification.

Triggered by a model editor's modification, an execution trace of refinement primitives can be viewed as an extension of the user's design decision, and if a decision is cancelled, its extension should also be cancelled, by rolling back the following automatic refinements. In our approach, we log all manual model modifications, along with their execution traces. When the user cancels some decision, there will be a set of automatic undo actions according to its execution trace.

4. Implementation

To demonstrate our approach, we implement a prototype, which is an independent component of our software architecture modeling and component composing tool, ABCTool. ABCTool is implemented as a group of plug-ins under the Eclipse Modeling Framework (EMF) [6]. EMF can be regarded as an efficient Java implementation of the essential parts of MOF standard [13], and can be used to generate modeling tools according to the meta-model, which is defined by ECore (the core meta-model in EMF).

4.1 Primitive implementation

We define refinement primitives as operations of the root model element type (that means all other model elements inherit these operations from it) as illustrated in Figure 5. Then we rewrite the generated java method codes according to their behaviors described in 3.2.

MOF does not allow dynamic accession of new attributes to a model element, so we define a class, ExtendedAttribute, and an association between ModelElement and ExtendedAttribute. Along with the assignment of the rule set, a set of ExtendedAttribute are also introduced into the model, and the effect of xAttr is to set new relations between model elements and extended attributes. xGet and xSet operations are used to read and write these extended attributes. Notice that xGet is side-effect free, and can be used in common OCL expressions. As every attribute or

reference has a unique name, we use this string value to indicate the concerned attribute or reference, and this makes the rule easier to write.

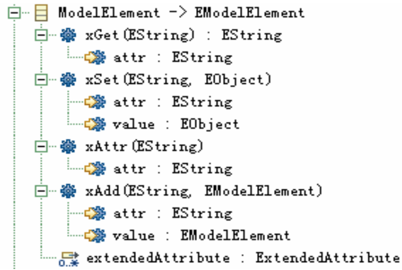

Figure 5: **Root element in ABCTool**

We use the Eclipse OCL project for parsing and evaluating OCL expressions. "Eclipse OCL" is an implementation of OCL OMG standard for EMF-based models. Although the OCL standard places a restriction that an expression must be side-effect free, Eclipse OCL does not make restriction on calling operations, that is to say, the invocation of our refinement primitives can be directly written in an expression, and this makes our refinement rules more compact and intelligible. Since our rules are not purely for specifying models, and we have demonstrated that the operations we provide will not lead the tool to an unstable status, so this "abuse" will not import mistakes to the tools. Figure 6 is a snapshot of rule editors.

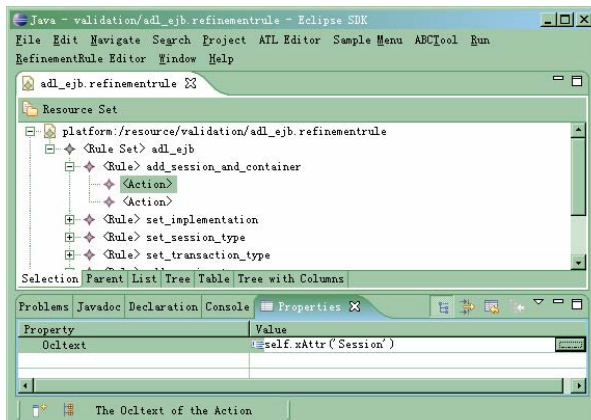

Figure 6: **Rule set example and editor**

There is another question. The *bexpr*, *dexpr* and *eexpr* in refinement rules still require to be side-effect free, otherwise, the checking or querying result will be untrustworthy. Fortunately, all these expressions have to return a value for further use, so do the inside sub-expressions, whose return value will be used by their parents. So, these "void" operations (refinement primitives) may cause exceptions like "*Cannot find operation (=(String)) for the type (null)*". That is to say, the invocation of refinement primitives can only appear at the end of an action, just as the rule syntax in Figure 3.

4.3 Rule execution implementation

We implement our triggering mechanism by using "instance-scope" approach [5]. We consider a tuple <rule, root-element> as a rule instance, and when evaluating a rule instance, the tool associates it with all model elements visited, and if an element is modified, only the associated rule instances will be executed.

The modifications actions are caught by using EMF Notification mechanism. Whenever the model state changes, there will be a notification sent to the registered observer. We implement our rule interpreter as such an observer, so that it can respond the model modifications.

5. Case Study

5.1 Solution to the Motivating Problem

In this subsection, we present how to use automatic refinement rules to support the "stepwise architecture refinement and component composition" modeling process described in Section 2. In Section 2, we have listed four sample automatic refinement actions, and have revealed the requirement for the modeling tool to perform these actions instantly as model changes. In this section, we continue this example and write a set of rules to describe these actions. We list four sample rules according to the four actions marked in Figure 1.

