Formal Analysis of UML-Based Designs

Boumediene Belkhouche and Anastasia Nix
EECS Department
Tulane University
New Orleans, LA 70118
{bb,nix}@eecs.tulane.edu

Abstract

This paper describes a design specification and analysis framework to support the OO design stage. Structural and behavioral specifications of objects are integrated and formalized. To achieve this task, an object-oriented design language that captures the structural and behavioral models was defined. An environment consisting of a graphical user interface (GUI) and a processor for the language was implemented. The GUI supports the generation of UML-based designs. The major tasks of the processor are syntactic and semantic analyses, and code generation. Thus, designs can be evaluated and validated before implementation.

Keywords: object–oriented design, design specification, formal analysis, code generation.

1 Introduction

Object-oriented design provides two distinct views of systems, the structural (static) view and the behavioral (dynamic) view. The structural view describes the static structure and relationship of the objects, while the behavioral view describes the dynamic behavior of objects over time. The techniques used in structural modeling are well developed to the point of being standardized (e.g., the UML effort). The behavioral view is concerned with how the system changes as it progresses through time. It is typically modeled with concepts such as objects, services, state and state transitions, and message connections. Compared to the structural model, concepts and notations used in behavior view across different methodologies vary significantly. So far, no consensus appears to be emerging in this area. Besides this lack consensus, there seems to be a mushrooming of ad–hoc proposals all of which trying to capture a specific feature. The result is an overflow of notations which cannot be composed with each other, thus causing more ad-hoc solutions. It is an atmosphere that does not foster formality, abstraction, compositionality, and scalability. Even though, formalisms to support OO design are being developed ([1], [2], [3]), the mainstream in the field of object-oriented methods is still dominated by informal methods. These methods are typically characterized by informal pictorial notations supplemented by natural language textual descriptions. As a result, these methods lack rigor. Our research addresses these issues and provides a simple and effective solution.

The paper is organized as follows. Section 2 discusses some of the issues associated with object-oriented analysis. In Section 3 TOODL is presented. In Section 4 an introductory overview of CSP-based modeling is given followed by a description of CSP constructs used in our OO modeling framework. A framework for object-oriented behavior analysis using CSP is introduced. We then describe how CSP can be used to capture this object-oriented behavioral model. Section 5 describes the language processor implemented for our specification language. The language processor performs syntax and semantic checking on the specification of the model. In Section 6 the code generation process is described. Section 7 summarizes this work and suggests directions for future research.

2 Issues and Objectives

Within the OOD community, the state-based technique is the major approach for behavioral modeling. This approach uses state and state transitions to describe the behavior and has become one of the most common techniques. The UML has adopted it as its main behavioral modeling tool. There are two major drawbacks with this approach: state explosion and non-compositionality. It requires all possible states of a system to be identified. Theoretically, the specification of large systems can become impractical to specify due to state explosion. Also, specifications cannot be composed in a transparent way.

Our view on state-based approach is that it overspecifies as a general tool for specification and it has an unnatural fitting onto the OO execution model. A state-based approach captures object behavior at the expense of having to enumerate all the possible states of the object. States in this context imply internal states, and to specify internal states we have to look inside the object, not just its externally visible behavior. The state-based approach violates the concept of treating an object as a black box at the specification stage.

This paper addresses issues associated with the specification, analysis and evaluation of object-oriented designs expressed in a UML-equivalent notation. That is, UML designs are translated into a textual object-oriented language (TOODL) and then analyzed. TOODL was designed to capture the UML core [4]. A system to process TOODL designs (compiler) was implemented and evaluated. This system is used to analyze and validate OO designs. The proposed study is intended to provide a framework for object-oriented behavior analysis. The major objectives of this research are as follows:

1. To translate the UML designs into a formal notation.

2. To develop a framework for object behavior modeling based on the concepts of object life history and Communicating Sequential Processes.

3. To implement a processor for the proposed language to facilitate automated analysis, model validation, and code generation.

4. To illustrate the feasibility of the proposed approach and implementation by applying it to some existing examples found in literature.

