Formal Action Semantics for a UML Action Language

Mikai Yang
(Heriot-Watt University, U.K.
my25@hw.ac.uk)

Greg J. Michaelson
(Heriot-Watt University, U.K.
greg@hw.ac.com)

Rob J. Pooley
(Heriot-Watt University, U.K.
ley@hw.ac.com)

Abstract: The abstract syntax and static semantics of UML, the widely-used general-purpose graphical modeling language, have been standardized in a four-layer meta-modeling framework. However UML's dynamic semantics, such as UML Precise Action Semantics and the behaviors like activities, interactions and state machines, are only standardized in a natural language—English. It is commonly argued that such informal description inevitably involves ambiguities and lacks rigorousness, precluding the early simulation and reasoning about a UML system design. Here we select Action Semantics (AS) as the vehicle to formalize UML. AS is a mature semantics description framework which has advantages of intelligibility, modularity and practicability. In our approach, we formalize UML indirectly by formalizing its textual correspondent—an extended Action Language, which plays a key role as the interface between UML and its action semantics.

Key Words: Action Semantics, formal semantics, action language, Unified Modeling Language

Category: D.3.1, D.3.1, D.2.1, D.2.2

1 Introduction

Unified Modeling Language (UML) is a general-purpose graphical modeling language that is widely applied in system design and documentation. Previously, UML lacked sufficient expressivity in describing dynamic aspects of systems, such as method bodies and exit/entry actions of state machines, and as such has to resort to a plain natural language or an existing programming language as a complementary formalism. These two compromising methods have their drawbacks: 1) using natural languages causes ambiguities in the description and hampers rigorous model checking and early system simulation; 2) using programming languages usually involves unnecessary implementation-specific details and requires the related background of the user. To address this problem, UML Precise Action Semantics (UPAS) [Mellor and Trockey 2001] has been incorporated

into UML 2.0 [OMG 2006a, OMG 2006b] to provide precise behavioral primitives such that large-scale realistic behaviors can be built systematically based on them. Consequently, UML has been made expressive more in defining the dynamic parts of the system, and thus becomes executable.

However, although the syntax and the static semantics of UML have been formally specified using MetaObject Facility (MOF) [OMG 2006c] in a four-layer metamodeling framework facilitated by Object Constraint Language (OCL), UPAS and the behaviors of UML such as state machines, activities and interactions, are only standardized in English. It is well-known that natural languages inevitably involve vagueness and lack reasonability. So it is desired to provide a formal semantics for this aspect of UML.

There have been a variety of approaches to formalizing UML within which we may distinguish three main but closely related themes: formal semantics (e.g. krtUML [Damm et al. 2003]), meta-modelling (e.g. pUML [Alvarez et al. 2001]) and translational (e.g. CASL [Reggio et al. 2001]). We draw on all three approaches, integrating them through the formalisation of a UML Action Language. We employ Action Semantics (AS) [Mosses 1992] because the formal semantics of UML demands more readability and extensibility due to its fast evolution and a wider range of users, and AS has such properties compared to other frameworks, including operational semantics [Plotkin 2004] and denotational semantics [Schmidt 1986]. Furthermore, instead of defining UML directly, we design an Action Language (AL) [Mellor and Balcer 2003] and use it as the intermediary between UML and action semantics of UML. The AL is characterized by heterogeneity, combining simultaneously the features of Object Oriented Programming Languages (OOPL), Object Query Languages (OQL) [Cattell and Barry 2000], Model Description Languages (MDL) and complicated behaviors like state machines. Thus using AS to formalize such a hybrid language has considerable significance in exploring the adequacy and applicability of AS.

2   Action Semantics Framework

AS is a mature framework for formalizing semantics of programming languages, created to make semantics specifications more comprehensible and thus more widely acceptable. AS, a hybrid framework, incorporates the best features of the following formal semantics.

Denotational Semantics (DS). In describing action semantics of a programming language, one always begins with specifying its abstract syntax, which acts as the bridge between its concrete syntax and its corresponding semantics. Then, semantic functions, each expressed by one or more semantic equations, are defined to map each phrase in the abstract syntax to a semantic entity. The semantics of a compound phrase or a non-terminal is composed by those of the

sub phrases. This quality of semantics is called *compositionality* [Plotkin 2004]. By this means, the semantics of the target language are denoted by the corresponding semantic entities, mainly *actions*. This process is extremely similar to that of describing a denotational semantics. Differently, the semantic entities in AS are intelligible actions, not the higher-order cryptic mathematic functions adopted in DS.

Structural Operation Semantics (SOS) [Plotkin 2004]. AS uses SOS to provide a formal base to *actions*, where behaviors of an abstract machine are defined. This abstract machine takes *actions* as instructions and performs them, and thus achieves computational effects, e.g. updating store, changing binding environment, sending/receiving messages and creating transient information.

Algebraic Specification. AS adopts an unorthodox framework for algebraic specification, *unified algebraic specification* [Mosses 1989, Mosses 1994], to define sorts that model abstract data types. AS has already defined a wide range of commonly-used data sorts including the primitive ones such as truth values, integer and characters, and some generic ones such as map, list and set. So the user can simply import them into their AS descriptions.

