An Axiomatic Formalization of UML Models

Toshiaki Aoki Takaaki Tateishi Takuya Katayama

School of Information Science
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi, Noumi, Ishikawa, JAPAN
{toshiaki, ttate, katayama}@jaist.ac.jp

Abstract: Though it is widely recognized that object-oriented methodologies are most effective in developing large scale software, it is still difficult to apply high level computer support to it, as current methodologies are informal, especially at their upstream phases. This means that we cannot expect sophisticated supports for checking consistency among analysis models and validating/verifying their appropriateness. As quality of analysis models determines that of succeeding design and implementation phases, formalization of the analysis phase is very important. In this paper, we propose a formal approach to analysis phase, consisting of (1)formal analysis models, (2)unification of the models, (3)an axiom system for consistency verification and (4)verification environment.

1 Introduction

Though the object-oriented methodologies are most effective in developing complex software systems, the problem of current methodologies is that they are informal, especially at their upstream phases. That is, their analysis phase for building analysis models from the system requirements is very informal and this makes it difficult to apply computer support in this phase. As is manifested in UML models[UML99], we usually construct multiple analysis models where inconsistencies remain even though software designers pay much attention in constructing them. This will lead to the introduction of grave bugs in the implemented software system.

One approach to solve this problem is to adopt formal techniques in the analysis phase. This may include formalization of analysis models, their prototype execution and verification techniques for showing that they are consistent and satisfy the requirements.

In this paper, we show our formal approach for object-oriented analysis modeling. It consists of (1) formal analysis models, (2) unification of the analysis models into a single executable model using the concept of unification mapping, (3) consistency verification of the models and (4) an axiomatic system for the consistency verification. After introducing how the analysis models are formalized and how they are unified into the unified model, we show consistency verification between static model described by a class diagram and the dynamic model described by state diagrams. That is, we show how constraints or assertions attached in the class diagram are proved to hold. The method used in our approach is a typical invariant assertion method. The similar method is adopted in the current methodologies such as Catalysis[DW98]. To describe such constraints, a language called OCL[WK98] is proposed. Though they are to describe constraints in a target system, they do not provide methods for checking whether the constraints are satisfied in the constructed

model or not. Our approach provides one of methods to prove that such constraints are satisfied. In our approach, the constraints are proved in an axiomatic system. This axiomatic system defines rigorous semantics of UML models and allows us to adopt theorem proving systems for supporting the verification.

We have used a theorem proving system HOL[HOL91] for higher order logic for verifying the consistency. We found that HOL is extremely useful by its higher order modeling power. The biggest merit of the higher order logic is that we can formalize properties of OO models at as abstract level as we want and make them instantiate to a concrete level when we use them. This matches with the well-known software design principle that we should not bring in unnecessary details until they are required and this, of course, applies to the analysis phase in OO methodologies.

This paper is organized as follows. In the next section, we show formal analysis models and their unification. In Section 3, we propose a consistency verification and an axiomatic system for the verification. In Section 4, we present the implemented verification environment and discuss about usefulness of the higher order modeling power that HOL has. In Section 5, we discuss related works. Section 6 gives conclusions and the directions of our future work.

2 Formalizing OO Analysis Models

In UML, there are multiple models to describe a single system. Usually, they are made independently and the relation among them tends to be ambiguous. As these models are merged in the succeeding design and implementation, this ambiguity may bring in a big bug in the implemented programs. To solve this problem, we need to formalize the analysis phase. In our approach, the analysis phase consists of two phases, (1) building independent basic models and (2) unifying them into a unified model. The unified model should be described formally enough to be verified. In Section 2, we introduce a *basic class model* and a *basic statechart model* which are defined independently in the sense that elements in one model do not appear in another model. These are based on UML's class diagram and statechart diagram respectively. Then we unify these models using unification mappings. These two diagrams are very important to capture the whole behavior of the target system and are used in almost all system developments.

2.1 Basic Models

Each basic model has the sets of identifiers which are independent of the sets of the other models.We call such sets *basic sets*. The identifiers are atoms that identify abstractions appearing in the associated basic model. These identifiers represent particular concepts in the system, so they have meanings which are defined by documents related to them. The documents may be described by some formal languages or by natural languages. It is not essential here whether they are described formally or not. We are interested only in the relations which hold among the sets and in describing them formally.

The basic class model is defined using basic sets $AttrID$, $FuncID$, $ClassID$, $AssocID$, $AggrID$ and $InherID$. These basic sets represent a set of attribute identifiers, a set of function identifiers, a set of class identifiers, a set of association identifiers, a set of aggregation identifiers and inheritance identifiers respectively. The basic class model is built from the above identifiers. For example, the set of attributes $\mathbf{A}$ appearing in

the basic class model of a target system is a subset of the basic set $AttrID$. Formally, a basic class model CM is defined as follows.