3 An Object-Oriented Design Language (TOODL)

Our goal is to integrate structural and behavioral OO modeling by developing a formal framework for object–oriented design with an emphasis on behaviorial specification and analysis. The major objectives of this research are as follows: (1) to provide a GUI supporting UML designs; (2) to translate these graphical designs into a formal intermediate representation; (3) to develop an approach for object behavior modeling based on the concepts of object life history and Communicating Sequential Processes (CSP); (4) to implement a processor for the proposed language in order to facilitate automated analysis and validation; and (5) to illustrate the feasibility of the proposed approach and implementation by applying it to some examples found in the literature.

In our approach, we treat the specification of each object as a black box. This is consistent with the OO principle of encapsulation or information hiding. In the OO semantic model, message communications are a major part of how an object can affect and be affected by other objects or entities outside of its black box boundary. It is this communication and other forms of input/output of the object that we wish to model. An OO model should be able to capture abstraction, compositionality, communication, and concurrency. The CSP model fits well into the OO execution model. Multiple processes are multiple objects that must synchronize their I/O, or message communications in order to interact with one another. The issue of interaction between objects is yet another advantage of using CSP over state–based approaches. In state-based approaches, the STD technique for modeling object life cycle is unable to capture interactions between objects. Thus a separate and disjoint technique is required for modeling object interactions. In the CSP approach only one single technique is needed to capture both object life history and object interactions. Model validations and checkings as well as other forms of analysis can be performed on the specification supported by its underlying formal concept.

The need for analyzability of designs motivated the definition of a textual language for expressing object–oriented designs. This language is a synthesis of constructs proposed in the literature. TOODL is used to describe the structural and behaviroal components of an OO design. That is, TOODL contains the necessary mechanisms to define formally a set of classes and objects, the individual attributes and operations, all the relations that capture the architectural view of the system, as well as express the behavior of the design. Also, TOODL was designed to make the mapping from the UML core into TOODL straightforward.

The main components of TOODL will be explained using the abstract structure shown in Table 1. The basic unit in TOODL is the design module which consists of the following three sections:

- IMPORT. The import section is used to import classes that are externally defined.

- SUBJECT. A subject is a high-level view of a group of interrelated classes. This view provides an upper level of abstraction necessary to modularize the system.

- CLASS. This section defines the class. The structure of a class is detailed in Table 2.

Table 1. Design Module Syntax
BEGIN DESIGN *identifier*
IMPORT *import list*
SUBJECT SECTION
CLASSLIST
END DESIGN *identifier*

Table 2. Class syntax
CLASS *identifier*
BODY *properties*
ATTRIBUTES *attributes list*
RELATIONS *relations list*
OPERATIONS *operationslist*
BEHAVIOR *behaviorslist*
END *identifier*

The class section contains six further sections: CLASS, BODY, ATTRIBUTE, RELATIONS, OPERATIONS, and BEHAVIOR. The CLASS section identifies the name of the class. The BODY section defines four properties relevant to implementation: CARDINALITY, CONCURRENCE, PERSISTENCE, and VISIBILITY. The ATTRIBUTE section defines the associated attributes. In the RELATIONS section, two relationships are used to define the structure. ISA defines the inheritance structure, including multiple inheritance. HASA defines the composition whole–part structure. The interface and interaction is defined by the INSTANTIATES and CALLS relation. The INSTANTIATES models static association, while CALLS models the dynamic message connection. The OPERATIONS section defines the propotypes of the operations of the given class. The BEHAVIOR section specifies the behavior of objects and operations.

Figure 1 summarizes the constructs provided by TOODL. It supports the fundamental OO concepts of abstraction, encapsulation, modularity, hierarchy, and inheritance.

