Proceedings of the Fourth IASTED International Conference
MODELLING, SIMULATION, AND OPTIMIZATION
August 17-19, 2004, Kauai, Hawaii, USA

VALIDATING INTER-OBJECT INTERACTION IN OBJECT-ORIENTED DESIGNS

Vishv Malhotra and Simon C Stanton
School of Computing, Private Box 100,
University of Tasmania, Hobart 7001 Australia
{vishv.malhotra, sstanton}@utas.edu.au

ABSTRACT

Object classes are the building blocks for object-oriented software. Design methodologies have focused on methods, tools and representations to build classes taking advantages of inheritance and encapsulation properties. The guiding principle being that if all classes are correctly constructed a system consisting of objects in these classes will be correct. Efforts to include object constraints in object-oriented programs have not attained the role commensurate with the role invariants play in traditional imperative programs in understanding the programs and in establishing correctness properties. The paper describes use of a model checker to establish the correctness of an object-oriented design.

KEY WORDS

Finite state process, Object-oriented design, Invariants, Object constraints, Model-checking.

1. Introduction

The program execution fundamentals of object-oriented systems are rooted in the imperative programming paradigm. Thus, the need for invariants and execution stages [1], [2] remains important in understanding the correctness and other properties of object-oriented programs and systems. The dominant developments in object-oriented methodologies have focused on class construction; correctness of interactions among objects has received a smaller role in these methodologies.

A design is not ready just because each class has been designed inheriting behaviour and code from appropriate super-classes. We need to also be sure that the class objects will interact with each other correctly. A methodology exclusively focused on object and class interfaces does not address some basic but important design needs:

- How do we know that all object classes have been defined?
- How do we know that all methods of interest have been found?
- How do we know that all behavioural details of interest have been captured in the specifications?
- Inconsistency in the specifications is another global property that escapes the confines of a single class interface.

We need a methodology that can consider properties of individual classes as well as properties of groups of classes and their objects.

Imperative languages, for example C [3], use the procedural abstraction as the central design methodology for understanding and comprehending software and its development processes. Invariants and predicates [2], [4] relate points in the static text of a program with the (expected) state that would exist when the *correct* program reaches those points during the execution. More recently Java has incorporated the traditional imperative language style assertions in the language. Thus, through pre-conditions and post-conditions programmers are able to express some, but not all, aspects of the contracts that server object methods have to the client objects invoking the methods. However, emphasis away from functional and procedural abstraction makes it difficult to associate a location in the static text of the object-oriented program with the states during execution. It is not convenient to write invariants expressing the system properties defining the system states during the program execution. The Object Constraint Language OCL [1] has somewhat limited success in expressing constraints on the values (states) in the programs.

Functional programming [5] alleviates invariants and the execution stages. The composition rules are defined to determine the properties of the composed functions from the properties of the composing functions. Functional programming has been a source of many insights and innovations in the way we program today. However, the focus of this paper being object-oriented paradigm we shall not be following an approach based on functional programs further in this paper.

In this paper, we suggest the use of model checking [6] tools as a way to express and verify properties that encompass multiple objects and their classes. Specifically, we use Labelled Transition System (LTS) by Magee and Kramer [7] as the verification tool. The tool models a concurrent system of objects as a composition of finite state processes (FSP). LTSA (Labelled Transition System Analyser) being an analyser for a concurrent system focuses on establishing the *safety* and *liveness* properties of the modelled systems.

A model is an abstract specification of a system. Each object in the model is represented as a concurrent component. The invariant properties of the object-oriented model can be expressed as the safety properties over the LTS description. A deadlock or a liveness concern in the LTS model has interpretation in the object-oriented domain underscoring an issue that has remained unaddressed.

In Section 2, we briefly describe the software development process as we view it. The section also provides some basics of the LTS specification language and its processing. In section 3, using a few examples from a case study we give a flavour of the kinds of outputs one receives from the tool. Section 4 summarises the results from the effort and makes some concluding remarks listing the advantages the approach delivers to the object-oriented software development methodologies.

2. Object-Oriented Software Development

We follow validation led software development process [8]. In brief, the object-oriented software development begins with a text description of the system. The objects and object classes can usually be discerned from the text. For each class, some data members may also become obvious at this stage. Class hierarchies and other inter-class relationships are organised to take advantage of standard object-oriented modelling practices, for example UML [9] [10]. To obtain a complete and consistent specification, the methodology suggests the use of object lifecycle models.