AS has advantages over traditional frameworks, such as the mentioned DS and SOS, of intelligibility and modularity, primarily because *actions* are carefully designed to be suggestive and intuitive English words. Note that the Action Notation (AN), specifying the lexical symbols of actions, has been evolved from AN-1 [Mosses 1992] to AN-2 [Lassen et al. 2000]. However, AN-1 is adopted in this paper. Owing to space limited, the reader is referred to [Mosses 1992] for the elaboration of actions and their operational semantics.

3 The Extended Action Language

We do not formalize UML directly for the following reasons.

- Since an action semantics is syntax-directed and compositional, we must specify the abstract syntax *tree* of UML before composing its action semantics. However, the abstract syntax of UML is NOT tree-structured, at least not intuitively, in that it is formalized in an object-oriented four-layer metamodeling architecture, namely using graph-like class diagrams.

- UML, as a modeling language intended for early stages in system development and a broad spectrum of different application domains, unavoidably includes some ambiguous and execution-unrelated constructs. So we need to remove these semantics-weak constructs and confine our attention to an executable subset of UML.

Consequently, we need a textual programming language corresponding to a rigorous and executable subset of UML as an interface between UML and its action

semantics. This textual correspondent is required to genuinely embody the major dynamic semantics of UML. We consider that an AL is the best candidate because it definitely incorporates most major dynamic semantics of UML; after all, ALs are created to provide concrete syntax for the basic behavioral units of UML—actions. (The reader should not confuse the actions in UPMS with those in AS).

So far, several ALs have existed for years, such as Action Specification Language (ASL) [KCL 2000b], the BridgePoint Action Language [PT 2005], and the widely-used SDL [ITU 1999] in the telecommunications industry. Although these ALs, generally speaking, have reflected the actions of UPAS, they each are not complete programming languages. They are more like scripting languages intended to being embedded in UML models in that they lack model description constructs to represent UML diagrams such as class diagrams and state charts. Therefore, we cannot simply reuse one of them as the intermediary between UML and its action semantics. This gives rise to an extended action language, called ALx, which provides not only the syntax of the dynamic aspect of UML like the existing ALs but also textual representations for UML diagrams. Note that we are mainly concerned with the dynamic semantics of UML rather than any syntax, including the syntax of ALx, so we do not detail the syntax of the selected subset of UML and ALx.

4 Related Action Semantics

AS has been applied to model a diversified range of realistic languages, such as Java [Watt 1997, Brown and Watt 1999], standard ML [Watt 1988] and Pascal [Mosses and Watt 1993]. Most relevant to our research is [Watt 1997], in which David Watt utilized AS to describe the semantics of JOOS, a subset of Java concerning the main concepts, such as classes, fields, inheritance, dynamic method selection and object constructors.

We have the following thoughts about the semantics of JOOS. On the one hand, the action semantics of JOOS has demonstrated that AS is capable of describing the major semantics of an OOPL, however it has not been demonstrated that AS is expressive or elegant sufficient to describe the semantics of a higher-level descriptive language like an OQL, or the semantics of complex behaviors such as state machines. This becomes one of our departure points of our work of applying AS to ALx.

On the other hand, we can re-use, extend or modify many parts of the action semantics of JOOS in our action semantics for ALx because the two languages share a lot in the semantics relevant to basic expressions, imperative commands and object-oriented constructs. In this way, we can take full advantages of AS modularity and extensibility to save efforts in our semantics description. This

is also an opportunity to explore and check these benefits of AS in practice. As a result, we are allowed to focus on the semantics of those constructs that are greatly different from those of JOOS, such as class declaration and object creation/destruction, and absent in ALx, such as object query, state transition and link traversal.

To sum up, our action semantics of ALx is based on Watt's action semantics for JOOS. The reader is recommended to refer to action semantics of JOOS when reading the semantics of ALx for better understanding.

5 Action Semantics of ALx

5.1 Class and Class Declaration

To model ALx classes, a user-defined and composite sort **class** is specified as follows:

$$\text{class} = \text{class of (class-token, type-bindings, method-bindings,}$$
$$\text{constructor, state-machine}^?, \text{class}^?).$$

which indicates that a class is constructed from various components as follows:

- a **class-token**, which corresponds to the simple name of the class.

- a **type-bindings**, essentially a map from token to type, where the token corresponds to the name of a field and the type to the declared type of this field.

- a **method-bindings**, essentially a map from token to method, where a method is an abstraction encapsulating an action denoting the semantics of the method body.

- a **constructor**, a special method to be invoked during object creation.

- an optional **state-machine**, representing the state-machine behavior of the class.

- an optional **class** is the direct superclass of this class.

The sort **class** is equipped with operations to access the components of classes, including **method-bindings** _ , **class-token** _ , **type-bindings** _ , **constructor** _ , **state-machine** _ , **superclass** _ and **superclasses** _ . Their uses are straightforward: for example, the operation **method-bindings** _ is for obtaining the method-bindings component of the given class. Among them, the operation **type-bindings** _ is worth highlighting, because it returns the programmer-defined type-bindings of the given class, plus a special type-binding in which the token is "_ LinkRecord" and the type is **set**. This implies, every object of every

class has an implicitly-defined field, the name of which is specially designed to be unique in the scope of an object. This treatment is used to record the links associated with the object.

