Using Models to Assess Impact of Defective Software

Juan C. Augusto[1], George Wilkie[1], Chunping Li[2], Hui Wang[1] and Jun Liu[1]

[1] CSRI and School of Computing and Mathematics, University of Ulster, Coleraine, U.K.
[2] School of Software, Tsinghua University, Tsinghua, P.R. China

Abstract. We explain a new strategy to model a set of java classes and to abstract a model from them that can be use to study the impact that defects affecting specific classes can have on the whole system. We use the models of the software implementation as an abstraction of the software that can be used for experimentation We used simulation and verification in SPIN but the idea can be applied to implementations in other languages than java and the analysis of defects impact can be done with other verification tools as well.

1 Introduction

As software applications become more complex, software reliability is becoming more important. To improve reliability we need to reduce probability of software failure, which can be catastrophic. An important approach to reducing probability of failure is by predicting and then removing software defects. Existing methods for defect prediction include those proposed in (Gaffney, 1984), (Khoshgoftaar and Seliya, 2004) and (Subramanyam and Krishnan, 2003). These methods are all machine learning based using some program features or program complexity measures such as the CK measure (Chidamber and Kemerer, 1994). The performance of these methods is still not desirable and much research is needed. For example, Menzies et al. studied defect prediction for 5 software systems of NASA (Menzies et al., 2004), with an average prediction accuracy of 36%. (Wilkie and Hylands, 1998) reported that approximately 48% of defects could be predicted by the combination of two of the CK measures.

This paper presents part of the findings in our BEACON project which focuses on the creation of strategies that facilitates the study of software defects and their consequences. We present here one of the strategies we investigated which as a qualitative analysis focus and relies on the abstraction of models out of a collection of java classes in such a way that the fundamental entities and interactions are represented. This model is then used to assess the effect of injecting defects in different parts of the system and through simulation and verification (Berard et al., 2001) study how they affect the system. The content of the paper is organized as follows. Section 2 explains the step of model creation then in section 3 we explain how we used it and we finalize in Section 4 with conclusions and proposed further work.

Augusto J., Wilkie G., Wang H., Liu J. and Li C..
Using Models to Assess Impact of Defective Software.
DOI: 10.5220/0003596000710077
In *Proceedings of the International Joint Workshop on Information Value Management, Future Trends of Model-Driven Development, Recent Trends in SOA Based Information Systems and Modelling and Simulation, Verification and Validation* (MSVVEIS-2011), pages 71-77
ISBN: 978-989-8425-60-7

2 Model Extraction

Part of the project was devoted to qualitative analysis of defects and how features from the code can be used to create models which inform the analysis of impact of those defects in the overall system. Starting with a java implementation of a system we looked at the classes and their interaction to extract a model. We choose to model systems using a well known and respected system: Promela/SPIN (Holzmann, 2003).

The analysis of a java implementation used for didactical purposes led the identification of classes/methods which suggested the network of possible interactions amongst the classes depicted in Figure 1. These classes are used to illustrate basic concepts in java to the students of one of our modules.

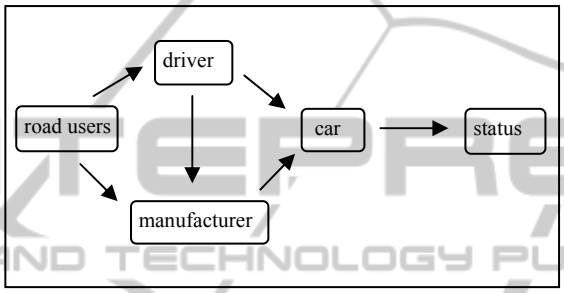

Fig. 1. Network representing the interactions of classes.

We use this network as a basis to define the processes and their interactions in the Promela model. A basic transformation algorithm can be easily implemented to translate a network into a Promela model:

1) for each node create a process with that name
2) for each transition create a channel with the names of both
3) the direction of the arrow in the graph dictates who send/receives the message
4) the interaction can be normal or defective
5) once a process is contacted by a defective module it sends a defective message to everyone it interacts with and then blocks

A more subtle analysis can be carried of in the java code to identify the modality of that interaction:

- for any interaction: is it mandatory or is only a possibility?
- when more than one call, are they independent or related (all have to be done)
- if more than one incoming transition: are both required as a precondition for the receiving node to operate or just some of them are enough?