For each significant object class, the text description usually provides an initial description of the object's lifecycle. For example, verbs provide clues to the existence of various states. Nature of these verbs may suggest various forms of transitions between the states of the system.

However, text descriptions are notorious for their ambiguity and inconsistency [11]. At the same time, much of the description is generally left unexpressed. One does not expect the initial lifecycle models of the object classes drawn from the text to be perfect models. An extensive example is provided in [8] to describe a validation led process that can be used to iteratively develop the lifecycles to their final refined levels.

In each iterative cycle of the validation led development, the lifecycles are matched against each other to identify inconsistencies and incompleteness. Each identified lacuna requires the lifecycles to be revised to correct the concern. The reported example, however, relied on a manual analysis of the lifecycles to identify the lacunae. A tool to perform this analysis is a necessary step to improve the reliability and effectiveness of the methodology.

We address this need by using a model checker to identify the mismatches in the object lifecycle specifications. The emerging specification being formal delivers another potential benefit in the form of automating the task of program generation. This paper, however, does not pursue this goal.

We shall use the text description of a lift in a building as it appears in [8] (and, reproduced in [12]) without reproducing it here. The lift (elevator) system in a building is generally well-known and does not need re-stating here. We do not expect the readers to have any difficulty in understanding the example used in this paper. Figure 1 shows the lifecycle of a lift passenger from a text description of a lift system. As the analysis progressed, the specifications were refined, and it became evident that a passenger arriving at the ground floor or the top floor had a lifecycle somewhat different from the lifecycles of those on the other floors of the building.

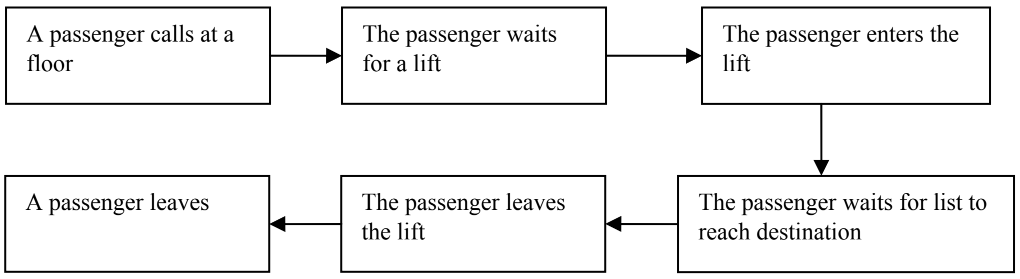
A passenger calls at a floor
The passenger waits for a lift
The passenger enters the lift
The passenger waits for list to reach destination
The passenger leaves the lift
A passenger leaves

Figure 1: Initial finite state process for a lift passenger as it emerges from a text description.

2.1 Labelled Transition System

Using Labelled Transition System (LTS) [7] we can model the lifecycles of the entities as finite state processes (FSP). The system analyser (LTSA) can analyse the processes for progress and safety violations. In this section, we give a brief flavour of an FSP description. A finite state process, or simply a process, consists of a sequence of actions. As an action occurs the system changes its state over a finite set. It is often helpful to define a finite process in terms of other finite processes.

Figure 2 shows a process modelling a lift passenger as process `PASSENGER`. In an LTS description, uppercase identifiers denote processes and lowercase identifiers denote actions. Thus, the process `PASSENGER`, can follow one of the three alternative sequences of actions. In each alternative, an action of calling the lift is followed by a wait process. The actions and processes may use parameters for better expressive power – for example, the arriving passengers wait for the lift on the floor from which they called the lift. Multiple instances of a process can be distinguished by adding a prefixing label – all actions of the process will carry the label. A `WHEN` guard can be added to the alternatives to control the choice of the alternative. Without a guard one alternative is chosen non-deterministically. In addition to defining a process in terms of the other processes it is possible to run multiple processes in parallel (concurrently).