The semantics of a class declaration of JOOS is that a class is constructed and then bound to a class-token which corresponds to the class name. This forms an entry of the bindings map—the scoped information. Thus, the class can be obtained based on its name. However, a class declaration of ALx will additionally allocate a cell specialized to store a list intended to memorize identities of all objects of this class. Such lists are referred to as object identity lists. Initially, in the runtime, when a class is created, the object identity list of this class is an empty list. Each time an object of this class is created, the identity of the newly created object is added to the list. Furthermore, the object identity list of a class is accessible because cell holding its object identity list is bound to a token obtained by the operation **identity-list-token** _ and specific to the class name. This semantics is described in ASD 5.1.

ASD 5.1 Class Declaration

- elaborate _ :: Class-Declaration → action [binding | storing][using current bindings | current storage].

(1) elaborate ⟦"class" I_1: Identifier "extends" I_2: Identifier "{"
     F: Field-Declaration* C: Constructor-Declaration$^?$ M: Method-Declaration*
     S: State-Machine-Declaration "}"⟧ =
     |recursively bind I_1 to the class of(the type-bindings of F, the method-bindings
     |of M, the constructor of C , the state-machine of S, the class bound to I_2).
     and
     |allocate a cell then
     ||store an empty-list in it and bind the identity-list-token of I_1 to it

---

5.2   Objects

The sort **object** is defined as follows to model objects.

$$\text{object} = \text{object of (class, variable-bindings, identity)}$$

This means, an object consists of three components: 1) a **class**, which classifies this object. 2) a **variable-bindings**, essentially a map from field names to cells which hold values of the corresponding fields. 3) an **identity**, uniquely identifying the object, which is actually a cell allocated when the object is initialized.

Likewise, the sort **object** also provides operations to access the components of the specified object, such as **class** _ , **field-variable-bindings** _ and **identity** _ . Notably, the object can be obtained from its identity by the operation **the-object-with-identity** _.

ASD 5.2 Field Initialization

- allocate an object of _ :: yielder [of a class] → action [storing ⫿ giving an object] [using current storage ⫿ current bindings]

(1)   allocate an object of c: yielder [of a class] = instantiate the field-type-bindings of c
        and allocate an identity and initialize state of c then
          |give the object of (the class yielded by c, disjoint-union (the given variable-
          | bindings#1, the given variable-bindings#3), the given identity #2)

- instantiate _ :: yielder [of type-bindings] → action [storing ⫿ giving variable-bindings] [using current storage].

(2)   instantiate t: yielder [of type-bindings] =
          |check (t is the empty-map) and then give the empty-map
        or
          |give t and choose a token [in the mapped-set of t] then
          ||instantiate (the given type-bindings#1 omitting the set of the given token#2)
          ||and
          ||give the given token #2 and allocate a variable initialized to the default-value
          ||of the type yielded by (the given type-bindings #1 at the given token #2)
          |then give the disjoint-union of (the given variable-bindings #1,
          | the map of the given token#2 to the given variable #3).

An object initialization, the major process in creating an object, takes the following procedure. Its action semantics is illustrated in ASD 5.2.

1. Instantiate the object's fields.

   (a) Obtain the field-type bindings of its class (the class is known).

   (b) Allocate a cell for each field.

   (c) Store the default value into the allocated cell based on the type of the field.

2. Allocate a cell to be the identity of the object being initialized.

3. Set the current state of the object to the initial state if its class has a state-machine behavior using the auxiliary function **initialize state of** _ .

4. Construct the object using the components produced by the previous steps.

In the procedure, the steps 1 to 3 can be carried out concurrently. This semantics is similar to the corresponding semantics of JOOS. Note that the special field "_ LinkRecord", mentioned in Sub-section 5.1, is also initialized, along with other programmer-defined fields, to an empty set, being prepared to store the associated links. Furthermore, in ASD 5.2, the auxiliary function **initialize state of** _ is used to allocate a cell to store the current state of the object if this object has a state-machine behavior. Likewise, this cell is bound to a unique token for later access.

5.3 Object Query

The object query mechanism enables retrieving objects of a given class from the runtime environment, usually based on a condition. The resulting objects are put into a variable of **set** type.

ASD 5.3 Object Creation

- execute _ :: Object-Creation → action [storing ι diverging ι escaping ι binding] [using current bindings ι current storage]

(1) execute ⟦"create-object" I_1: Identifier "of" I_2: Identifier⟧ =
        allocate an object of the class bound to I_2 then
        |bind I_1 to the given object#1 and
        |recursively add identity (the given object#1) to class (the given object#1)

- recursively add _ to _ :: identity, class → action [storing ι diverging] [using current bindings ι current storage]

(2) recursively add I: identity to C: class and its superclasses =
        ||give the identity-list stored in the cell bound to
        ||the identity-list-token of (class-token C) then
        |||store concatenation (the given identity-list, the list of I)
        |||to the cell bound to the identity-list token of I
        |and give (superclass C)
        then
        ||check (the given tuple is()) and then complete
        or
        ||check (not(the given tuple is()) and then recursively add I to the given class)

To accomplish this object query mechanism, we have deliberately used, as mentioned in Section 5.1, a special cell for a class to hold its object identity list so as to keep a record of the identities of all its objects, including the objects of all its direct and indirect sub classes. Apart from that, the following two post-conditions respectively of object creation and object deletion should be enforced.