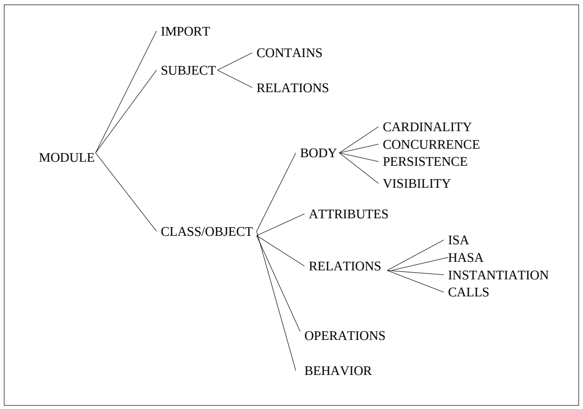

Figure 1: Constructs of TOODL

4 OO Behavior Modeling

To address the issues of state explosion, compositionality, abstraction, and formality, we developed a CSP-based behavioral model. The model consists of a number of objects (viewed as CSP processes) making up an OO system. Objects communicate with each other using communication events, and interacts with the environment with general events. Communication events are the sending and receiving of messages between objects. Exchanging information with the environment is captured by general events, which are used to model interactions between the system and the environment. The parallel composition construct from CSP enables the description of the behavior of any number of collaborating objects. This construct provides the basis for a systematic approch to compose objects into an abstract system.

To define the behavior of an object (name it Object Life History or OLH), we perform the following steps: (1) identify each object's events (alphabet); (2) construct a CSP-based specification of each object; and (3) compose the overall system by integrating the objects. The alphabet includes all the communication events and general events the object may be involved in over its lifetime. Both type of events may be either incoming or outgoing. To identify the alphabet, we examine the product of the structural analysis. Associations, services, and message connections identified during the structural analysis can give us most if not all of the events. If an event can be modeled as a message connection, it is a communication event; otherwise it is a general event.

Consider a simple car object with three relevant events, move forward, backward, and stop. Let us now look at the OLH for this car object. Event names are written in lowercase italic letters ($move\_forward$, $move\_backward$, $stop$). The three events are general events because they are events the car object is directly involved in with the environment. The alphabet for the object car (denoted as αCAR) is specified as:

$$\alpha CAR = \{move\_forward, move\_backward, stop\}$$

Afterwards, we specify the OLH using CSP. The notation $e_1 \rightarrow e_2$ means perform event e_1 then event e_2. Thus, the OLH for a car object that moves forward then stops or moves backward then stops is specified as:

$$CAR \quad = \quad move\_forward \rightarrow stop$$

$$CAR = move\_backward \rightarrow stop$$

Oftentimes, choices between different actions are made. The OLH for an object may have different next events depending on some factors. The operator | is used to denote a choice between two sequences of events:

$$(x \rightarrow P \mid y \rightarrow Q)$$

This indicates that the OLH either first engages in the event x, and then behaves as specified by P, or it first engages in the event y and then behaves as specified by Q. This is pronounced "x then P choice y then Q". If our car object can move either forward or backward, then stops, and then can move in either direction again, this can be specified by:

$$\begin{aligned} CAR \quad &= \quad (move\_forward \rightarrow stop \rightarrow CAR \\ &\mid \quad move\_backward \rightarrow stop \rightarrow CAR) \end{aligned}$$

For objects that are involved in the creation of other objects, the instantiation is modeled as a communication event. The class name to be instantiated is used to denote the channel name, and the message is instantiated. For example, if the above car object creates a trip object everytime it is moved, we can denote it as

$$\begin{aligned} CAR \quad &= \quad move\_forward \rightarrow trip!instantiate \\ &\rightarrow stop \rightarrow CAR \end{aligned}$$

Let us now look at a different, a more complicated car system example and illustrate its specification. The structure of the car system consists of four different objects: the car, engine, transmission, and brake objects, which are identified during the structural analysis. The only motion of interest is moving forward, moving backward, and stopping. The three motions represent the effects the car object have on its relationship with the environment and are modeled as general events. The interactions of the car object with its component objects are message communications between objects and are modeled as communication events. A complete specification of the Car System is shown in Appendix A and the corresponding C++ code is shown in Appendix B.

Composition of objects is supported by the the parallel composition operator. This operator allows us to compose a number of objects into a single object. The resulting composite object is treated as a single object, and can in turn be combined with other objects. This is a simple and powerful architectural model adopted from CSP. In the case of the car example, a composite car_system object is created by combining the car, engine, transmission, and brake objects. Communication events that are concealed within the object boundary are considered internal to this new object and will not be shown in the model.

$$\begin{aligned} CAR\_SYSTEM \quad &= \quad (CAR \parallel TRANSMISSION \\ &\parallel \quad ENGINE \parallel BRAKE) \end{aligned}$$

5 The Analysis System

The function of the analysis system is to perform automated model checking and verification on the specification model. The analyses performed by this system are relevant to object–oriented specification model as well as to some CSP properties. The processor accepts an input source file, processes the design specification, and generates several outputs. The outputs of the analysis are information intended to aid analysts in their process of constructing a valid OO model.