The LTS Analyser can verify a given model for two kinds of errors. A *progress* violation occurs when the system reaches a state from which it can not guarantee an occurrence of an action from a set of actions. For example, a progress violation would have occurred if the modelled system can not assert that, at all instances of time, the lift will visit the ground floor again at some time in the future.

```
const UP = 0
const DOWN = 1
Range UPF: 2..MAX_FLR-1
Range DIR: UP..DOWN

PASSENGER = {call_at_ground_level
        -> WAITING_FOR_LIFT[1]
   | call_at_top_floor ->
        WAITING_FOR_LIFT[MAX_FLR]
   | call_at_floor[f: UPF][d: DIR]
        -> WAITING_FOR_LIFT[f]
}
```

Figure 2: Finite State Process PASSENGER defined using actions and other FSPs.

There is a special process called `ERROR`. The process `ERROR` can be reached explicitly by specifying actions leading to it. Alternately, one can specify one or more *safety* properties. A safety property is a sequence of, not necessarily consecutive, actions that represent an acceptable behaviour. Any violation denotes an error prompting process `ERROR` to manifest. A safety requirement may insist that each `door open` action is followed by a `door close` action before the lift moves.

Each safety or progress violation detected by the tool is reported by the analyser by an action sequence leading to the deadlock or error state. This we found to be very useful information in correcting the errors and remodelling the lifecycles of the involved entities and processes.

3. Experience with the model checker

In this section, we report about our experiences in the use of the model checker for validation led development of object-oriented software. The full study can be accessed in [12].

To keep the report focused we consider the movements of one lift in a multi-floor building. At various points in its lifecycle, the lift invokes algorithm `WALK` to determine the next action that the lift should execute. The points on the lift lifecycle at which the algorithm is invoked are when the lift door closes; when the lift approaches the next floor level; and by an idle lift when a floor button is pressed to call the lift.

Initially a rather rudimentary `WALK` algorithm was coded in the FSP model. The LTS analyser was then used to report incompleteness and inconsistency errors in the model. Figure 3 depicts a LTSA report for a model of the lift system. The annotations have been added to the analysis report to provide easy interpretation. The reader will notice that the description does not (yet) fit with the typical configuration of a real lift system. The differences represent the still-evolving state of the FSP model used in the illustration. The example model is not the final model.

As errors were reported by the analyser, alterations were made to improve the model. Table 1 summarises a sample of the reported errors and the actions taken to correct the reported errors. The full final version of the lift specification spans some 19 pages.

3.1 Software Testing versus Model Checking

Table 2 provides an indication of the efforts required to verify the finite state process (FSP) model of the lift for various sizes of the model. These provide some interesting insights into the traditional software design and develop methodologies. A naïve black-box testing [13] would tend to show growth in the required number of test cases in proportion to the potential state-space size. As is obvious from the numbers appearing in Table 2, any practical testing exercise based on the black-box approach can only test a microscopic fraction of the potential test cases.

White-box testing [13] takes advantage of implementation-related information. Thus, it will follow the growth trend shown as reachable state space and/or as the number of transitions. Again, the practical testing efforts can cover only a small fraction of the test cases needed for a perfect result.

The model checking, notwithstanding its tedium, emerges as a powerful tool if we wish to deliver high quality software that is substantially free from errors and mistakes. Model checkers, such as LTSA, contribute to this process in many ways:

- The formal FSP descriptions that LTSA requires are directly associated with the objects in the system specifications.
- The FSP description of the objects is formal and is capable of interpretative execution. It can be run in steps through an animator – an integrated part of LTSA – for better understanding of the nature of a problem and the circumstances leading to it.
- The verification process employed by a model checker is like a simultaneous execution of all animations – analysis provides an effective and efficient mean for identifying all potential violations of progress and safety properties in the specifications.
- Each identified violation is reported as a shortest sequence of actions that leads to the identified violation. This makes it easy to comprehend the problem and supports the efforts to correct it.
- The formal specifications can be easily transformed into programs in Java and other object-oriented languages through automated and semi-automated processes.

```
Composition:
LP = p.1: PASSENGER || LIFT (btnUp, btnDown, dptCount) || btnUp.1:BUTTON ||
btnUp.2:BUTTON || btnDown.2:BUTTON || btnDown.3:BUTTON || dptCount.1:BUTTON ||
dptCount.2:BUTTON || dptCount.3:BUTTON
State Space:
 22 * 882 * 3 * 3 * 3 * 3 * 3 * 3 * 3 = 2 ** 29
Analysing...
Depth 25 -- States: 217 Transitions: 445 Memory used: 4482K
Trace to DEADLOCK:
      p.1.arrival.1                // p.1 arrives at level 1 of the building
      p.1.passenger.1              // wants to ride the lift from level 1
      p.1.call.1.1                 // calls the lift
      delay.1                      // waits – this lift is right there!
      door_is_open_i.1.1           // Lift door opens
      dptCount.1.seek_button.0     // Lift checks: the departure count at floor 1 is 0 –
      btnUp.1.seek_button.1        // Lift checks: the up button at floor 1 shows 1 person waiting
      btnUp.1.off                  // Lift turns the first floor up button off
      p.1.enter_lift               // Passenger p.1 may board the lift
      p.1.entered_lift.1           // p.1 enters lift
      p.1.press_dest.1             // p.1 wants to go to floor 1 – same floor
      pause
      door_is_closed.1.1           // Lift door closes p.1 is inside
      dptCount.1.seek_button.1     // Lift determines that it needs to go to floor 1
      door_is_open_i.1.1           // Lift opens the door
      dptCount.1.seek_button.1     // Anyone getting down here?
      p.1.destination_reached.1    // p.1 is to be let off here
      p.1.left_lift.1              // p.1 leaves
      p.1.arrival.1                // p.1 is back on the level 1
      p.1.passenger.1              // wants to ride the lift from level 1. Again!
      dptCount.1.seek_button.0     // Lift checks: the departure count at floor 1 is 0
      btnUp.1.seek_button.0        // Lift checks: the up button at floor 1 shows 0 persons waiting
      btnDown.1.seek_button.1      // Lift checks: down button at floor 1 shows 1 person waiting
                                   // btnDown.1 is defined by LIFT(btnUp,btnDown,dptCount)
      btnDown.1.off                // Lift turns the first floor down button off
Analysed in: 180ms
```

Figure 3: LTSA analysis report indicating a sequence of actions leading to a deadlock in a FSP model for a lift system. The comments have been added to provide interpretation for the readers of this paper.

Table 1: Summary of some errors reported by the analyser during validation led development of a FSP model for a lift system in a building with 3 floors.

ERR OR	DESCRIPTION OF THE ERROR	COMMENTS/SOLUTION
2	Passenger arrives at floor 3 and calls lift. Lift arrives at floor 3 and passenger gets in. Presses for floor 1. The door closes, and lift invokes `WALK` algorithm. The algorithm proceeds to on-floor down button and then goes into idling.	Problem arises from a support process in `WALK`, specifically the first guard of LOOK_DOWN_INTERNAL required range check to be equal to lowest floor number to allow the recursive call to reach the in-lift button at the first floor.
7	Passenger arrives at the ground floor and presses the call button. The door opens and the passenger enters the lift and indicates to travel to the current floor. The door closes and then opens and the passenger leaves the lift. A passenger then arrives at the ground floor but does not press call yet. The lift invokes `WALK` but halts. (This case is the one reported in Figure 1)	The actions that return the number of calls in any queue are called at the top of the DROPPING_PASSENGERS process without excluding illogical combinations of floor and button direction. For example, the bottom floor in this scenario has a down button. The lift reads this error state as an instruction to travel down, but cannot, so halts. To rectify this situation the initial seeks in DROPPING_PASSENGERS are wrapped up in local processes so that the correct behaviour is applied to each of the three separate cases. This creates the processes DP_ANY_FLOOR, DP_TOP_FLOOR and DP_BOTTOM_FLOOR.
14	The passenger arrives at the second floor and calls the lift. The lift then travels up to the second floor and opens the door. The passenger enters the lift and indicates to travel up to the top floor. The door is closed and the `WALK` algorithm completes and the lift enters the idle state. The passenger is still in the lift!	The passenger had initially indicated to travel down, however when the passenger was in the lift the passenger indicated to travel up. The lift though is expecting to go down and so performs the `WALK` in the downward direction. When the `WALK` reaches the bottom it believes that it has looked everywhere and so goes to idle. The fix for this involved adding in two new local processes at the composite level and using them as Boolean masks against having checked up or down, so that the lift would look up if still needed and vice versa. These new local processes are termed LOOKED.
22	P calls to go up from 2nd floor, lift goes to second floor, 2nd passenger calls lift from third floor, to go down. The 1st P gets in and indicates to go down! The lift does it scan, is still in a mood to go up, and sees the third floor call. The lift goes to the third floor, and should open the door, instead, hits T17 which fulfils current conditions against guards, and so travels on to the fourth floor without stopping at the third.	This problem would not arise if the Ps did what they said what they were going to, ie travel in the direction they indicated. In the absence of a property to enforce this behaviour, which is not realistic anyhow, another solution is needed. What should happen? The lift should pick up the 2nd passenger at the third floor, and then move on from there.