– Whenever an object is created, its identity is put into the identity list of its class and its super classes. See ASD 5.3. We use an auxiliary function with the following signature

       recursively add _ to _ :: identity, class → action

to recursively add identity of the newly created object to object identity lists of its corresponding class and **all superclasses**.

– Whenever an object is deleted, its identity is removed from the object identity list of its class and its super classes. The formal description of object deletion is omitted here to save space.

Now that the identities of all objects of a class have been recorded, object selection is a matter of iterating over the objects collection, and picking out the

objects which satisfy the specified condition. See ASD 5.4 for details. Note that the object being visited in each iteration is bound to the token "selected" for immediately later use in evaluating conditions.

ASD 5.4 Object Query

- execute _ :: Selection → action [storing ǀ diverging][using current bindings ǀ current storage]

(1) execute ⟦ "select" I_1: Identifier "from instances of " I_2: Identifier "where" E: Expression ⟧ =
  | select instances in (the identity-list stored in the cell bound to
  | the identity-list-token of I_2) by the condition E
  | then
  | store the given set to the cell bound to I_1.

- select instances in _ by the condition _ :: identity-list, Expression → action [giving a set ǀ diverging] [using current bindings ǀ current storage]

(2) select instances in I: identity-list by the condition E: Expression =
  | check (I is empty-list) and then give empty-set
  | or
  | check (not (I is empty-list)) and then
  | give the-object-with-identity (head I) then bind "selected" to the given object
  | thence
  | | evaluate E and select instances in (tail I) by the condition E and
  | | give the given object
  | then
  | | check(the given truth-value#1 is true) and then
  | | give disjoint-union(set of(the given object#3), the given set#2)
  | | or
  | | check(not(the given truth-value#1 is true) and give the given set#2

5.4   Link Traversal

The sort **link** is defined to model links, which are instances of relations, as follows:

$$\text{link} = \text{link of(relation, (object, object), identity)}$$

This implies, a link contains its classifying relation, the connected two objects and its identity. To be simple, the sort **relation** is defined as follows:

$$\text{relation} = \text{relation of (relation-token, class, class)}$$

where the **relation-token** is corresponding to the relation name; the two **classes** are ones that participate in the relation. We do not consider multiplicities of associations because multiplicities are more related to static semantics. Parallel to classes, which are produced in class declarations, relations are generated in relation declarations. Both class declarations and relation declarations of ALx,

the model description parts of ALx, can completely represent textually a rigorous UML class diagram, the primary static aspect of the system.

ASD 5.5 Link Creation

- execute $_-$ :: Link-Creation → action [storing|diverging] [using current bindings|current storage]

(1) execute $[\![$ "relate" I_1: Identifier to I_2: Identifier across I_3: Identifier $]\!]$ =
  | allocate a cell then
  | give the link of (I_1, (the object stored in the cell bound to I_2,
  | the object stored in the cell bound to I_3), the given cell)
  | then
  | add the given link to the object stored in the cell bound to I_2 and
  | add the given link to the object stored in the cell bound to I_3.

- add $_-$ to $_-$ :: link,object → action [storing | diverging] [using current bindings | current storage]

(2) add L:link to O : Object = give the field-variable-bindings of O
  then give (the given variable-bindings at "$_-$ LinkRecord")
  then store disjoint-union of (the set stored in the given variable,
  the set of the identity of L) in the given variable.

The link traversal mechanism of ALx enables navigating from one object to another across a link and can be considered as a special kind of object query. Its fulfillment necessitates that each object records all the links connected to it. For this purpose, we have intentionally incorporated a field ("$_-$ LinkRecord") in every object, as mentioned in Sub-section 5.1. So, whenever a link is created, it is definitely added to both fields of the two linked objects. The formal semantics for link creation is shown in ASD 5.5. When the link is destroyed, it is removed from the fields. The formal semantics for link deletion is not illustrated here.

See ASD 5.6. The object selection based on link traversal involves the following major steps:

1. Give all links of the given object. This is carried out by the auxiliary function

    get the links from $_-$ :: object → action

  which returns values of the aforementioned field "$_-$ LinkRecord" of the given object. Its formal definition is not shown here.

2. For each link, see whether it is an instance of the given relation (using a defined operation $_-$ **is an instance of** $_-$).

   – If so, retrieve the object connected with the given object by this link and put it into a set. This object retrieving is accomplished by the following operation

      the object linked with $_-$ by $_-$ :: object, link → object

---

ASD 5.6 Link Traversal

- execute _ :: Selection → action [storing ∣ diverging] [using current bindings ∣ current storage]

(1)   execute ⟦ "select" I_1: Identifier "related by" I_2: Identifier "− >" I_3: Identifier⟧ =
   | give (the object stored in the variable bound to I_2) and
   | give the relation bound to I_3
   then (regive and get the links from the given object#1) then
   | get the linked objects of the given object#1
   | from the given set#3 related by the given relation#2
   then store the given set to the variable bound to I_1.

- get the linked objects of _ from _ related by _ :: object, set, relation → action [giving a set ∣ diverging]

(2)   get the linked objects of o: object from s: set related by r: relation =
   choose a link [in s] then
   | get the linked of o from the intersection of (s, the set of the given link)
   | and give the given link
   then
   ‖ check (the given link#2 is an instance of r) and then
   ‖ give disjoint-union (the set of the object
   ‖ linked with o by the given link#2, the given set#1)
   ‖ or
   ‖ check (not(the given link is an instance of r) and then give the given set#1.

