VALIDATION LED DEVELOPMENT OF OBJECT-ORIENTED SOFTWARE USING A MODEL VERIFIER

Simon Stanton
School of Computing, University of Tasmania
Private Box 100, Hobart 7001 Australia
sstanton@postoffice.utas.edu.au

Vishv Malhotra
School of Computing, University of Tasmania
Private Box 100, Hobart 7001 Australia
vishv.malhotra@utas.edu.au

ABSTRACT

Object-oriented methodologies focus on the design of object classes as the building blocks of systems. The class interface provides a way to encapsulate focus to a single object/class at a time. However, general system-wide issues are important and need attention in the design endeavour also. The paper reports on our efforts to use a model verifier to enact interactions of multiple objects and classes to perform a system-wide analysis.

KEYWORDS

Object-oriented design, Invariants, Finite-State process.

1. INTRODUCTION

Object oriented analysis, design and programming (Booch 1998; Rumbaugh 1999) is the method of choice for developing software today. Objects are designed to model real and perceived entities. Entities are modelled through their states and their behaviour. The state is defined by a set of instance and class variables (attributes). The behaviour is mapped onto methods. Objects with shared behaviour are grouped as a class; methods are defined for the whole class. Each object, however, has its individual and independent state.

A methodology exclusively focused on the object and class interfaces may not address the following questions: 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 confine of a single class interface.

In this paper we report on use of a model verification tool to address these questions. We used Labelled Transition System (LTS) (Magee and Kramer 1999) as the verification tool. The tool models a concurrent system of objects using Finite State Process (FSP). The associated analyser, LTSA (Labelled Transition System Analyser) is designed for analysis of concurrent systems but, as it turns out, is suitable for analysing object-oriented designs too.

In Section 2, we briefly describe the software development process as we view it. In section 3, an example from a case study is presented. Section 4 concludes the paper with suggestions about the benefits that the use of model verifiers may provide to object-oriented design.

2. SOFTWARE DEVELOPMENT METHODOLOGY

Validation led software development process (Lakos and Malhotra 2002) begins with a text description of the system. The objects and object classes are initially discerned from the text as are some of the attributes (data members) of the classes. It is usually possible to arrange classes into inheritance hierarchy and other inter-class relationships using standard object-oriented modelling practices and tools. To move the design to a consistent and complete specification the methodology advocates the use of object lifecycles.

For each significant object class, the text description provides the lifecycle description. It is possible to identify some of the main states, and transitions between them, from the text description. However, text descriptions are notorious for being ambiguous, incomplete, and inconsistent (Sommerville 1995). One does not expect the lifecycles of the object classes to be the *ready-to-use* specifications. Validation led process iteratively analyses and develops the object lifecycles.

In each iterative cycle of the validation led specification, the lifecycles are matched against each other to identify inconsistencies and incompleteness. Each identified mismatch requires the lifecycles to be revised to address the identified concern. The reported methodology, however, relies on a manual analysis of the lifecycles. A tool to do this analysis is essential to make the methodology reliable and effective.

We used a model verifier to identify the mismatches in the object lifecycle specifications. Each object is modelled by representing its lifecycle as a concurrent LTS 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. The LTS specifications being formal, the approach also has potential to automate the task of program generation. However, this goal was not pursued in this work.

2.1 Labelled Transition System (LTS) and LTS Analyser

Labelled Transition System (Magee and Kramer 1999) uses Finite State Process (FSP) descriptions of the entities. A Finite State Process consists of a sequence of actions terminating in a special (pre-defined) process `STOP`. It is often helpful to define a finite process in terms of other finite processes. For example, a process modelling a passenger joining a lift (elevator) system can read as follows:

```
const UP = 0, DOWN = 1

PASSENGER = {
  call_at_ground_level -> WAITING_FOR_LIFT[1]
| call_at_top_floor -> WAITING_FOR_LIFT[MAX_FLR]
| call_at_floor[f:2..MAX_FLR-1][d:UP..DOWN] -> WAITING_FOR_LIFT[f]
}
```

In the above example modelling a lift passenger, uppercase identifiers denote processes and lowercase identifiers denote actions. Thus, a process `PASSENGER` can follow one of the three alternative sequences of actions. In each alternative, the action of calling the lift is followed by a wait process. Parameters in the processes and actions are useful mechanism for passing values between states.