4. Conclusions

Notwithstanding a successful application of a model checker in verifying the object-oriented design of a lift system, the methodology needs further developments to be universally applicable. The model checkers available inexpensively often have limitation in regard to the size of the systems they are able to analyse successfully. State explosion problem is the Achilles' heel for the model checkers.

At the same time, the benefits that model checkers provide in verifying the design is important [14, 15]. It has long been understood that quality cannot be added to software after it has been developed. Software engineers are well aware of the rapid escalations in cost at later software development phases. Software development practices, methodologies and tools continuously strive to make it possible to find errors earlier in the development process.

Table 2: Growth in the state space size for the modelled lift system with the number of floors in the building and the number of simultaneous lift users

Number of simultaneous passengers in the modelled system		Number of floors in the modelled building			
		3	5	7	10
1	Reachable states	1267	5067	12987	35952
	Potential state space	$\sim 10^{24}$	$\sim 10^{33}$	$\sim 10^{40}$	$\sim 10^{50}$
	Number of transitions	1327	5267	13407	36852
2	Reachable states	8236	57899	209454	
	Potential state space	$\sim 10^{36}$	$\sim 10^{52}$	$\sim 10^{67}$	
	Number of transitions	9146	812408	224180	
3	Reachable states	56664	697580	3580800	
	Potential state space	$\sim 10^{41}$	$\sim 10^{58}$	$\sim 10^{73}$	
	Number of transitions	68388	812408	4060332	
4	Reachable states	405172			
	Potential state space	$\sim 10^{53}$			
	Number of transitions	537502			

The paper has illustrated that model checkers support the software engineering goals well. They not only provide a comprehensive route for testing the software design earlier than the traditional testing based methodologies, but also promise to help automate the following stages of the software development.

REFERENCES

[1] J. Warmer & A. Kleppe, *The object constraint language – precise modeling with UML* (Reading:Ma: Addison Wesley Longman, 1999).

[2] E.W. Dijkstra, *A Discipline of programming,* (Englewood Cliffs, NJ: Prentice-Hall 1976).

[3] B. Kernighan & D. Ritchie, *The C programming Language*, (Englewood Cliff, NJ: Prentice Hall, 1988).

[4] R. Sethi, *Programming languages: concepts and constructs*, (Cambridge, Ma: Addison-Wesley, 1996)

[5] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Communications of ACM, 21*(8), 1978, 613-641.

[6] B. Berard, , M. Bidoit, A. Finkel, F. Laroussinie et al, *Systems and software verification: Model-checking techniques and tools*, (Berlin: Springer-Verlag. 2001).

[7] J. Magee, J. & J. Kramer, *Concurrency: State models and java programs, (*Chichester, England: John Wiley & Sons, 1999)

[8] C.A. Lakos & V.M. Malhotra, Validation led development of software specifications, *International Journal of Modelling and Simulation, 22*(1), 2002, 57-74.

[9] G. Booch, J. Rumbaugh & I. Jacobson, *The Unified modeling language user guide* (Reading, Ma: Addison Wesley Longman Inc. 1999)

[10] T. Quatrani, *Visual modelling with rational rose and UML*, (Reading, Ma: Addison Wesley Longman Inc. 1998)

[11] I. Sommerville, *Software engineering,* (Wokingham, England: Addison Wesley Publ. Co., 1995)

[12] S.C. Stanton, *Validation and verification of software design using finite state process (Honours thesis)*, (Hobart, Australia: School of Computing, University of Tasmania, 2002).

[13] W. Perry, *Effective methods for software testing*, (NY: John Wiley & Sons, 1995)

[14] S.C. Stanton & V Malhotra,, Model checking an object-oriented design, *Proc. 6th Intl. Conf. on Enterprise Information systems*, Porto, Portugal, Vol 3, 2004, 605-608.

[15] S.C. Stanton & V. Malhotra, Validation led development of object-oriented software using a model verifier*, Proc. Of the IADIS International Conf. Applied Computing*, Lisbon, Portugal, 2004, pp. II 7-10.