  – If not, go to the next link.

  Note that this step is implemented in a recursively defined auxiliary function.

 3. Bind the resulting set to a variable in the storage.

5.5 State Machine

To represent state machines, various sorts are defined, in particular including **state-machine**, **state** and **transition-table**. Their definitions are given in ASD 5.7 and self-explainable. We highlight that the sort **transition-table** is actually a map that implements transition functions of state transitions, and the entry-action of a state is an abstraction encapsulating an action that is performed when the object moves into this state while the exit-action is performed as the object exits this state.

ASD 5.7 Sorts for modeling state machines

- state-machine = state-machine of (initial-state-token, transition-table, state-bindings)

- state = state of (state-token, entry-action$^?$, exit-action$^?$)

- transition-table = map [(state-token, event-token) to state-token]

According to UML 2.0, a state machine has an event pool which holds incoming events until they are dispatched; and the event occurrence processing is the major behavior of a state machine and is based on the run-to-completion assumption, interpreted as run-to-completion processing. The run-to-completion means that an event occurrence can only be processed if the processing of the previous event occurrence is fully completed. As for ALx, the semantics of this process is implemented in the construct "Event-Generation", which sends an event to an object with a behavior of state machine and then may trigger a state transition. ASD 5.8 shows the formal semantics of processing events, where various

---

ASD 5.8 State Machine

- execute _ :: Event-Generation → action [storing ǀ diverging] [using current bindings ǀ current storage]

(1) execute ⟦ "generate" I_1: Identifier "to" I_2: Identifier ⟧ =
        give the object stored in the cell bound to I_2 then
          ‖get the current state of the given object and regive then
          ‖enact the application of the exit-action of the given state#1 to
          ‖   the given object#2
          ‖and then
          ‖get the destination state of the given object when the event-token of I_2 and
          ‖regive then
          ‖set the current state of the given object#2 to the given state#1 and then
          ‖enact the application of the entry-action of the given state#1 to
          ‖   the given object#2

---

self-explanatory operations are defined to make the semantic description concise. Among them, the operation **get the destination state of _ when _** is an operation for searching the transition table and returning a destination state when an event occurs. Informally speaking, when an event happens, the exit-action, a method abstraction, of the current state is enacted and executed. Subsequently, the transition table is consulted for the target state, and the current state of the object is changed to this state. Finally the entry-action of the target state is executed. Note that it is assumed that any generated event will definitely cause a change of state in the state machine. This assumption is reasonable in our case because it could be assured via static analysis of the program.

6   Implementation

To give assurance of the validity of the action semantics description, we have recently implemented a prototype ALx-to-Java translator, underpinned by our formal semantic description of the action language, using Model Driven Architecture (MDA) [Miller and Mukerji 2003], where models are the first-class and
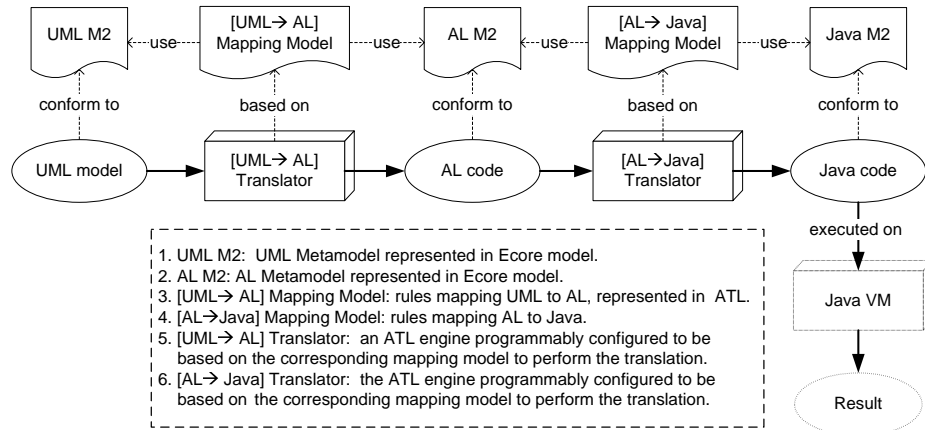
Figure 1: The architecture of the UML-to-Java translator.

major artifacts, and systems are achieved by model transformations and evolutions, and code generation. The open-source Eclipse [Eclipse 2008] projects provide us with a complete list of capable toolkits supporting MDA, particularly including:

1. Eclipse Modeling Facility (EMF) [Budinsky et al.]. The core EMF framework includes a meta model (Ecore) for describing models and runtime support for the models including serializing/loading models in/from XMI files [OMG 2002], and a reflective API for manipulating EMF objects generically. In addition, a graphical authoring tool for Ecore model is also developed by a satellite project.

2. ATL (ATLAS Transformation Language) [ATLAS 2006], a model transformation language, which can be used to formalize executable mapping rules in transforming two different kinds of models. Also available is an ATL engine, which, having the mapping rules and two different metamodels in Ecore as the knowledge base, can translate a source model conforming to one metamodel to a target one conforming to the other metamodel. Notably, the ATL engine can be easily incorporated into a Java application and all its required models and metamodels can be programmably configured in the runtime.