In addition to defining a process in terms of the other processes it is possible to run multiple processes in parallel (concurrently). Actions in two finite state processes with the matching names are synchronised and must happen on all concurrent processes containing the action name simultaneously.

The analyser can verify a given model for two kinds of errors. A progress violation occurs when the system gets into a state other than process `STOP` from which it can not perform any further action.

There is another special process called `ERROR`. The process `ERROR` can be reached explicitly by specifying actions leading to it. Alternately, one can specify safety properties. A safety property is a sequence of, not necessarily consecutive, actions that represent an acceptable behaviour. A violation of the safety property denotes an error prompting process `ERROR` to manifest.

The analyser, LTSA, takes an LTS description and generates an animation. The animation can be used to pace through the defining entities lifecycles. The animator is a useful tool for visualisation but has only a

limited application in debugging the specifications. We rely on the model verification functionality of LTSA for this purpose

A *safety check* involves exercising all possible sequences of the actions that may cause process `ERROR` to be entered. For example, for a lift system we would wish to ensure that the lift door is closed before the lift moves. Likewise, one may specify the safety property that requires the (simulated) lift to be empty before it enters its `idle` state.

The other check that LTSA performs is exposing *progress violations*. Again, the analyser checks all possible sequences of actions to find a sequence for which there is no further action possible. A sequence of actions ending in `STOP`, however, is not a bad sequence in this respect.

There is considerable leeway in modelling certain requirements. For example, a safety requirement can be programmatically specified as a guard, and therefore manifests as progress violation rather than as a safety violation.

Each safety or progress violation detected by the tool is accompanied by an action sequence that reports how the system ends in a blind alley (deadlock) or an error state. We found this to be very useful information for correcting errors and subsequent remodelling of lifecycles of involved entities and processes.

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 4 floors.

Error No.	Description of Progress Violation or Deadlock reported by the analyser	Comments and Solution used to correct the reported condition
2	Passenger arrived at floor 3 and called lift. Lift arrived at floor 3 and passenger stepped in. Pressed for floor 1. The door closed, and lift invoked WALK algorithm. The algorithm proceeds to check on-floor down button and then idle the lift.	Problem arose from a process in WALK – specifically the first guard of LOOK_DOWN_INTERNAL process required a range check to match the lowest floor number to let the recursive calls reach the in lift button at the first floor.
5	Passenger arrived at top floor, and called lift. The lift travelled up to the top floor. The door opened and the passenger stepped in and pressed the button for the same floor. The door closed and the lift invoked WALK and then entered the idle state. The passenger is still in the lift.	The invoked WALK algorithm was unsuccessful because the instruction assumed that the conditions at that point meant the entire WALK check has been performed when there is no further passenger to drop or pick at a floor further up in the current direction of travel of the lift. The entry to the idle state is replaced by an instruction to go to IDLE after an instruction to continue the WALK in the opposite direction to allow the lift to look for calls from floors on the other side of the travel.
23	A passenger called at the second floor and the lift arrived at the second floor. One more passenger called at the second floor, before the lift opened its doors. The first passenger entered the lift and pressed the in-lift button for the third floor; the second passenger entered the lift and pressed the in-lift button for the first floor. The lift door closed. One more passenger arrived and called the lift to the third floor. Lift travelled to the third floor and opened the door and the first passenger left. The door closed and a passenger called at second floor. Trace halted. Passenger three is left at third floor, the forth passenger at the second floor, and the second passenger (going to first floor) is in the lift.	Provision of a correct re-entrant behaviour to the WALK algorithm corrected this behaviour. This was the last error reported in our study.

