Behavioural analysis of an I2C Linux driver

Behavioural analysis of an I^2C 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

Abstract. We present an analysis of the behaviour of an I^2C Linux driver, by means of model checking with the mCRL2 toolset and static analysis with UNO. We have reverse engineered the source code to obtain the structure and interactions of the driver. Based on these results, we have semi-automatically created an mCRL2 model of the behaviour of the driver, on which we have checked mutual exclusion properties. This revealed non-trivial potential errors, like unprotected usage of shared memory variables due to inconsistent locking and disabling/enabling of interrupts. We also applied UNO on the instrumented source code and were able to find the same errors. These defects were confirmed by the developers.

1 Introduction

Formal methods for the analysis of system behaviour offer solutions to problems with concurrency, such as race conditions and deadlocks. In this paper we employ two such methods that are presently most applied in industry [24]: model checking [11,29] and static analysis [26]. We use these techniques to analyse the behaviour of a Linux driver for an I^2C (Inter-Integrated Circuit) device.

We present some experiences and results of the project that we carried out within the Laboratory for Quality Software (LaQuSo), at the Computer Science Department of the Technische Universiteit Eindhoven, for an industrial client.[3] The goal of the project was to analyse the feasibility of the techniques used within LaQuSo, like model checking and advanced static analysis, for industrial size software. For that purpose, an I^2C Linux device driver was chosen as a case study. The 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 [16] 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 function calls to lower-level functions. In addition, we performed checks with UNO [19], a static analysis

[3] Because of confidentiality, the references to the client are made anonymous and the names of the components, functions, and variables are changed. However, this does not affect the presented results.

tool that works directly with the source code. Besides standard static analysis checks, UNO allows the user to define custom properties, which we employed 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.

Map of the paper. The next section introduces the I^{2}C Linux driver. Section 3 describes the reverse engineering we applied to the source code. In Sections 4 and 5 the results of the model checking and static analysis, respectively, are presented. The Conclusions (Section 6) evaluates the two techniques used, and discusses related and future work.

2 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 [22,23]:

1. Chip driver: a device-dependent part which interacts between user space and the core module.
2. Core module: a device-independent part containing an implementation of the I^{2}C protocol.
3. Bus driver: a device-dependent part which interacts between the core module and the actual hardware.

The core module is part of the Linux kernel, as are a number of chip drivers and bus drivers. This layered structure is depicted in Fig. 1.

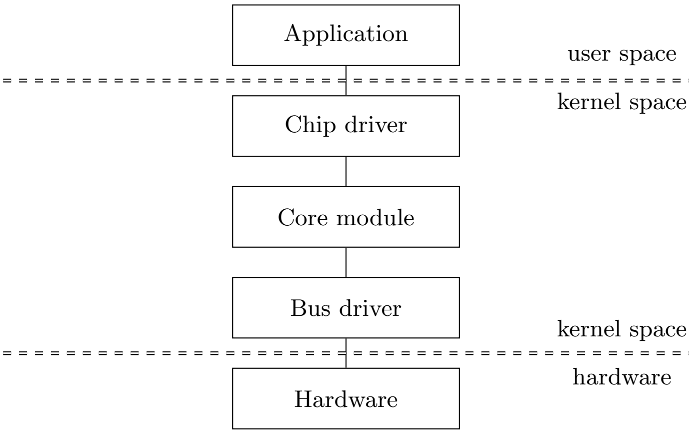


Fig. 1. High-level layer structure of the I^{2}C Linux driver stack.

In our case, an I^{2}C bus driver was supplied by the client. The code mainly performs two tasks:

- handle input/output control (ioctl) calls from user space, offered via the core module;
- handle interrupts from the hardware.

By means of testing, the client had encountered race conditions in their driver, but could not localise them. For this reason, the client was interested in discovering such dynamic issues, preferrably by means of verification.

3 Reverse engineering the structure and behaviour

Since race conditions are typically caused by the interaction between parallel components, we have chosen to focus on the interaction between the two parallel components of the driver: the ioctl handler and the interrupt service routine.

In order to get insight into the structure of the I^{2}C bus driver and the interaction between its components, we have used the SQuAVisiT toolset [28] to generate a call graph. From this graph, we could see how the ioctl handler and the interrupt service routine communicated: via shared memory and via the hardware. A high-level view of this is depicted in Fig. 2.