The general architecture, including the relevant model and metamodels, is given in Figure 1. The whole translation process is divided into two sequential sub translations, the translation from UML to ALx and the translation from ALx to Java. We compose three metamodels using Ecore, UML metamodel,
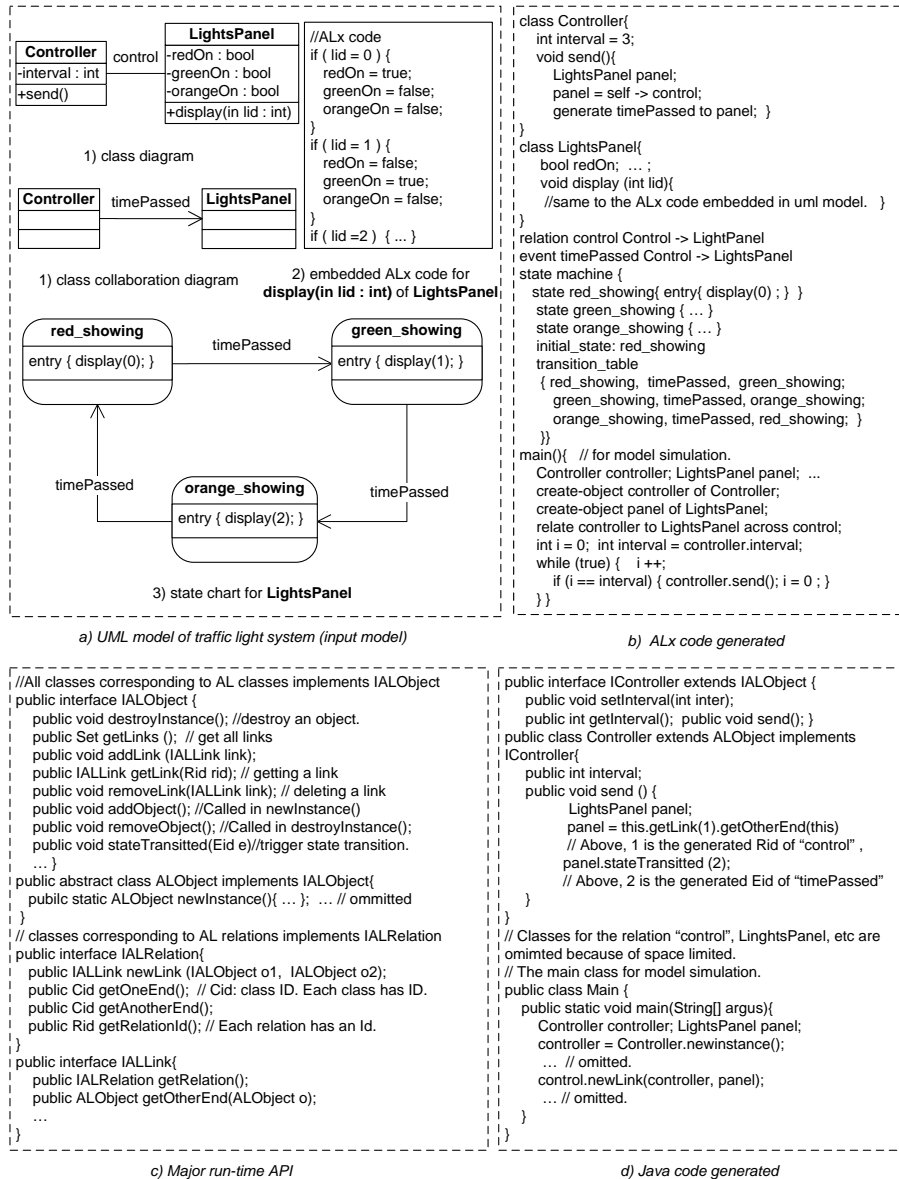
a) UML model of traffic light system (input model)

Class diagram boxes:

```
Controller        LightsPanel
-interval : int   -redOn : bool
+send()           -greenOn : bool
    control       -orangeOn : bool
                  +display(in lid : int)
```

1) class diagram

```
//ALx code
if ( lid = 0 ) {
    redOn = true;
    greenOn = false;
    orangeOn = false;
}
if ( lid = 1 ) {
    redOn = false;
    greenOn = true;
    orangeOn = false;
}
if ( lid =2 )  { ... }
```

2) embedded ALx code for **display(in lid : int)** of **LightsPanel**

```
Controller   timePassed   LightsPanel
```

1) class collaboration diagram

```
red_showing              green_showing
entry { display(0); }    entry { display(1); }
```
timePassed (red_showing → green_showing)

```
orange_showing
entry { display(2); }
```
timePassed (orange_showing → red_showing), timePassed (green_showing → orange_showing)