In our example we considered all the arrows in figure 1 as “and’s”, that is that a node with two outgoing arrows call methods from both destination nodes and that a node with two incoming arrows is called by both classes in the origin nodes (notice this still omit details like in which order the methods are called) then we can refine the above models to obtain (a subset of the first model of this doc).

The following model is a first approach to representing possible behaviour in the

set of java classes under examination.

```
mtype = {ok};
chan car_status = [0] of {mtype};
chan driver_car = [0] of {mtype};
chan driver_manufacturer = [0] of {mtype};
chan manufacturer_car = [0] of {mtype};
chan roadusers_driver = [0] of {mtype};
chan roadusers_manufacturer= [0] of {mtype};
active proctype roadUsers ()
{
    if  /* i.e., methods in the classes of
        both destination nodes are invoked
        but we do not know in which order
        so we need to provide SPIN with the
        option to choose any of the
        possible orders */
    :: roadusers_manufacturer!ok ;
       roadusers_driver!ok
    :: roadusers_driver!ok;
       roadusers_manufacturer!ok
    fi
}
active proctype driver ()
{   if
    :: roadusers_driver?ok -->
       if
       :: driver_car!ok ;
          driver_manufacturer!ok
       :: driver_manufacturer!ok;
          driver_car!ok
       fi
    fi
   }
active proctype manufacturer ()
{  if  /* it has to be contacted by both in
       any order to proceed contacting the
       next class */
   :: driver_manufacturer?ok ;
      roadusers_manufacturer?ok -->
      manufacturer_car!ok
   :: roadusers_manufacturer?ok ;
      driver_manufacturer?ok  -->
      manufacturer_car!ok
   fi
}
active proctype car ()
{  if  /* it has to be contacted by both in
      any order to proceed contacting the
      next class */
   :: driver_car?ok; manufacturer_car?ok-->
      car_status!ok
   :: manufacturer_car?ok; driver_car?ok-->
      car_status!ok
   fi
}
active proctype status ()
{  if
   :: car_status?ok
   fi
}
```

3 Exploring the Model

Once we have a model we can take advantage of SPIN's multiple features to examine a model and inform us of the potential implications of a system built following the strategy represented by that model.

On first inspection the model created in the previous section does not seem to uncover any problems. Figure 2 shows a simulation.

We can now inject faults into the system. This was done manually but a recommendation of our project is that this can be incorporated to a tool that first distils a model from a group of classes and then allows the user to select in which areas of the model to inject faulty interactions which can then be analyzed using SPIN.

The following model has been injected faulty interactions in a couple of arbitrary places. The message types are now offering SPIN two types of interactions. A line "manufacturer_car!defect" has been added to proctype 'manufacturer' and another line "manufacturer_car?defect --> skip" has been added to proctype 'car'. We have also modified the model to allow the checking of some behavioural properties which we can use to force SPIN to conduct a more rigorous analysis. We have added to the model introducing a boolean variable 'end_reached' which we use to create a "guarantee" formula checking if the end of 'status' can be reached (see that 'status' is the final class in according to the network of Figure 1).


```
mtype = {ok, defect};
chan car_status = [0] of {mtype};
chan driver_car = [0] of {mtype};
chan driver_manufacturer = [0] of {mtype};
chan manufacturer_car = [0] of {mtype};
chan roadusers_driver = [0] of {mtype};
chan roadusers_manufacturer = [0] of {mtype};
bool end_reached=false;
active proctype roadUsers ()
{
    if  /* i.e., methods in the classes of
        both destination nodes are invoked
        but we do not know in which order
        so we need to provide SPIN with the
        option to choose any of the
        possible orders */
    :: roadusers_manufacturer!ok ;
       roadusers_driver!ok
    :: roadusers_driver!ok;
       roadusers_manufacturer!ok
    fi
}
active proctype driver ()
{   if
    :: roadusers_driver?ok -->
       if
       :: driver_car!ok ;
          driver_manufacturer!ok
       :: driver_manufacturer!ok;
          driver_car!ok
       fi
    fi
```

```
   }
active proctype manufacturer ()
{  if  /* it has to be contacted by both in
          any order to proceed contacting the
          next class */
   :: driver_manufacturer?ok ;
      roadusers_manufacturer?ok -->
      manufacturer_car!defect
   :: driver_manufacturer?ok ;
      roadusers_manufacturer?ok -->
      manufacturer_car!ok
   :: roadusers_manufacturer?ok ;
      driver_manufacturer?ok  -->
      manufacturer_car!ok
   fi
}
active proctype car ()
{  if  /* it has to be contacted by both in
      any order to proceed contacting the
      next class */
   :: driver_car?ok; manufacturer_car?ok-->
      car_status!ok
   :: manufacturer_car?ok; driver_car?ok-->
      car_status!ok
   :: manufacturer_car?defect --> skip
   fi
}
active proctype status ()
{  if
   :: car_status?ok
   fi;
   end_reached=true
}
```