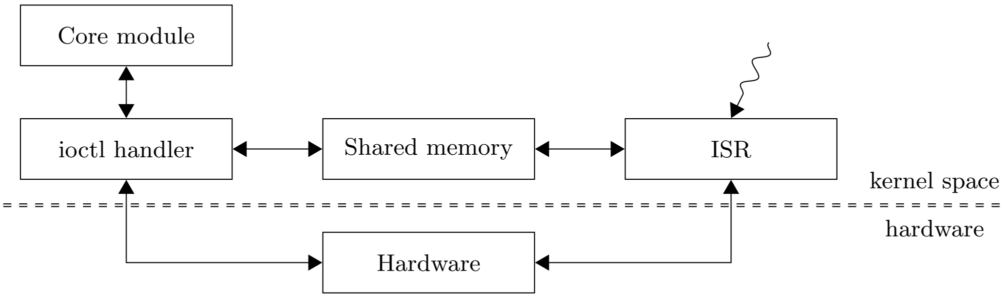


Fig. 2. High-level structure of the I^{2}C Linux bus driver.

While studying the call graph in combination with the source code, we encountered a number of potential issues with regard to the enabling and disabling of interrupts:

- The ioctl handler could sometimes access the shared memory without previously disabling interrupts.
- There was one place in the code where interrupts are disabled twice in a row (without being enabled in the meantime) after which they are enabled twice.
- The usage of kernel wakeup functions was inconsistent and quite unclear.
- False timeouts could possibly be reported, due to interleaving inbetween two subsequent statements.

We considered the possible non-exclusive accesses to the shared memory to be most important. Therefore, we have chosen to focus on this aspect for the mCRL2 and UNO analysis.

4 mCRL2 analysis

The mCRL2 language [17] and toolset [16] allows users to model, validate and automatically verify the behaviour of distributed systems. Systems can be modelled using a process algebra enriched with data and time. Validation can be performed by means of simulation, automated testing and visualisation. Automated verification is supported by dedicated checks for deadlocks or occurrences of specific actions, by model equivalence checks, and by full model checking using properties expressed as temporal logic formulae.

4.1 Modeling

Based on the source code of the I^2C bus driver we have created an mCRL2 model consisting of the following elements:

- A translation of the ioctl handler and the interrupt service routine.
- The environment in which these functions occur:
    - repeated calls to these functions;
    - shared memory;
    - part of the Linux kernel that takes care of enabling/disabling interrupts, and threading.

At the highest level, our model is composed of the components that make up the environment, as depicted in Fig. 3. Here, the components (ioctl handler)* and (ISR)* represent wrappers around the corresponding functions. Their purpose is to call the ioctl handler and the interrupt service routines *repeatedly* with *any possible values* as arguments.

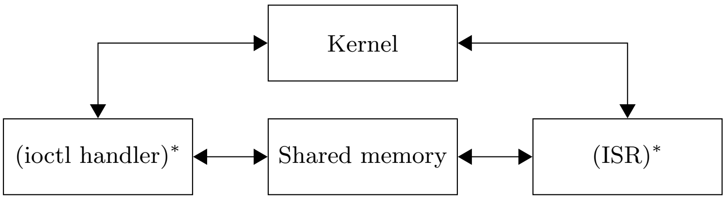


Fig. 3. Structure of the mCRL2 model of the I^2C bus driver.

We will now sketch how these components and their interactions are specified in mCRL2. For details of the mCRL2 constructs used, we refer the reader to [16] for an informal explanation and to [17] for a formal definition. In a nutshell, the primary concept in mCRL2 is the *action*, which represents an elementary activity or communication. Actions can be composed into *process expressions* using process algebraic constructs, such as sequential composition, non-deterministic choice, references to process definitions, parallelism and synchronous communication. Furthermore, mCRL2 contains a functional programming language in which data types can be represented. These data types can be used to form conditional process expressions and parameterised actions and process definitions.

The translation of the ioctl handler and the interrupt service routine from the C source code to mCRL2 was carried out manually, using the following translation scheme:

- The declarations of the ioctl handler, the interrupt service routine and of the local functions they call are modelled as *process definitions.*
- Statements in function bodies are modelled as *process expressions*, as follows:
  - sequential statements are modelled as *sequences* of process expressions;
  - conditions and switch statements are modelled as *conditional process expressions* or as *non-deterministic choice*, depending on their relevance;
  - calls to local functions are modelled as *process references*;
  - calls to kernel functions are modelled as *actions*;
  - entry and exit points of the ioctl handler, the interrupt service routine and local functions are modelled as *actions*;
  - access to shared memory variables is also modelled by *actions.*
- Data types are modelled by their mCRL2 counterpart.