The major phases of the analysis are syntactic and semantic analyses. Syntax checking is performed on the specification to validate the legal use of language expressions. The semantic analysis is carried out by the following processes:

1. Check attributes section semantics. An attribute must obey the following rules: (1) an attribute cannot be duplicated in the same object; and (2) an attribute type must be an object of the design or an imported object.

2. Check relations section semantics. The lists of ISA, HASA, INSTANTIATES and CALLS relations are explored to detect undefined related objects and objects defined more than once.

3. Check the main semantics. The main semantic analysis is performed on the operation list of each object, i.e., here is where the interface among the objects is established. The issues are whether ISA and HASA relations are exploited, as well as the visibility of the objects.

4. Event declaration and usage check. For each OLH, these rules are enforced: (1) all events used must be declared in the alphabet; (2) All alphabets declared must be used (3) duplications are not allowed in the alphabet; and (4) events must be in one of three forms: (a) a general event: `<event>`; (b) an input communication event: `<channel>?<message>`; or (c) an output communication event: `<channel>!<message>`.

5. Process consistency check: validate the use of OLH.

6. Communication consistency check. This check is to ensure the consistency of message connections and identify spurious messages. For each communication event, the channel name must correspond to an OLH of the system. For each output communication event, there must be an appropriate input communication event at the other end of the communication channel to complete the message connection. The channel name, input/output relationship, and message name must be consistent. Formally, for a system that satisfies the event declaration and usage check, if P and Q are any two OLHs in the system, then the following must be true for all OLH P and Q:

$$\forall c!v1 : \quad \alpha P.(c = Q) \Rightarrow \exists c?v2 : \alpha Q.((c = P) \wedge (v1 = v2))$$

Extending the above property for input communication events, for a system that satisfies the event declaration and usage check, if P and Q are any two OLHs in the system, then the following must be true for all OLH P and Q:

$$\forall c?v1 : \quad \alpha P.(c = Q) \Rightarrow \exists c!v2 : \alpha Q.((c = P) \wedge (v1 = v2))$$

7. Divergence check. According to Hinchey and Jarvis, a direct opposite of deadlock is livelock. When deadlock occurs nothing will happen, but when a livelock occurs the system is out of control. In some way livelock is worst than deadlock, deadlock does not compromise the safety of the system because nothing can happen, whereas livelock does [5]. The inclusion of the CSP parallel composition operator and the hiding operator introduce the possibility of divergence.

8. Identify interface objects. Identify objects that interact with the environment and specify their interactions. Interface objects are objects that deal with the environment directly.

9. Construct service relations. Construct service relationships or client/server relationships between objects. For each object, find the client objects related to it according to services provided; and find the server objects related to it according to services requested.

10. Object-dependency cycle check. Analyze the client/server dependencies between objects. Indicate any cycles that exist in the dependency graph. Client/server relationships in the OO message passing paradigm correspond to the messages an object receives and sends. In general, any message an object receives is a request by the sender of the message for some service to be performed; the object act as a server. Any message an object sends is a request to the receiver of the message for some service to be performed; the object acts as a client. These dependencies between objects are analyzed for cycles. By uncovering cycles in object dependency, we gain an insight into the possibility of a deadlock.

11. Identify groups of cohesive classes. The identifications suggest the grouping of cohesive classes into subsystems by composition. This can also be used to indicate couplings for quality assessment. Classes not related statically by either superclass/subclass or composition should be loosely coupled. One obvious criteria for assessing cohesiveness between objects in our model is to base the cohesiveness on the number of message connections. Cohesive groups of classes are closely tied by a number of message connections, whereas non–cohesive groups of classes have few or no message connections.

12. Identify all communication channels in the system and list their alphabet. A communication channel is

unidirectional and connects exactly two objects. The alphabet of a channel consist of the set of messages that can be send through the channel.