If we run a verification of that property, SPIN detects a problem as indicated in Figure 3.

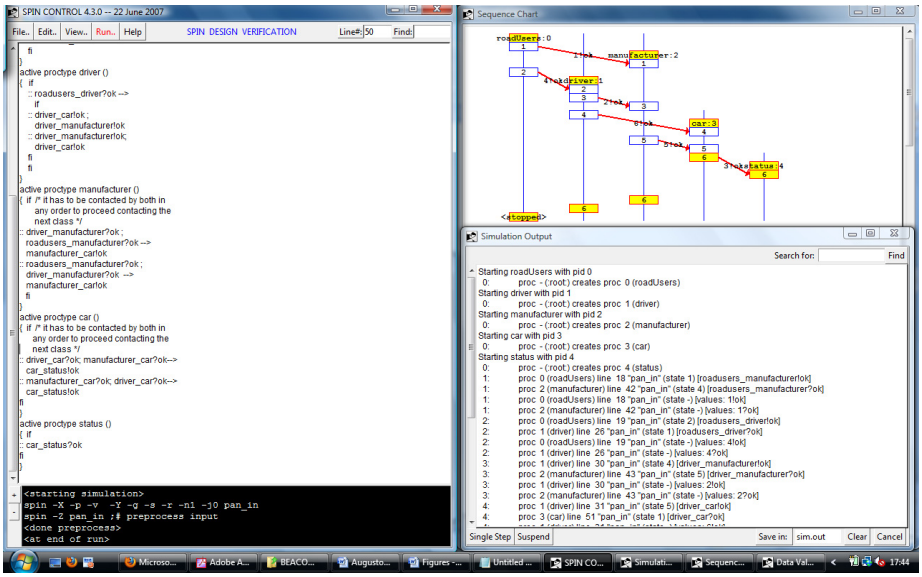

Fig. 2. A simple simulation in SPIN showing a possible way of interaction amongst classes.

If we follow the Guided Simulation offered by SPIN we get a counterexample as feedback which we show in Figure 4 and shows the system gets stalled when ‘car’ expects an OK from ‘manufacturer’ and gets a defective interaction instead.

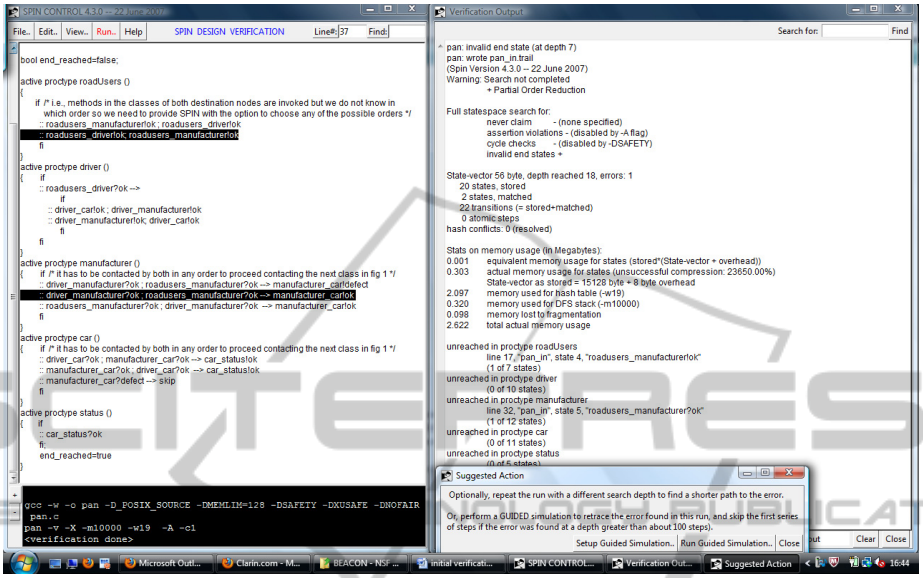

Fig. 3. Problems detected by SPIN after injection of faults in the model.