1. The sample rule for adding marking points to guide developers to think about the role of component.

context Component **do**:
  not self.attributes->exists(e|e.name='BeanType')
  =>self.xAttr('BeanType')

2. Guiding developers to decide the component details.

context Component **do**:
  self.xGet('BeanType')='Session'
  =>self.xAttr('Session'); self.xAttr('Transaction')

3. Establishing the implementation association.

context Player **do**:
  not self.parent.impl->includes (self.impl.parent)
  =>self.parent.xAdd('impl', self.impl.parent)

4. Setting attributes according to the implementation:

context Component **do**:
  self.impl.count>0 **and** self.impl.forAll(e|e.isStateless)
  =>self.xSet('Session','Stateless')

We import these rules into ABCTool, and use it to compose this simple EJB application, as shown in Figure 7.

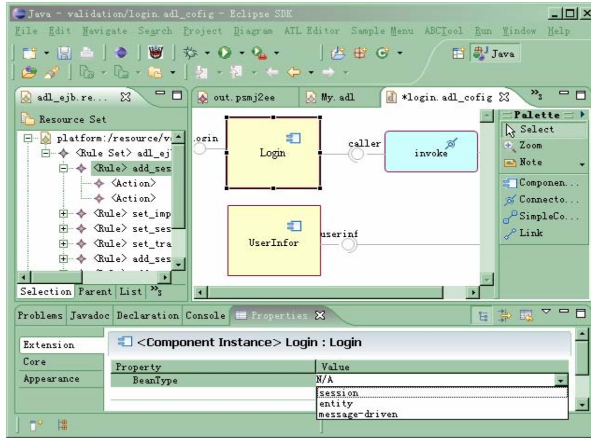

Figure 7: **A snapshot of ABCTool**

5.2 Two Cases in UML

Our approach is not ad hoc to our own modeling tool, nor ad hoc to component composition. It is a general approach for modeling. To demonstrate this, we choose two consistency requirements from UML [15] , and show that maintaining them mechanically is routine work for model designers. We will show how to use our refinement rules to ensure them instantly while a designer is modeling.

Case 1. The constraint "subsets" is widely used in UML specifications, e.g., the association between Class and Operation "ownedOperation" is a subset of "feature". We can use the refinement rule below to achieve the automatic maintenance.

```
context Class do:
    not self.feature->includesAll(self.ownedOperation)
    =>self.xAdd('feature',
          (Self.ownedOperation - self.feature)->first()
       )
```

Case 2. "Provided" is an association in Component definition of UML 2.0, which specifies the interfaces that a component exposes to its environment. Here is the constraint of this association copied from UML specification, which means the provided Interfaces "may be realized by the Component or any of its realizingClassifiers, or they may be the Interfaces that are provided by its public Ports".

```
context Component::provided derive:
  let implInterfaces = self.implementation->collect(impl|impl.contract)
  and let realizedInterfaces = RealizedInterfaces(self)
  and let reaClsInterfaces = RealizedInterfaces(self.realizingClassifier)
  and let typesOfRequiredPorts = self.ownedPort.provided in
    (((implementedInterfaces->union(realizedInterfaces)->
    union(realizingClassifierInterfaces))-> union(typesOfRequiredPorts))
     ->asSet()
def: RealizedInterfaces : (classifier : Classifier) : Interface =
  (classifier.clientDependency->
      select(dependency|dependency.oclIsKindOf(Realization) and
      dependency.supplier.oclIsKindOf(Interface)))
   -> collect(dependency|dependency.client)
```

We can use four rules to specify the automatic refinements to maintain this constraint, corresponding to the four "let" statements. We present one of them below, and the other three are quite the same:

```
context Component do:
  let realizedInterfaces = (self.clientDependency->
    select(dependency|dependency.oclIsKindOf(Realization) and
    dependency.supplier.oclIsKindOf(Interface)))->
    collect(dependency|dependency.client) in
  not self.provided->includesAll(realizedInterfaces)
    =>self.xAdd('provided',
        (realizedInterfaces - self.provided)->first()
      )
```

The rule looks a little complicated, but actually the main part of this rule is directly copied from the constraint.

6. Related Work

There are various approaches to assist and guide model designers in making decisions, and our work is mainly rooted in two kinds of them, i.e. inconsistency analysis and model transformation.

"Inconsistency analysis" means analyzing the violations of consistency rules by locating the involved model elements [4], detecting the error reason [9], or even providing potential repair actions [12]. The model designers use these analysis results as hints to make further refinements. Compared with inconsistency analysis, the rules in our approach do not only contain the hints, but also the instructions to modify the models automatically. Some researchers in this field try to achieve checking and analyzing consistency rules instantly as model changed [4, 5, 18]. We also face this common problem, and our solution is rooted in the "rule-instance scope" approach presented by Egyed [5].