Definition 2.1 *Basic Class Model*

$$CM = (\mathbf{A}, \mathbf{F}, \mathbf{Class}, \mathbf{Assoc}, \mathbf{Aggr}, \mathbf{Inher}, \mathcal{M}_{CM})$$
where $\mathbf{Class} \subseteq ClassID, \mathbf{Assoc} \subseteq AssocID, \mathbf{Aggr} \subseteq AggrID,$
$\mathbf{Inher} \subseteq InherID, \mathbf{A} \subseteq AttrID, \mathbf{F} \subseteq FuncID$

In the above, $\mathcal{M}_{CM}$ is a function representing relations between identifiers. $\mathcal{M}_{CM}$ is a direct sum of functions $\mathcal{M}_{class}$, $\mathcal{M}_{assoc}$, $\mathcal{M}_{aggr}$ and $\mathcal{M}_{inher}$, that is, formally defined as follows.

$$\mathcal{M}_{CM} = \mathcal{M}_{class} \oplus \mathcal{M}_{assoc} \oplus \mathcal{M}_{aggr} \oplus \mathcal{M}_{inher}$$

- *The function $\mathcal{M}_{class}$ represents the fact that a certain class has a specific set of attributes and functions. That is, this function has the following domain and range.*

  $$\mathcal{M}_{class} : \mathbf{Class} \rightharpoonup Pow(\mathbf{A}) \times Pow(\mathbf{F})$$

  where $Pow(X)$ means a power set of X.
- *The function $\mathcal{M}_{assoc}$ relates each association identifier to a relation between classes with multiplicities. This function has the following domain and range.*

  $$\mathcal{M}_{assoc} : AssocID \rightharpoonup (ClassID \times M) \times (ClassID \times M)$$

  where M is a set of the multiplicities. For a set of natural numbers N, M is defined as follows.

  $$M = \{(m, n) | m \in N, n \in N \cup \{\infty\}, m \leq n\}$$

  In this definition, ∞ represents an infinite number, that is, the property $\forall n \in N.n \leq \infty$ holds.
- *The function $\mathcal{M}_{aggr}$ relates each aggregation identifier to a relation between assembly class and part classes. This function has the following domain and range.*

  $$\mathcal{M}_{aggr} : AggrID \rightharpoonup ClassID \times Pow(ClassID \times M)$$

- *The function $\mathcal{M}_{inher}$ relates each inheritance identifier to a relation between a super-class and sub-classes. This function has the following domain and range.*

  $$\mathcal{M}_{inher} : InherID \rightharpoonup ClassID \times Pow(ClassID)$$

Similarly a basic statechart model SM is defined using its own basic sets $STDID$, $EventID$, $ActionID$, $CondID$ and $StateID$. They represent a set of state transition model identifiers, a set of event identifiers, a set of action identifiers, a set of condition identifiers and a set of state identifiers. The statechart model consists of state transition models. A basic statechart model SM is formally defined as follows.

Definition 2.2 *Basic Statechart Model*

$$SM = (STD, \mathcal{M}_{SM})$$
where $STD \subseteq STDID, \mathcal{M}_{SM} : STDID \rightharpoonup ST$

In the above definition, ST is a set of state transition models. Each state transition model is defined as follows.

Definition 2.3 *State Transition Model*
A state transition model st is defined as follows.

$st = (\mathbf{S}, \mathbf{Evt}, \mathbf{Act}, \mathbf{Cond}, \mathbf{Trans}, s)$
where $\mathbf{S} \subseteq StateID, \mathbf{Evt} \subseteq EventID, \mathbf{Act} \subseteq ActionID, \mathbf{Cond} \subseteq CondID,$
$\mathbf{Trans} \subseteq \mathbf{S} \times \mathbf{Evt} \times \mathbf{Cond} \times \mathbf{Act} \times Pow(\mathbf{Evt}) \times \mathbf{S}, s \in \mathbf{S}$

where **S**, **Evt**, **Act**, **Cond** *and* **Trans** *represent sets of states, events, actions, conditions and state transitions respectively. The state identifier s is an initial state of this state transition model.*

In this definition, each state transition is represented by a formula $(s, e, c, a, \{e_1, \cdots, e_n\}, s')$, where s and s' represent the source state and the destination state of this transition respectively. This formula represents the fact that the transition fires, the action a is performed and the events $e_1, \cdots$ and e_n are sent to the other objects if the event e is received and the condition c holds in the state s.

The other models used in object-oriented analysis are defined similarly.

2.2 Unifying Basic Models

In building the basic models, we independently model each view of the target system and describe it as a basic model. The details of elements appearing in the basic models are determined in the progress of the analysis phase. These elements have then enough information to relate them with the elements appearing in the other basic models. To relate such elements, we use *unification mappings*. In determining the unification mappings, we may fail to relate an element with others. Such failures occur when for the element we cannot extract related elements in other basic models. In other words, these models are inconsistent syntactically. In this case, we must remove the element from the basic model or add some elements in the other basic models.

Unification mappings map elements in the basic models to *common components*. Common components are expressions defined from elements in the basic models.

Consider an example shown in the Figure 1. In this example, we assume that the action identifier $Initialize$ appears in the basic statechart model. This identifier represents an action which initializes the target system. On the other hand, the attribute $number$ appears in the basic class model and its initial value is 0. Using this information we find and define an expression $Counter.number := 0$ which means to assign the value 0 to the attribute $number$ in the class $Counter$. This expression is a member of common components. We relate it with the action identifier $Initialize$ using a mapping $Initialize \mapsto Counter.number := 0$. This mapping unifies the attribute $number$ appearing in the basic class model and the action identifier $Initialize$ in the basic statechart model. This is a fragment of a unification mapping $\mathcal{U}_{action}$ for action identifiers.