3. AN EXAMPLE

To test the use of model verifiers in object oriented design we simulated an object based model of a lift system (Stanton 2002). Some further results to relate model verification effort with the program testing strategies are presented in (Stanton and Malhotra 2004). The study was focused on issues related to the movements of a lift in a multi-floor building. Instead of focusing on the full functionality of the simulator, we restricted ourselves to the lift controller that determines the next action of the lift. The controlling algorithm was termed WALK. At various points in its lifecycle – for example, when the lift door closes – the lift invokes algorithm WALK to determine the next action that it should execute.

The algorithm evolved in stages as the specifications were refined. Initially, only a rudimentary WALK algorithm could be discerned from the text description of the lift problem. This simplistic algorithm was coded in the FSP models. As the errors – incompleteness and inconsistency – were reported by the analyser during the specification development process, alterations were made to correct the algorithm. Each error detected by the LTS analyser required changes in the lifecycle model of one or more objects. The Lift Problem being well known meant that required changes were clear once the need was detected. Real world applications would normally require reference to domain experts for advice to correct mistakes. We had a total of 23 iterations of the refinement step. A sample of there errors and the actions taken to correct them is shown in Table 1.

4. CONCLUSION

It is well understood that quality cannot be added to software after it has been developed (Sommerville 1995). Software engineers are well aware of the rapid escalations in the cost of bug removal (and fixing) in the later software development phases. Therefore, software development methodologies and tools continuously strive to find errors in the earliest software development phase possible.

Model verifiers fill this goal very effectively for object-oriented system development. This has been illustrated by the example described in this paper.

Model verifiers such as LTSA (Magee and Kramer 1999) contribute to this process in many ways. Firstly, the formal FSP descriptions that LTSA requires is directly associated to the objects in the system specifications. The formal FSP supports interpretative execution and can be run in steps through an animator – an integrated part of LTSA. Thirdly, the verification process 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. Fourthly, the formal specifications can be easily transformed into programs in Java and other object-oriented languages through automated and semi-automated processes.

REFERENCES

Booch, G. et al, 1999. , *The Unified Modeling Language User Guide*, Addison-Wesley Object Technology Series, Addison Wesley Longman Inc., Reading, Ma.

Lakos, C. and Malhotra, V., 2002. Validation Led Development of Software Specifications, *International Journal of Modelling and Simulation*, Vol. 22 No. 1, pp. 57-74.

Magee J. and Kramer, J., 1999. *Concurrency: State Models & Java Programs,* Worldwide Series in Computer Science, John Wiley & Sons, Chichester, England..

Rumbaugh, J., 1999. *The Unified Modeling language Reference Manual,* Addison-Wesley Object Technology Series, Addison Wesley Longman Inc., Reading, Ma.

Sommerville, I., 1995. *Software Engineering,* International Computer Science Series, Addison Wesley Publishing Company, Wokingham, England.

Stanton, S. C., 2002. *Validation and Verification of Software Design Using Finite State Process (Honours thesis)*, School of Computing, University of Tasmania, Hobart, Australia.

Stanton, S. C. and Malhotra, V., 2004. Model Checking an Object-Oriented Design*, Sixth International Conf. on Enterprise Information Systems, Porto, Portugal (to appear).*

[redacted]

To: "Vishv Malhotra" [redacted]
Subject: Applied Computing 2004: submission results

Dear Author

I am pleased to inform you that your submission to the IADIS International Conference Applied Computing 2004 has been accepted as a "SHORT PAPER".

