# Behavioural Analysis of an I$^2$C Linux Driver

Dragan Bošnački[1], Aad Mathijssen[1], and Yaroslav S. Usenko[2]

[1] Technische Universiteit Eindhoven, The Netherlands
[2] Centrum Wiskunde en Informatica, Amsterdam, The Netherlands

*Introduction.* Formal methods for the analysis of system behaviour offer solutions to problems with concurrency, such as race conditions and deadlocks. We employ two such methods that are presently most applied in industry: model checking and static analysis on a common case study to analyse the behaviour of a Linux driver for I$^2$C (Inter-Integrated Circuit).

An industrial client provided us with the source code of the driver for which it was known that it contained defects. Based on the code, some documentation, and feedback by the developers we extracted a model of the device driver. The model was checked using the mCRL2 toolset [3] and some potential defects were revealed which were later confirmed by the developers. The errors were caused by inconsistent use of routines for interrupt enabling and disabling, resulting in unprotected references to shared memory and calls to lower-level functions. In addition, we performed checks with UNO [4], a static analysis tool that works directly with the source code. We employed UNO to statically detect the errors that were found by the dynamic analysis in the model checking phase. Based on our findings, we modified the source code to avoid the discovered potential defects. Although some errors remained unsolved, an improvement was observed in the standard tests that were carried out with our fixed version.

*The* I$^2$C *Linux driver.* In general, the Linux 2.6 kernel contains an I$^2$C driver stack that is split up into three layers [5]: chip driver, core module and bus driver. The core module is part of the Linux kernel, as are a number of chip drivers and bus drivers. In our case, an I$^2$C bus driver was supplied by the client. The code mainly performs two tasks: handle ioctl calls from user space, offered via the core module, and handle interrupts from the hardware.

To find race conditions we focused on the interaction between the two parallel components of the driver: the ioctl handler and the interrupt service routine.

*mCRL2 analysis.* The mCRL2 language and toolset [3] allows users to model and automatically verify the behaviour of distributed systems. Systems can be modelled using a process algebra enriched with data types. Automated verification is supported by checking temporal properties on all states of the model.

Based on the source code of the I$^2$C bus driver we have created an mCRL2 model consisting of a translation of the ioctl handler and the interrupt service routine and the environment in which these functions occur. For the verification of our model we focused on violation of mutual exclusion of shared memory accesses. Exploration of all states and transitions revealed two types of violations: more than 100 concurrent shared memory accesses and one concurrent access of low-level functions.

M. Alpuente, B. Cook, and C. Joubert (Eds.): FMICS 2009, LNCS 5825, pp. 205–206, 2009.

These violations were caused by misplaced or absent calls to functions that disable and enable interrupts. We fixed this by making a number of small changes to the source code, by moving or adding these functions to protect the usage of shared memory and low-level functions. We have also made these changes to our mCRL2 model. Verification of this model showed us that these violations have been resolved.

State space exploration for instances involving multiple ioctl threads became prohibitively large. To resolve this, we have employed symbolic techniques as implemented in the LTSmin toolset [1].

*Static Analysis Results.* We applied UNO to find the same violations as reported by the mCRL2 analysis. The mutual exclusion properties needed to be encoded as property automata. A property automaton monitors the traversal of the control flow graphs of the C functions. UNO produces an error trace, in case a violation of the property is found.

After formulating the property automata, UNO was able to reproduce all possible defects that were discovered with mCRL2: the errors of accessing shared memory without previously disabling interrupts and unsafe function calls.

*Conclusions.* By means of both model checking using mCRL2 and static analysis using UNO, we were able to find possible non-trivial defects, which have been confirmed by the developers. Furthermore, we have provided a verified fix for the found defects.

Although in general model checking is a more powerful technique than static analysis, in this case study it seems that they are evenly matched. We think that this is due to the low number of parallel components involved in the properties we wanted to check. Instead of choosing between model checking and static analysis, we can also use them in tandem, e.g. by employing static analysis as a light-weight analysis to locate possible problems. Once the possible defects are located, one can apply the more expensive fully-fledged model checking only to the critical modules in the code base.

A more detailed account of this summary can be found in [2].

## References

1. Blom, S., van de Pol, J.: Symbolic Reachability for Process Algebras with Recursive Data Types. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 81–95. Springer, Heidelberg (2008)
2. Bošnački, D., Mathijssen, A., Usenko, Y.S.: Behavioural analysis of an $I^2C$ Linux Driver, CS-Report 09/09, Technische Universiteit Eindhoven (2009)
3. Groote, J.F., Mathijssen, A.H.J., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.J.: Analysis of distributed systems with mCRL2. In: Alexander, M., Gardner, W. (eds.) Process Algebra for Parallel and Distributed Processing, pp. 99–128. Chapman and Hall, Boca Raton (2008)
4. Holzmann, G.J.: Static Source Code Checking for User-Defined Properties. In: Proc. World Conference on Integrated Design & Process Technology, IDPT (2002)
5. Kroah-Hartman, G.: I2C Drivers, Part I. Linux Journal (December 2003)