Definition 2.4 *Unification mapping for action identifiers.*

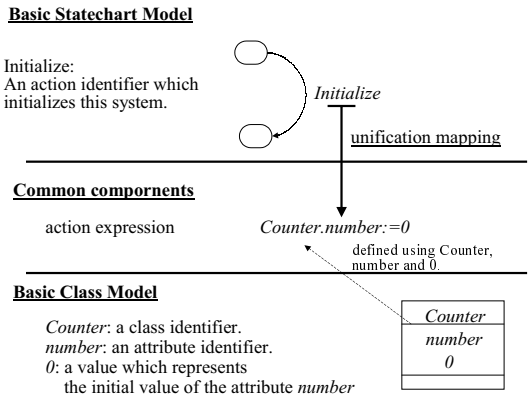

Figure 1: An example of unification mapping

Unification mapping for action identifiers is represented by the function $\mathcal{U}_{action}$ whose domain and range are defined as follows.

$$\mathcal{U}_{action} : ActionID \rightarrow AExp$$

where $AExp$ is a set of action expressions and defined as follows.

$$\begin{aligned} AExp &= T \mid c.a := T \mid AExp; AExp \\ T &= c.a | c.f(Tlist) \\ Tlist &= T, Tlist \mid \epsilon \end{aligned}$$

In the above, c is a class identifier or an event identifier, a is an attribute identifier and f is a function identifier. The term $c.a$ represents an attribute a in a class c.

This $\mathcal{U}_{action}$ is a unification mapping connecting the basic class diagram and the basic statechart diagram. We have identified the five unification mappings $\mathcal{U}_{ST}$, $\mathcal{U}_{del}$, $\mathcal{U}_{cond}$, $\mathcal{U}_{event\text{-}attr}$ and $\mathcal{U}_{event\text{-}dst}$ in addition to $\mathcal{U}_{action}$.

The unification mapping represented by the function $\mathcal{U}_{ST}$ defines behavior of objects instantiated from a class by assigning a state transition model to that class.

Definition 2.5 *Unification mapping for the behavior of objects*
$\mathcal{U}_{ST}$ is a function whose domain and range are defined as follows.

$$\mathcal{U}_{ST} : ClassID \rightarrow STDID$$

The unification mapping represented by the function $\mathcal{U}_{del}$ defines the details of delegation in the case where there is an aggregation relation in the basic class model. This unification mapping relates a state of the assembly class in an aggregation relation to a set of the part classes. This means that if objects of the assembly class are in the state related to a set of the part classes, these objects pass all the received events to objects of the classes in this set.

Definition 2.6 *Unification mapping for the delegation*
$\mathcal{U}_{del}$ *is a function whose domain and range are defined as follows.*

$$\mathcal{U}_{del} : StateID \rightarrow Pow(ClassID)$$

The unification mapping for the condition identifiers is represented by the function $\mathcal{U}_{cond}$.

Definition 2.7 *Unification mapping for condition identifiers*

$$\mathcal{U}_{cond} : CondID \rightarrow BExp$$

In the above, $BExp$ *is defined as follows.*

$$BExp \ = \ T \mid T \wedge T \mid T \vee T \mid \neg T$$

where T *is defined in the definition of* $AExp$.

The events in the basic statechart model may have attributes. The unification mapping represented by the function $\mathcal{U}_{event\text{-}attr}$ assigns such attributes to events.

Definition 2.8 *Unification mapping for event attributes*
$\mathcal{U}_{event\text{-}attr}$ *is a function whose domain and range are defined as follows.*

$$\mathcal{U}_{event\text{-}attr} : EventID \rightarrow Pow(AttrID)$$

In our model, the events output from a state transition model are transmitted by links instantiated from associations in the basic class model. The unification mapping represented by the function $\mathcal{U}_{event\text{-}dst}$ defines destinations of output events by assigning them to the associations.

Definition 2.9 *Unification mapping for event destination*
$\mathcal{U}_{event\text{-}dst}$ *is a function whose domain and range are defined as follows.*

$$\mathcal{U}_{event\text{-}dst} : EventID \rightarrow Pow(AssocID)$$

Having defined the unification mappings, a unified model UM is defined as a pair of the basic models $BasicModels$ and the unification mappings $\mathcal{U}$. The unified model defines behavior of the target system within the context of the unification mappings $\mathcal{U}$, which is enough to specify its state transition semantics.

Definition 2.10 *Definition of Unified Model* UM

$$UM = (BasicModels, \mathcal{U})$$

In this definition, BasicModels represents a set of basic models, that is, CM, SM $\in$ *BasicModels.* $\mathcal{U}$ *is a direct sum of unification mappings, that is,* $\mathcal{U} = \mathcal{U}_{action} \oplus \mathcal{U}_{ST} \oplus \mathcal{U}_{del} \oplus \mathcal{U}_{cond} \oplus \mathcal{U}_{event\text{-}attr} \oplus \mathcal{U}_{event\text{-}dst}$.

3 Consistency Verification

In large-scale system development, it is impossible to extract all the elements in the system at once when constructing its analysis model. Complete models are obtained by constructing incomplete models and refining them repeatedly. In this style of development, we need a guide to tell us whether the extracted elements are complete or not. The basic models and unification mappings provide one of the solutions for this problem because elements to be extracted are defined by identifiers and completeness of the model is ensured by defining formal relations among these identifiers, that is, unification mappings. Such completeness ensures the syntactical consistency among the basic models in the sense that extracted elements are neither too many nor too few among the basic models. However, this does not ensure the basic models are semantically consistent. There are many semantical consistencies among the basic models and we have, so far, proposed a verification method which ensures one of such consistencies, a consistency between the basic statechart model and dataflow diagram[AK98]. In this section, we propose a verification method to check consistency between the basic class model and the basic statechart model.