Please,
1 - make the suggested corrections to your paper (see details below), use the correct format available at http://www.iadis.org/ac2004/submissions.asp (very important: if this format is not followed we cannot accept your contribution and it won't be published in the proceedings). Make sure that your submission has the number of pages allowed for this category which is 4 pages (additional pages up to 2 will be charged as specified in the registration form). Also note that your final submission must be a WORD file since proceedings are produced in WORD,
2 - submit the final version at http://www.iadis.org/confman_ac2004/final_papers.asp (please submit from this link only), and
3 - register for the conference (deadline for this procedure is 9 February 2004 - if not registered the paper won't be published in the proceedings. Also, to take advantage of the early registration rate you must register for the conference until 9 February 2004). Registration is available at http://www.iadis.org/ac2004/registration.asp

Hope to see you in Lisbon, in March.

Best regards,
Pedro Isaias
Applied Computing 2004 Conference Co-Chair

****

Please consider the following data for your final submission (using the link above):

Your Login is: [redacted]

Your password is: [redacted]

****

Paper Title: VALIDATION LED DEVELOPMENT OF OBJECT-ORIENTED SOFTWARE USING A MODEL VERIFIER

Submission code: 259

Evaluation Results:

[redacted]

Originality: 5 - Average
Significance: 6 - Good
Technical: 6 - Good
Relevance: 7 - Excellent
Classification: 6 - Good
Comments: I like your use of an LTS analyser to verify liveness and safety properties. I find it acceptable. However, I am concerned about the limited level of originality reported in this work (in section 3) since it is based on only one case study and have not seen generalized conclusions/lessons, future developments, etc...

Originality: 4 - Neutral
Significance: 4 - Neutral
Technical: 4 - Neutral
Relevance: 4 - Neutral
Classification: 3 - Not Very Good
Comments: The paper presents the application of a model verifier to identify the mismatches in the object lifecycle specifications.
However it is not clear the real contribution of this paper. LTSA is a well know verification tool which can be used for different purposes. Do the author want to put the emphasis on the use of LTSA for developing and refining specification? In this case they should describe better the section 3, presenting the procedural steps performed and use of LTSA.
Moreover the paper is not well structured and presented. It is hardly to read and concepts are written in a confused manner. Moreover many times the sentences are very short or even not related each other
In detail:

Section 2.1 in a section with title “Label transition system” there should be the description of the LTS. What does it means “ overwhelming the reader with strange notations and interpretation”. The LTS are formal and well described and not ”strange notations”. It would be better to report the exact definition of LTS. In the paper it is only described an example of a text based description.
In any case it could be better a restructuring of the section, which is described in muddled way. Probably a table, in which for each construct is associated the right meaning, can clarify the presentation.

Section 2.2 It could be interesting knowing how the properties are specified and used by the LTSA.

Section 3. Developing a consistent specification is not a trivial problem. Rarely in the real word industries and software developers provide with a complete LTS description of the system.
Moreover the number of transitions, even if for a trivial example like the lift elevator system, can be so high to prevent any application of the methodology proposed. These are the main limitations of the approach presented in this paper and should be considered and put in the right light.
Even if the proposed approach could be more complete with respect to

techniques like white box and black-box testing, sometimes they are the only techniques applicable using the information and the specifications provided in the real word.
More criticisms could be given to the description of this experience.

Positive Points: The paper describes and interesting application of the LTSA but the presentation must be improved including more details of the use of the verifier tool.

Negative Points: The paper should be restructured describing clearly its focus of the paper, its contribution and case study analysed

Originality: 3 - Not Very Good
Significance: 6 - Good
Technical: 2 - Weak
Relevance: 6 - Good
Classification: 2 - Weak
Comments: This paper describes the use of a model verification tool.

This reviewer failed to see the added value of this paper. The paper is not about the model verifier, it simply uses it in one relatively simple scenario. The connection to OO development -- one of the major claims of the paper -- is also not clear at all. The paper does address the questions raised in the 4th paragraph of the introduction. The example could perfectly be modelled, designed and implemented without OO.

Maybe there is something interesting and different in this paper; if so, the authors need to make the message more clear.

Typo:
- in deed -> indeed