Model transformation is widely used for automatic model refinement [7, 17]. An up-to-date survey and discussion on model driven engineering can be found in the literature [7]. As revealed in the first section, the key point differentiates our approach from model

transformation is that our automatic refinement actions are performed instantly once the models have changed.

Sendall [17] has revealed that among the approaches for defining transformations, a generic transformation language support offers the most flexibility. ATL [8] is one of such languages, and its transformation rules look quite similar to our refinement rules, i.e., using OCL expressions to find a certain pattern in the source model, and modifying the target model according to the defined operations. As a batched automatic refinement work, automatic modification actions during model transformation process would not affect further transformations, and so model transformation languages do not need to deal with the problem of non-deterministic behaviors in rule execution, nor the problem of efficiency, but these are the key issues in our approach.

Research work on architecture refinement is also related to ours. These approaches focus on the preservation of particular properties from the abstract model to the refined one. Such properties may be structural [11] or behavioral [2]. These approaches can be viewed complementary to ours, for they are concerned about the result of refinement, but we are more concerned about the process.

7. Conclusion

Instant automatic model refinement assists and guides model designers in making refinement decisions. This paper presents a way for developers to define such automatic work by using OCL and a group of predefined refinement primitives. Our approach can facilitate cooperation among developers. Domain experts compose refinement operations for the routine work as many as they can, while the developers focus on more creative work that is difficult to automate. In the mean time, our approach is extensible. Tool providers can define extra primitives, and we propose a formal way to check if the newly introduced primitives are also unambiguous and reliable for rule writers.

The implementation of our approach is just a prototype now. Our main future work is to improve our tool and use it in large-scale cases, and extract appropriate refinement primitives for different fields of modeling. We also consider analyzing rules to provide a more flexible and usable solution.

Acknowledgement

This effort is sponsored by the National Basic Research Program (973) of China under Grant No. 2005CB321805; the National High-Tech Research and Development Program (863) of China under Grant No. 2007AA01Z127; the National Natural Science Foundation of China under Grant No. 90412011, 60503028.

References

[1] D. H. Akehurst and B. Bordbar, "On querying UML data models with OCL," *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools,* pp. 91-103, 2001.

[2] L. Baresi, R. Heckel, S. Thöne, and D. Varró, "Style-based modeling and refinement of service-oriented architectures," *Software and Systems Modeling,* vol. 5, pp. 187-207, 2006.

[3] E. W. Dijkstra, *A Discipline of Programming*, Prentice:Prentice-Hall,1976.

[4] A. Egyed, "Fixing Inconsistencies in UML design models," *International Conference on Software Engineering,* 2007.

[5] A. Egyed, "Instant consistency checking for the UML," *International Conference on Software Engineering,* pp. 381-390, 2006.

[6] Eclipse Foundation, "Eclipse Modeling Framework," http://www.eclipse.org/modeling/emf/.

[7] R. France and B. Rumpe, "Model-driven development of complex software: a research roadmap," *International Conference on Software Engineering 2007 Future of Software Engineering,* pp. 37-54, 2007.

[8] ATLAS Group, "ATLAS Transformation Language (ATL)," http://www.eclipse.org/m2m/atl/.

[9] A. Hunter and B. Nuseibeh, "Managing inconsistent specifications: reasoning, analysis, and action," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 7, pp. 335-367, 1998.

[10] H. Mei, G. Huang, H. Zhao, and W. Jiao, "A software architecture centric engineering approach for Internetware," *Science in China Series F: Information Sciences,* vol. 49, pp. 702-730, 2006.

[11] M. Moriconi, X. Qian, and R. A. Riemenschneider, "Correct architecture refinement," *Software Engineering, IEEE Transactions on,* vol. 21, pp. 356-372, 1995.

[12] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," *Proceedings of the 25th International Conference on Software Engineering,* 2003.

[13] OMG, "Meta-Object Facility," in http://www.omg.org.

[14] OMG, "The OCL Specification," in http://www.omg.org

[15] OMG, "UML Specification 2.0 Supertructure," http://www.omg.org.

[16] B. Selic, "The pragmatics of model-driven development," *Software, IEEE,* vol. 20, pp. 19-25, 2003.

[17] S. Sendall and W. Kozaczynski, "Model transformation: the heart and soul of model-driven software development," *Software, IEEE,* vol. 20, pp. 42-45, 2003.

[18] C. Xu, S. C. Cheung, and W. K. Chan, "Incremental consistency checking for pervasive context," *International Conference on Software Engineering,* pp. 292-301, 2006.