3.1 Consistency

To specify constraints of objects, we usually assign an assertion to a class which instantiates these objects. Such assertion represents a property which always holds in these objects. The purpose of the verification method we propose in this section is to show that the assertion assigned to a class holds under any behavior of objects instantiated from this class.

In the proposed verification method, we call the assertion assigned to a class in the basic class model a *global assertion* for objects instantiated from that class. Such a global assertion involves attributes of the objects. The objects change the values of the attributes by executing actions defined in the basic statechart model. Though these actions appearing in the basic statechart model are independent of attributes appearing in the basic class model, we can verify invariance of the global assertion with unification mappings. This is because actions in the statechart model can be described as attribute evaluation expressions using unification mappings.

If we can prove that all assertions assigned to classes always hold, we say that the basic class model and the basic statechart model are consistent.

3.2 Preparation

In object-oriented approach, a target system consists of objects. Such objects can be obtained by instantiating from classes specified in the basic class model. They behave individually by collaborating with others using events. An event is sent and received between two objects through *links* which connect them. Here we introduce a set of object identifiers $ObjectID$ and define a system consisting of objects.

Definition 3.1 *Object System*

An object system $System$ *is defined as follows.*

$$System = (O, Link, Del)$$
$$\textit{where } Link \subseteq O \times O, Del \subseteq O \times O, O \subseteq ObjectID$$

In this definition, $Link$ represents a set of links and Del represents a set of delegation relations between objects in a target object system.

We introduce an operator called *instantiating operator* to express that objects are instantiated from classes.

Definition 3.2 *Instantiating Operator*

A set of objects which are instantiated from a class c is represented by $\mathcal{O}(c)$. This operator $\mathcal{O}$ has the following domain and range.

$$\mathcal{O} : ClassID \rightarrow Pow(ObjectID)$$

Each object has the values of its attributes. To express the current values of attributes, we introduce *attribute environments*.

Definition 3.3 *Attribute Environments*

Values of an attribute that an object o has in a state s are represented by an attribute environment $\sigma[o][s]$. The attribute environment $\sigma[o][s]$ is a function whose domain and range are defined as follows.

$$\sigma[o][s] : ClassID \times AttrID \rightarrow Value$$

where $Value$ is a set of values which all attributes in the set $AttrID$ may hold.

In this definition, an attribute owned by a class is represented by a pair of an attribute identifier and class identifier so that we can uniquely identify that attribute in the class model. We will use the notation $c.a$ instead of the pair (c, a) for the simplicity of descriptions.

Definition 3.4 *Event Attribute Environments*

If an object o receive an event e in a state s, values of an attribute that the event e has are represented by an event attribute environment $\tau[o][s]$. If an object o send an event e in a state s, the values of an attribute that the event e has are represented by an event attribute environment $\upsilon[o][s]$. The event attribute environment $\tau[o][s]$ and $\upsilon[o][s]$ are functions whose domain and range are defined as follows.

$$EventID \times AttrID \rightarrow Value$$

3.3 Overview of Verification

To prove that a global assertion holds for any behavior of an object to which the global assertion is assigned, it is sufficient to show that the global assertion holds at any state of the object. In the following, we show how to prove it using Figure 2 as a simple example. Note that, in this figure, the unification mappings have been applied to the basic models and identifiers in them are fully related.

In this figure, a class $Counter$ appears in the basic class model and it has an attribute $number$. The behavior of objects instantiated from this class is defined by a state transition model in the basic statechart model also shown in Figure 2. In this state transition model, the initial value of the attribute $number$ is 0. If the object receives the event e_+ in the

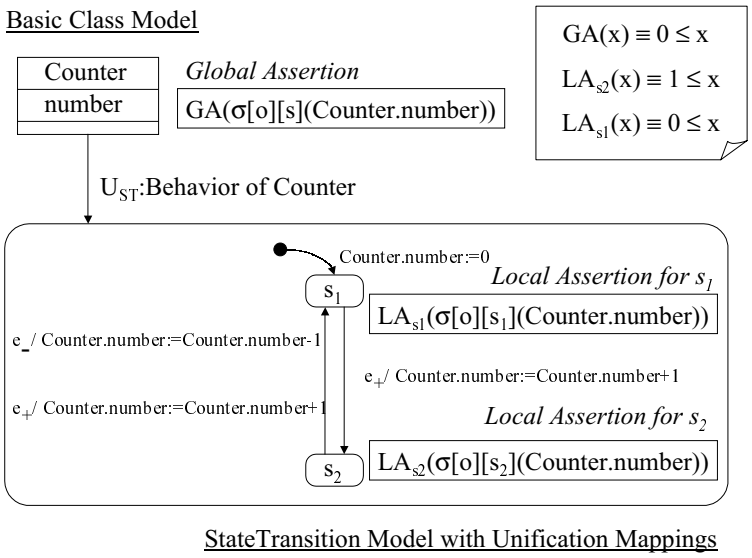

Figure 2: A Simple Example

state s_1, the attribute $number$ is incremented and the current state of the object becomes the state s_2. Then, if the object receives the event e_- in the state s_2, the attribute $number$ is decremented and the current state becomes the state s_1. The object can also receive the event e_+ in the state s_2. In this case, the attribute $number$ is incremented and the current state becomes the state s_1.