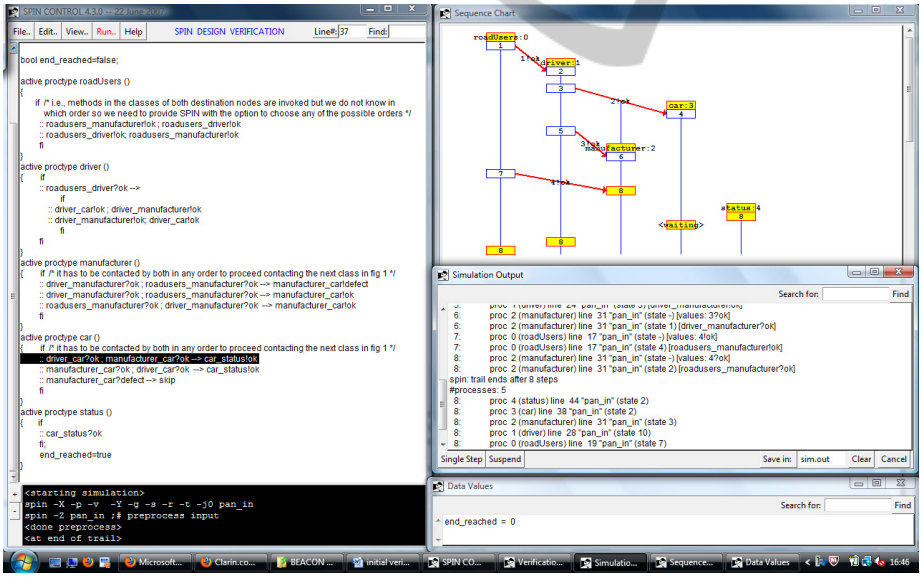

Fig. 4. Simulation with SPIN after faults in the model were detected.

4 Conclusions

The most important finding on this aspect of the project is that:

- there is a way to automatically translate java into Promela so that the impact of defects can be studied.
- a tool can be built which can perform the translation and also allow the injection of Promela code in the model representing faulty classes. These classes may be those which have been identified by statistical methods and this will enrich the assessment of the effect of defects within a system.

The strategy we explored in this paper is one of several approaches. The focus in this strategy was qualitative. Other strategies considered tried to quantify which classes may be more prone to faults. There is no single strategy or group of strategies that can provide definitive answers to these problems but a number of them may help to identify were a team should focus their efforts.

Acknowledgements

We want to acknowledge the financial support offered by the Royal Society to the BEACON project under the eGAP2 initiative.

References

1. Gaffney, J. R. Estimating the Number of Faults in Code, IEEE Trans. Software Eng., 10(4), 1984.
2. Khoshgoftaar, T. M. and Seliya, N. The Necessity of Assuring Quality in Software Measurement Data, Proc. 10th Int'l Symp. Software Metrics, IEEE Press, 2004.
3. Subramanyam, R. and Krishnan, M. S. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects, IEEE Trans. Software Eng., 29(4), 2003.
4. Chidamber, S. R. and Kemerer, C. F. A Metrics Suite for Object-Oriented Design, IEEE Trans. Software Eng., 20, 1994.
5. Menzies, T. DiStefano, J. Orrego, A. and Chapman, R. Assessing Predictors of Software Defects, Proc. Workshop Predictive Software Models, 2004.
6. Wilkie, F. G. and Hylands. B. Measuring Complexity in C++ Applications. Software Practice & Experience, 28(5), 1998.
7. Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P. Systems and Software Verification: Model-Checking Techniques and Tools, Springer Verlag, 2001.
8. Holzmann, G. J. The Spin Model Checker Primer and Reference Manual, Addison-Wesley, Reading, Massachusetts. 2003.