3) state chart for **LightsPanel**

b) ALx code generated

```
class Controller{
    int interval = 3;
    void send(){
        LightsPanel panel;
        panel = self -> control;
        generate timePassed to panel;  }
}
class LightsPanel{
    bool redOn; ... ;
    void display (int lid){
    //same to the ALx code embedded in uml model.   }
}
relation control Control -> LightPanel
event timePassed Control -> LightsPanel
state machine {
  state red_showing{ entry{ display(0) ; } }
    state green_showing { ... }
    state orange_showing { ... }
    initial_state: red_showing
    transition_table
    { red_showing, timePassed,  green_showing;
       green_showing, timePassed, orange_showing;
       orange_showing, timePassed, red_showing; }
    }}
main(){   // for model simulation.
    Controller controller; LightsPanel panel; ...
    create-object controller of Controller;
    create-object panel of LightsPanel;
    relate controller to LightsPanel across control;
    int i = 0;  int interval = controller.interval;
    while (true) {   i ++;
       if (i == interval) { controller.send(); i = 0 ; }
    } }
```

c) Major run-time API

```
//All classes corresponding to AL classes implements IALObject
public interface IALObject {
   public void destroyInstance(); //destroy an object.
   public Set getLinks (); // get all links
   public void addLink (IALLink link);
   public IALLink getLink(Rid rid); // getting a link
   public void removeLink(IALLink link); // deleting a link
   public void addObject(); //Called in newInstance()
   public void removeObject(); //Called in destroyInstance();
   public void stateTransitted(Eid e)//trigger state transition.
   ... }
public abstract class ALObject implements IALObject{
   pubilc static ALObject newInstance(){ ... };  ... // ommitted
}
// classes corresponding to AL relations implements IALRelation
public interface IALRelation{
   public IALLink newLink (IALObject o1,  IALObject o2);
   public Cid getOneEnd();  // Cid: class ID. Each class has ID.
   public Cid getAnotherEnd();
   public Rid getRelationId(); // Each relation has an Id.
}
public interface IALLink{
   public IALRelation getRelation();
   public ALObject getOtherEnd(ALObject o);
   ...
}
```

d) Java code generated

```
public interface IController extends IALObject {
    public void setInterval(int inter);
    public int getInterval();  public void send(); }
public class Controller extends ALObject implements
IController{
    public int interval;
    public void send () {
        LightsPanel panel;
        panel = this.getLink(1).getOtherEnd(this)
        // Above, 1 is the generated Rid of "control" ,
        panel.stateTransitted (2);
        // Above, 2 is the generated Eid of "timePassed"
    }
}
// Classes for the relation "control", LinghtsPanel, etc are
omimted because of space limited.
// The main class for model simulation.
public class Main {
    public static void main(String[] argus){
        Controller controller; LightsPanel panel;
        controller = Controller.newinstance();
        … // omitted.
        control.newLink(controller, panel);
        … // omitted.
    }
}
```

Figure 2: An example of a traffic light system.

ALx metamodel and Java metamodel, and two mapping models for the two sub translations. When the translator is launched, all metamodels and mapping models are loaded into the runtime environment, together with an input UML model. Note that it is also likely that some of these models have already resided in memory and do not need to be loaded for instance if the translator is integrated by a UML case tool and launched from its environment. Then, the embedded ATL engine is configured by code to do the first translation resulting in an in-memory ALx model, followed by the second translation, which is also performed by the same ATL engine, but configured differently. Finally, the Java model is obtained and code is generated. In fact, it is more sensible and efficient that we translate UML to Java directly without by-producing the ALx model.

When we compose the ALx to Java mapping model, the first principle is that it should be strictly loyal to our action semantics of ALx. Interestingly, we find the action semantics description strongly suggests the implementation of the language, especially for such high-level constructs as object creating, object query, link traversal and state machines. In other words, the action semantics is consistent with the metaphor of the language implementers, which is desired for a formal semantics. This fact corroborates the statement about AS that it has superior understandability.

The initial evaluation of this translator suggests that it is accurate and robust on small examples. One example of a simple traffic light is shown in Figure 2. We are currently investigating scalability to realistic UML models.

7   Conclusion

In conclusion, we find that AS is expressively adequate to formalize a heterogeneous language like ALx combining the features of OOPs, OQLs and MDLs. We observe that the resulting action semantics for UML is readable, and suggestive for language implementation through applying it to implementing a prototype translator in MDA approach. Meanwhile, we find that MDA is a very effort-saving and rigorous approach to implementing languages because the semantic rules can be easily and adherently represented by an executable transformation language.

We have not explored describing the concurrency of UML, e.g., asynchronous calls to behaviors, co-existence of multiple *active* objects [Selic 2004] each of which has its own thread, and asynchronous signal response, using communicative actions. This is because:

1. Each agent of AN-1, the abstraction of real computational processors, has its own local store, and no common store is provided to be shared by agents readily. It is feasible, but not trivial, to simulate a common store using

an auxiliary agent that reacts to messages about allocating, changing, and inspecting its local store.

2. If we use AN-1 to cover the concurrency of UML ignoring the difficulties in modeling the common store, we consider that the most feasible solution is that each object, whether *active* or *passive* [Selic 2004], is allocated with one agent, and interactions between objects are modeled by message exchange between agents. This solution implies that the interactions between agents may be asynchronous, or synchronous. However, using AN-1 to model synchronous communication is not straightforward and needs to resort to auxiliary agents, due to AN-1 adopting the single asynchronous notion of communications.

As such, AN-1 is not suitable, or at least not elegant, to describe some notions such as light-weight processes and threads, which probably share stores and necessitate synchronous communications. In fact, this limitation of AN-1 was realized by Mosses at the beginning [Mosses 1992].