As the value of the attribute $number$ of any object instantiated from the class $Counter$ must be always positive, we assign the global assertion $GA(\sigma[o][s](Counter.number))$ to that class in the basic class model for any object o instantiated from the class $Counter$. In this formula, GA is deÞned as $GA(x) \equiv 0 \leq x$, $Counter.number$ represents the pair $(Counter, number)$ and $\sigma[o][s](Counter.number)$ represents the value of the attribute $number$ of the object o in the state s. We expect that the attribute $number$ is always positive because the event e_- which decrements the value of the attribute $number$ is received only after receiving the event e_+ which increments its value.

The global assertion always holds, that is $\forall s.GA(\sigma[o][s](Counter.number))$ is proved in the following order.

1. We focus on an instantiated object and assign a *local assertion* to each state of this object. We call an assertion that is expected to hold in a state of the basic statechart model a local assertion for that state. In the example, we instantiate the object o from the class $Counter$ and assign the local assertion $LA_{s_1}(\sigma[o][s_1](Counter.number))$ to the state s_1 and $LA_{s_2}(\sigma[o][s_2](Counter.number))$ to the state s_2 to the state s_1 and s_2 respectively. In these local assertions, LA_{s1} is deÞned by $LA_{s_1}(x) \equiv 0 \leq x$ and LA_{s_2} is deÞned by $LA_{s_2}(x) \equiv 1 \leq x$.

2. Prove that any local assertion given in the previous step always hold for any state to which it is assigned. We can prove this fact using an induction on the structure of the state transition model. If local assertions can be proved by this induction, we say that these local assertions are *locally valid* and call them *local invariants*. In the example, we prove that the following four propositions hold.

   (base case) $LA_{s_1}(0)$
   (induction step 1.)

   $$LA_{s_1}(\sigma[o][s_1](Counter.number)) \Rightarrow LA_{s_2}(\sigma[o][s_1](Counter.number)+1)$$

   (induction step 2.)

   $$LA_{s_2}(\sigma[o][s_2](Counter.number)) \Rightarrow LA_{s_1}(\sigma[o][s_2](Counter.number)-1)$$

   (induction step 3.)

   $$LA_{s_2}(\sigma[o][s_2](Counter.number)) \Rightarrow LA_{s_1}(\sigma[o][s_2](Counter.number)+1)$$

   As we can prove the above four propositions, the local assertions LA_{s_1} and LA_{s_2} are locally valid, that is they are local invariants for the state s_1 and s_2 respectively.

3. To prove that the global assertion always holds for any state, it is sufficient to show that the global assertion is derived from each local invariant. In this example, we prove the following two propositions.
   - $LA_{s_1}(\sigma[o][s_1](Counter.number)) \Rightarrow GA(\sigma[o][s_1](Counter.number))$
   - $LA_{s_2}(\sigma[o][s_2](Counter.number)) \Rightarrow GA(\sigma[o][s_2](Counter.number))$

   As we can prove the above two propositions, the global assertion GA always holds under any behavior of the basic statechart model for an arbitrary object instantiated from the class $Counter$. In other words, the basic class model and the basic statechart model are consistent.

In the next section, we propose an axiomatic system for proving global assertions, which follows the verification procedure mentioned for the simple example.

3.4 An Axiomatic System

In our approach, the constructed models are verified as early as possible in the analysis phase. If we could detect some errors as a result of the verification, we have to modify the analysis model and we verify it again. In such analysis process, there may be many cycles of model verification and modification. In the process, we may have to define an axiomatic system for each of modified analysis models. We should automatically generate the axiomatic system for the constructed analysis model because it is costly to manually define it repeatedly. To do so, it is desirable if we can define the axiomatic system which is free from each model constructed and each local assertion assigned. Using a higher order logic makes it possible to define abstractly the axioms containing predicate variables, which are instantiated to concrete logical expressions when models are constructed. Thus, we can make the axioms independent of each constructed model.

3.4.1 Local Invariant Axiom

As we can see from the verification of the simple example, we derive local invariants of each state from the validity of local assertions. To do so, we first define the validity of local assertions as follows before defining an axiom to derive local invariants.

Definition 3.5 *Validity of local assertions*
Local assertions LA_{s_i}, $(1 \leq i \leq n)$ *of an object* o *assigned to a state* s_i, $(1 \leq i \leq n)$ *respectively are locally valid if the following* $valid[o](LA_{s_1}, \cdots, LA_{s_n})$ *holds.*

$$
\begin{array}{l}
valid[o](LA_{s_1}, \cdots, LA_{s_n}) \Leftrightarrow \\
LA_{s_1}(\overline{v}) \wedge (\bigwedge_{t \in T} LA_{s(t)}(\overline{\sigma}[o][s(t)])) \wedge cond_t(\overline{\sigma}[o][s(t)], \overline{\tau}[o][s(t)]) \\
\quad \Rightarrow LA_{d(t)}(\overline{\sigma}'[o][d(t)])
\end{array}
$$