6 Code Generation

Once a design is validated, code generation begins. Appendix B shows the C++ code generated from the object–oriented design shown in Appendix A. It consists of class headers for each class in the design and the executable code for each class OLH. Note that the statements are the events in the OLH. Since events are uninterpreted, a possible simulation would just generate the trace as defined by the OLH. Another possibility is to assign an interpretation to events (i.e., refine them) by associating a function with each event. In either case, we now have a tool to observe the dynamic behavior of the design.

7 Conclusion

An object-oriented framework to formally specify and analyze OO designs was developed. This framework consists of an OOD GUI, an object-oriented design language, a subset of CSP, and a processor. The OO behavioral model we proposed is based on externally visible behaviors of objects modeled by the notion of object life history is used to capture the behavior specification using CSP. A system to process OO designs expressed was implemented and evaluated. This system performs various analyses on the specification. These analyses are carried out to ensure that the specification defines a valid model and to give analysts insight into the model. Furthermore, C++ code is generated for valid designs.

References

[1] Evans, A. S., France, R. B., Lano, K. C., and Rumpe, B., "The UML as a Formal Modeling Notation," in *UML'98 – Beyond the Notation*, 1998.

[2] Egyed, A. F., "Automatically Validating Model Consistency During Refinement," tech. rep., University of Southern California, 2000.

[3] Engels, G., Groenewegen, L., Heckel, R., and Kuster, J. M., "A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models," 2001.

[4] Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[5] Hinchey, M. and Jarvis, S., *Concurrent Systems: Formal Development in CSP*. McGraw-Hill, 1995.

APPENDIX A

OOD of the Car System Expressed in TOODL

```
begin design car_system
begin package car_system
class Car
  attributes:
    private string: model;
    private int: year;
  relations:
    hasa Engine card[1...1],
         Transmission card[1...1],
         Brake card[1...1];
  operations:
    private
      move(string:direction): NULL;
    private stop(): NULL;
end Car

class Engine
  operations:
    public run(): bool;
    public idle(): bool;
end Engine

class Transmission
  attributes:
    private string: Transmission_state;
  operations:
    public shift(string:t_state): string;
end Transmission

class Brake
  operations:
    public apply(): bool;
    public release(): bool;
end Brake

begin behavior car_system

OLH Car
alphabet
  Car = {move<forward>,
         move<backward>,
         stop<>, Engine!idle<>,
         Transmission!shift<neutral>,
         Transmission!shift<forward>,
         Transmission!shift<backward>,
         Engine!run<>, Brake!apply<>,
         Brake!release<>}

Car = (move<forward> -> Brake!apply<>
    -> Transmission!shift<forward> ->
       Brake!release<> -> Engine!run<>
    -> Car
     | move<backward> -> Brake!apply<>
    -> Transmission!shift<backward> ->
       Brake!release<> -> Engine!run<>
    -> Car | stop<> ->
       Transmission!shift<neutral> ->
       Engine!idle<> -> Brake!apply<>)

OLH Engine
alphabet
  Engine = {Engine?run<>, Engine?idle<>}

Engine = (Engine?run<> -> Engine
       | Engine?idle<> -> Engine)

/** Definition of Transmison
    and brake omitted
**/
end behavior car_system
end package car_system
end design car_system
```

APPENDIX B

C++ Code Generated from the OOD of the Car System

```
//Car.h
class Car {
    private:
       string model;
       int year;
    public:
       Car();
       ~Car();
       void move(string);
       void stop();
};

//Brake.h
class Brake {
    private:
    public:
       Brake();
       ~Brake();
       bool apply();
       bool release();
};
//Transmission.h
class Transmission {
    private:
       string Transmission_state
    public:
       Transmission();
       ~Transmission();
       string shift(string);
};
//Engine.h
class Engine {
    private:
    public:
       Engine();
       ~Engine();
       bool run();
       bool idle();
};
// Executable Code
process Car {
  move(forward);
  Brake!apply();
  Transmission!shift(forward);
  Brake!release();
  Engine!run();
  Car();
  CHOICE
    move(backward);
    Brake!apply();
    Transmission!shift(backward);
    Brake!release();
    Engine!run();
    Car();
  CHOICE
    stop();
    Transmission!shift(neutral);
    Engine!idle();
    Brake!apply();
}
// Further code omitted
```