The newly developed AN-2 will make life easier in coping with concurrency since AN-2 allows agents to share and have global access to the storage. So, our major future work is to cover the concurrency of UML using AN-2. The translator mentioned in Section 6 will be correspondingly updated. We can envisage an expressive executable subset of UML that is formally defined in both syntax and semantics, and a translator or interpreter that can execute its models and is provably correct because it is based on the formal semantics and rigorously-defined model transformations.

References

[Alvarez et al. 2001] Alvarez, J.M., Clark, T., Evans, A., Sammut, P.: "An Action Semantics for MML"; Proc. of 4th International Conference on the Unified Modeling Language (UML2001), Toronto, LNCS. (1999), Springer.

[ATLAS 2006] ATLAS group: "ATL: Atlas Transformation Language, ATL User Manual, Version 0.7" Manual, `http://www.eclipse.org`.

[Brown and Watt 1999] Brown, D.F., Watt, D.A.: "JAS: a Java action semantics [C]"; 2nd International Workshop on Action Semantics, University of Aarhus, Denmark: BRICS NS. 3, 43-56 (1999).

[Budinsky et al.] Budinsky, F., Steinberg, D., Merks, E.: "Eclipse Modelling Framework". Addison Wesley, 2003.

[Cattell and Barry 2000] Cattell, R.G., Barry, D.K.: "The Object Data Standard: ODMG 3.0"; Morgan Kaufmann (2000).

[Damm et al. 2003] Damm, W., Josko, B., Pnueli, A., Votintseva, A.: "Understanding UML: A formal semantics of concurrency and communication in real-time UML"; Proceedings of FMCO (International Symposia on Formal Methods for Components and Objects), 2582 (2003).72-99, Springer.

[Eclipse 2008] Eclipse: An open source community. Website: `http://www.eclipse.org`.

[ITU 1999] ITU: "Specification and description language(SDL)"; ITU, Tech. Rep. 1999.

[KCL 2000b] Kennedy Carter Ltd.: "UML ASL Reference Guide, ASL Language Level 2.5" Manual, `http://www.kc.com`.

[Lassen et al. 2000] Lassen, S.B., Mosses, P.D., Watt, D.A.: "An introduction to AN-2, the proposed new version of Action Notation"; AS 2000, 19-36 (2000), Dept. of Comput. Sci. of Aarhus.

[Mellor and Balcer 2003] Mellor, S.J., Balcer, M.J.: "Executable UML: A Foundation for Model-Driven Architecture" Addison-Wesley (2003).

[Mellor and Trockey 2001] Mellor, S.J., Tockey, S.: "Action Semantics for UML. Response to OMG RFP ad/98-11-01 OMG ad/1002-08-04" Tech. Rep. Project Technology, Inc.

[Miller and Mukerji 2003] Miller, J., Mukerji, J.: "MDA Guide Version 1.0.1"; OMG (2003), `http://www.omg.org`.

[Mosses 1989] Mosses P.D.: "Unified Algebras and Module"; POPL, 329-343(1989).

[Mosses 1992] Mosses, P.D.: "Action Semantics"; Cambridge University Press (1992).

[Mosses 1994] Mosses, P.D.: "Unified Algebras and Abstract Syntax": Recent Trends in Data Type Specification, Proc. 9th Workshop on Specification of Abstract Data Types, Caldes de Malavella, 1992. 785 (1994), 280-294, Springer-Verlag.

[Mosses and Watt 1993] Mosses, P.D., Watt, D.A.: "Pascal action semantics"; University of Aarhus. Tech. Rep. 1993.

[OMG 2002] Object Management Group (OMG): "MOF 2.0/XMI Mapping, Version 2.1.1" Manual, `http://www.omg.org`.

[OMG 2006a] Object Management Group (OMG): "Unified Modeling Language: Infrastructure, version 2.0" Manual, `http://www.omg.org/`.

[OMG 2006b] Object Management Group (OMG): "Unified Modeling Language: Superstructure, version 2.0" `http://www.omg.org/`.

[OMG 2006c] Object Management Group (OMG): "Meta Object Facility (MOF) Core Specification, Version 2.0;"; `http://www.omg.org`.

[Plotkin 2004] Plotkin, G.D.: "A structural approach to operational semantics"; J. Log. Algebr. Program, 60-61 (2004), 17-139.

[PT 2005] Project Technology, Inc.: "BridgePoint Action Language (AL) Manual" Manual, `http://www.projtech.com`.

[Reggio et al. 2001] Reggio, G., Cerioli, M., Astesiano, E.: "Towards a rigorous semantics of UML supporting its multiview approach"; Proc. FASE. (2001), Springer.

[Schmidt 1986] Schmidt, D.A.: "Denotational Semantics: A Methodology for Language Development"; AL lyn and Bacon, Inc., Newton, Mass (1986).

[Selic 2004] Selic, B.: "On the Semantic Foundations of Standard UML 2.0"; Lecture Notes in Computer Science, 3185, 185-199 (2004) Springer.

[Watt 1988] Watt, D.A.: "An Action Semantics of Standard ML"; Proceedings of the 3rd Workshop on Mathematical Foundations of Programming Language Semantics, 572-598 (1988), Springer-Verlag.

[Watt 1997] Watt, D.A.: "JOOS action semantics. Version 1"; `http://www.dcs.gla.ac.uk/~daw/publications/JOOS.ps`.