In this definition, each local assertion has formal parameters which correspond to values of attributes $a_1, \cdots, a_m$ that the object o has. The notation $\overline{v}$ represents a vector of initial values of the attributes, that is, $v_1, \cdots, v_m$, where v_j is the initial value of the attribute a_j for $1 \leq j \leq m$. $\overline{\sigma}[o][s]$ represents a vector of attribute values in the state s of the object o and is formally defined by $\overline{\sigma}[o][s] \equiv \sigma[o][s](a_1), \cdots, \sigma[o][s](a_m)$. T is a set of transitions appearing in a state transition model of the object o. $s(t)$ and $d(t)$ represent a source state and a destination state of the transition t respectively. $\overline{\sigma}'[o][d(t)]$ represents a vector of the values changed by an action assigned to the transition t. If the transition t has an action which changes the attribute value $\sigma[o][s(t)](a_j)$ into $f_j(\overline{\sigma}[o][s(t)], \overline{\tau}[o][s(t)])$, $(1 \leq j \leq m)$, $\overline{\sigma}'[o][d(t)]$ is equivalent to $f_1(\overline{\sigma}[o][s(t)], \overline{\tau}[o][s(t)]), \cdots, f_m(\overline{\sigma}[o][s(t)], \overline{\tau}[o][s(t)])$. $\overline{\tau}[o][s(t)]$ represents a vector of attribute values of an event that the object o receives in the state $s(t)$. The symbol $cond_t$ represents a transition condition which gets the values of the attributes and returns a boolean value.

We introduce an axiom to derive local invariants from the validity of local assertions. Such axiom is defined as *local invariant axiom*.

Axiom 3.1 *Local Invariant Axiom*
The local invariant axiom for a class c *and its state* $s_i, (1 \leq i \leq n)$ *is defined as follows.*

$$\forall o \in \mathcal{O}(c). \forall LA_{s_1}, \cdots, LA_{s_n}. valid[o](LA_{s_1}, \cdots, LA_{s_n}) \Rightarrow \bigwedge_{1 \leq i \leq n} LA_{s_i}(\overline{\sigma}[o][s_i])$$

In this axiom, the variables $LA_{s_1}, \cdots, LA_{s_n}$ representing local assertions are specialized by concrete logical expressions when we derive local invariants. Using variables which hold logical expressions we can make this axiom independent of local assertions for a specific constructed analysis model.

3.4.2 Global Invariant Axiom

To prove that the global assertion holds in any state, it is sufficient to show that it holds in each state. Such sufficient condition is usually derived by weakening each local invariant. If we can prove that the global assertion always holds in any state, we call it a *global invariant*. We introduce an axiom, which derives a global invariant by weakening each local invariant, as *global invariant axiom*.

Axiom 3.2 *Global Invariant Axiom*

The global invariant axiom for a class c and its states $s_i, (1 \leq i \leq n)$ is defined as follows.

$$\forall o \in \mathcal{O}(c).\forall GA. \bigwedge_{1 \leq i \leq n} GA(\overline{\sigma}[o][s_i]) \Rightarrow (\forall s.GA(\overline{\sigma}[o][s]))$$

3.4.3 Event Communication

An object communicates with other objects using events. When a transition fires, the object sends events. To represent such communications, we introduce three axioms called *event introduction axiom*, *event communication axiom* and *event attribute axiom*. The event introduction axiom represents the production of events when a transition fires. The event communication axiom represents that an event sent from an object is received by another object. The event attribute axiom is to derive the fact that a property always hold with respect to event attributes.

Axiom 3.3 *Event Introduction Axiom*

The event introduction axiom for a set of transitions T of a class c is defined as follows.

$$\forall o \in \mathcal{O}(c). \bigwedge_{t \in T} (o?e@s(t) \wedge cond_t(\overline{\sigma}[o][s(t)], \overline{\tau}[o][s(t)])) \Rightarrow \bigwedge_{1 \leq k \leq l} o!e'_k@d(t)$$

where e is an input event of the transition t, $e'_1, \cdots, e'_l$ are output events of the transition t.

In this axiom, $o?e@s$ is a logical expression which represents that the event e is received at the state s in the object o. $o!e'@s'$ is also a logical expression which represents that the event e' is produced at the state s' of the object o.

Axiom 3.4 *Event Communication Axiom*

For a class c which has an output event e and a state transition t' of a class c' which has an input event e, the event communication axiom is defined as follows.

$$\begin{array}{l} \forall o \in \mathcal{O}(c).\forall o' \in \mathcal{O}(c').\forall s.\forall EA_e. \\ o!e@s \wedge (o, o') \in Link \wedge \forall s.EA_e(\overline{v}[o][s]) \wedge com(e, o', EA_e) \Rightarrow \\ \quad o'?e@s(t') \wedge EA_e(\overline{\tau}[o'][s(t')]) \end{array}$$

In this axiom, $\overline{v}[o][s]$ represents a vector of attribute values that the output event e has in the state s of the object o.

$com(e, o, EA_e)$ is defined as follows for classes $c_1, \cdots c_n$ which send the event e and $c_i \neq c, 1 \leq i \leq n$.

$$com(e, o', EA_e) \Leftrightarrow \bigwedge_{1 \leq i \leq n} (\exists o_i \in \mathcal{O}(c_i).(o_i, o') \in Link \Rightarrow \forall s.EA_e(\overline{v}[o_i][s]))$$

The event communication axiom not only represents that an event sent by an object is received by another object but also that properties of event attributes in the sending object are passed to the object receiving that event. The variable EA_e appearing in the axiom represents a property of attributes of the event e which holds in the object o. This property also holds in the object o'. In other words, the property EA_e which holds in the object o is

passed to the object o'. In addition, we need a condition which represents that the property EA_e holds in all the objects sending the event e for providing consistency to the passed property. Such condition is represented by com.

To pass a property in an object, we need the fact that the property holds in any state of that object. This fact is derived from the following *event attribute axiom*.

Axiom 3.5 *Event Attribute Axiom*

The event attribute axiom is defined as follows for an output event e that a class c sends.

$$\forall o \in \mathcal{O}(c).\forall EA_e.(\bigwedge_{t \in T} EA_e(\overline{v}'[o][d(t)])) \Rightarrow \forall s.EA_e(\overline{v}[o][s])$$

where T is a set of state transitions that have the event e and $\overline{v}[o][s]$ is attribute values of the output event e.

In the above axiom, the transition t has an action which changes attribute values of the event e. If the action changes a value of an attribute a_j into $f_j(\overline{o}[s(t)], \overline{\tau}[o][s(t)]), 1 \leq j \leq l$, $\overline{v}'$ is defines as follows.

$$\overline{v}'[o][d(t)] = f_1(\overline{\sigma}[o][s(t)], \overline{\tau}[o][s(t)]), \cdots, f_l(\overline{\sigma}[o][s(t)], \overline{\tau}[o][s(t)])$$

3.4.4 Delegation Axiom

In our model, objects of the assembly class in an aggregation relation delegates its behavior to objects of the part classes in a state specified in the unification mappings $\mathcal{U}_{del}$. The object of the part classes can not change the attributes of the assembly class but can refer to these attributes. Therefore, the local invariant assigned to that state can be regarded as global invariant in the part classes.

We introduce an axiom to regard local invariants in the assembly class as global invariants in the part class. This axiom is defined as *delegation axiom*.

Axiom 3.6 *Delegation Axiom*

If an assembly class c delegates its behavior to part classes $c_1, \cdots, c_n$ in a state s, the following axiom is introduced in the axiomatic system.

$$\forall o \in \mathcal{O}(c).\forall o_1 \in \mathcal{O}(c_1). \cdots \forall o_n \in \mathcal{O}(c_n).\forall LA_s.$$
$$LA_s(\overline{\sigma}[o][s]) \Rightarrow \bigwedge_{1 \leq i \leq n}((o, o_i) \in Del \Rightarrow \forall s_i.LA_s(\overline{\sigma}[o_i][s_i]))$$

3.4.5 Inheritance Axiom

In our model, the semantics of inheritance is just to gather attributes and functions owned by its ancestor classes. Also, we do not consider any behavioral relation between super-classes and sub-classes in the inheritance. After assertions are assigned to each class, this inheritance concept regards global invariants of a super-class as global invariants of its sub-classes. This is useful as we can derive global invariants for a super-class without both defining its behavior and proving local assertions in the super-class.

We define *inheritance axiom* which allows us to do such replacement as follows.

Axiom 3.7 *Inheritance Axiom*

If a class c is a super-class of classes $c_1, \cdots c_n$ and no state transition model is assigned to the class c by the unification mappings $\mathcal{U}_{ST}$, the following axiom is introduced in the axiomatic system.

$$\forall o \in \mathcal{O}(o).\forall o_1 \in \mathcal{O}(c_1).\cdots.\forall o_n \in \mathcal{O}(c_n).$$
$$\forall GA_1, \cdots GA_n.\bigwedge_{1\leq i\leq n} \forall s_i.GA_i(\overline{\sigma}[o_i][s_i]) \Rightarrow \bigvee_{1\leq i\leq n} \forall s.GA_i(\overline{\sigma}[o][s])$$

where the attributes whose values are represented by $\overline{\sigma}[o_i][s_i]$ are the same to the attributes whose values are represented by $\overline{\sigma}[o][s]$.

4 Computer Support

In the consistency verification that we proposed in the previous section, very complex steps must be followed. We need computer support to facilitate such proofs effectively. As we mentioned in Section 3, we should automatically generate an axiomatic system for a constructed analysis model. We implemented such environment shown in Figure 3 using ML[Pa96], HOL and Java. This environment is called *F-Developer*. F-Developer consists of the three tools, *model editor*, *F-Verifier* and *F-Prototyper*.

In F-Verifier, we verify the constructed analysis model using HOL. HOL is a theorem prover which supports higher order logic and is implemented in ML. We graphically construct analysis model such as the basic class model and the basic statechart model using model editor implemented in Java. To realize an axiomatic system on HOL from the constructed analysis model, we have to assign, as additional descriptions, a type and a definition to each attribute and function appearing in the basic class model respectively. We can describe them using pop-up dialog windows of the model editor. In verifying the constructed analysis model, we automatically transform its data in the repository into a source code for HOL to build an axiomatic system using *axiomatic system generator* in F-Verifier. The source code is automatically loaded onto HOL interpreter which is running in the background of F-Verifier. F-Verifier has a window to operate the HOL interpreter running in the background. We prove propositions using that window. In HOL, we prove target propositions in two ways. One is forward proof and the other is goal directed proof. In the consistency verification, we use facilities in HOL which follow these proving methods.

Moreover, we can execute the constructed analysis model in F-Prototyper [AK01]. In prototyping the model, we can generate an executable ML source code automatically using *prototyping code generator* and operate it using another window in F-Prototyper.

5 Related Works

A verification method called *model checking*[CGL93] is proposed to check the cooperative behavior of state transition models. This method explores all possible states under the given state transition models and automatically checks properties described in a temporal logic called *CTL*. We can check the state transition models including huge number of states using *BDD(Binary Decision Diagram)*. However, this method can only check properties of state transition models consisting of finite states. We can not directly apply this method to the object-oriented analysis model because the state transition models of the

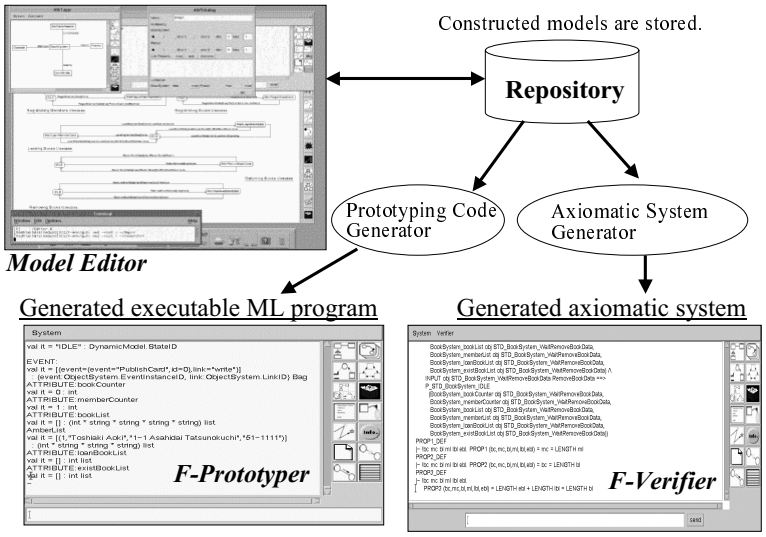

Figure 3: Architecture of F-Developer

analysis model have attributes which can hold inPnite number of values. In our approach, we verify such state transition models using mathematically proving technique such as mathematical induction.

A tool called *Alloy Analyzer*[Ja00] has been proposed for automatic analysis for Prst-order logic with sets and relations. First order logic is not decidable but this tool realizes automatic analysis by restricting its target to Pnite universe. This tool is useful in object-oriented analysis because there are many properties in the object-oriented analysis model which can be put into the Pnite universe. This approach is different from ours in that we focus on inPnite universe which allows us to deal with everything to be considered in the analysis phase and check them interactively.

Though the above two techniques are applied after extracting requirements of target systems, our approach is applied during extracting them. Our analysis method has an ability to follow the whole activities to be done in the analysis phase.

6 Conclusion

In this paper, we have proposed an axiomatic system for the consistency veriPcation of constructed analysis models according to UML models. This axiomatic system provides rigorous semantics of these models and allows us to prove assertions assigned to the models. In proving such assertions, we found that the theorem proving systems are useful for supporting its activity. Though theorem proving techniques could be adopted in many aspects of software development, we think that using the them in the analysis phase is very important as bugs or misunderstanding in this phase cause serious problems

in implemented software systems.

Theorem proving in this phase has a special meaning in contrast to, say, model checking applied to ÓmechanicalÓ elements where the correctness against predetermined criteria is the most important. Usage of theorem proving in the analysis phase has to focus on how to correctly model the world, where theorem proving process has to be considered as a part of the modeling activity. From this point of view, the use of HOL for higher order logic is very appropriate as the logic is full of modeling capability. Also, interactive proof procedure, in principle, Þts to the modeling activity done by software designer. The hardest point of using the prover is also its proof procedure. We have constructed an analysis model for a library system to manage a library business using F-Developer as an experiment. A theory module which was automatically generated from the analysis model has 60 term constants, 20 axioms and 15 deÞnitions. In this experiment, we succeeded in drastically reducing the cost for the preparation for the veriÞcation. However, many steps were still needed in each proof. A proof for consistency of book lists needed more than 30 proof steps and more than 1500 text lines are generated by HOL. This is because the proof step is so Þne-grained even though HOL system has capability for making a macro step as a *tactical*. Despite this problem, we think theorem proving systems like HOL will be useful as it has potential for modeling the world with its higher order capabilities. We should work on their customization to application domains so that software engineers in the domains can use them for their daily activities.

Bibliography

[UML99] Object Management Group, Inc: OMG UniÞed Modeling Language SpeciÞcation (version 1.3), 1999.

[HOL91] University of Cambridge Computer Laboratory: The HOL System Description, revised edition,1991

[Pa96] Paulson, L., C.: ML for the WORKING PROGRAMMER, University of Cambridge, 1996

[CGL93] Clarke, E., Grumberg, O. and Long, D.: VeriÞcation Tools for Finite-State Concurrent Systems, LNCS 803, 1993.

[Ha87] Harel, D., et.al.: On the Formal Semantics of Statecharts, Proc. of 2nd IEEE Sympo on Logic in Computer Science, pp.54-64, 1987.

[DW98] DÕSouza, D. and Wills, A.: Objects, Components and Frameworks with UML, Addison-Wesley, 1998.

[WK98] Warmer, J. and Kleppe, A.: The Object Constraint Language, Addison-Wesley, 1998.

[Fo97] Fowler, M.: Analysis Patterns - Reusable Object Models, Addison-Wesley, 1997.

[Ja00] Jackson, D.: Automating First-Order Relational Logic, International Conference on Foundations of Software Engineering, 2000.

[AK98] Aoki, T., Katayama, T.: UniÞcation and Consistency VeriÞcation of Object-Oriented Analysis Models, APSEC Õ98, pp.296-303, 1998

[AK01] Aoki, T. and Katayama, T.: Prototype Execution of Independently Constructed Object-Oriented Analysis Model ,Automating the Object-Oriented Software Development Methods, ECOOP 2001 Workshop, 2001.