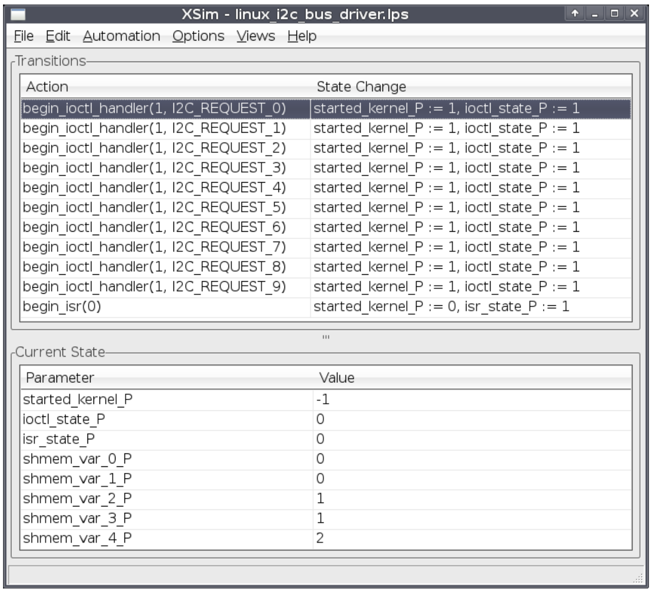
Note that we tried to stay as close to the source code as possible. However, in order to keep model checking within feasible bounds, we abstracted from details irrelevant to the properties we wanted to check, and by restricting the range of data types.

The behaviour of the Linux kernel and the shared memory are modelled as *process definitions* representing the behaviour they are expected to exhibit when interacting with the ioctl handler and the interrupt service routine. The model is then put together by letting the *parallel composition* of the models of all components (ioctl handler, interrupt service routine, kernel and shared memory) *synchronously communicate* on the actions of the individual components.

4.2 Validation

We have validated the model by using simulation to replay scenarios. Fig. 4 shows the simulator tool of the mCRL2 toolset, just after startup. In this tool, the user can inspect the following information:

- Current state (lower part): a combination of all states in which the environment can be in.
- Possible transitions (upper part): the actions that can be performed in the current state, i.e., the calls to kernel functions, function entries and exits, and shared memory accesses.


XSim - linux_i2c_bus_driver.lps
File Edit Automation Options Views Help
Transitions
Current State

Action	State Change
begin_ioctl_handler(1, I2C_REQUEST_0)	started_kernel_P := 1, ioctl_state_P := 1
begin_ioctl_handler(1, I2C_REQUEST_1)	started_kernel_P := 1, ioctl_state_P := 1
begin_ioctl_handler(1, I2C_REQUEST_2)	started_kernel_P := 1, ioctl_state_P := 1
begin_ioctl_handler(1, I2C_REQUEST_3)	started_kernel_P := 1, ioctl_state_P := 1
begin_ioctl_handler(1, I2C_REQUEST_4)	started_kernel_P := 1, ioctl_state_P := 1
begin_ioctl_handler(1, I2C_REQUEST_5)	started_kernel_P := 1, ioctl_state_P := 1
begin_ioctl_handler(1, I2C_REQUEST_6)	started_kernel_P := 1, ioctl_state_P := 1
begin_ioctl_handler(1, I2C_REQUEST_7)	started_kernel_P := 1, ioctl_state_P := 1
begin_ioctl_handler(1, I2C_REQUEST_8)	started_kernel_P := 1, ioctl_state_P := 1
begin_ioctl_handler(1, I2C_REQUEST_9)	started_kernel_P := 1, ioctl_state_P := 1
begin_isr(0)	started_kernel_P := 0, isr_state_P := 1

Parameter	Value
started_kernel_P	-1
ioctl_state_P	0
isr_state_P	0
shmem_var_0_P	0
shmem_var_1_P	0
shmem_var_2_P	1
shmem_var_3_P	1
shmem_var_4_P	2

Fig. 4. Simulation of scenarios using the mCRL2 toolset.

4.3 Verification

For the verification of our model we focused on violation of mutual exclusion of shared memory accesses. To this end, the model was extended with an auditing processes that checks for violations of this property, and issues an error action when one is found.

However, while creating this verification model, we encountered another related possible issue. A number of low-level functions that could be called by both the ioctl handler and the interrupt service routine were marked as non reentrant. For this property we have also created an auditing process.

The structure of the interaction of the two auditing processes with the rest of the model is depicted in Fig. 5.

With the verification model, automated verification boils down to exploring the state space while checking occurrences of the error actions. Exploration of all 62.725.197 states and 102.847.475 transitions revealed two types of violations.

The critical section observer for auditing the mutual exclusion property of shared memory accesses triggered more than 100 violations. The structure of a typical error trace of this violation is as follows:

```
begin_ioctl_handler(1, I2C_REQUEST_i)
begin_cs(1)
shm_write(1, shmem_var, val)
begin_isr(0)
error(0, CS_ERROR_ISR)
```

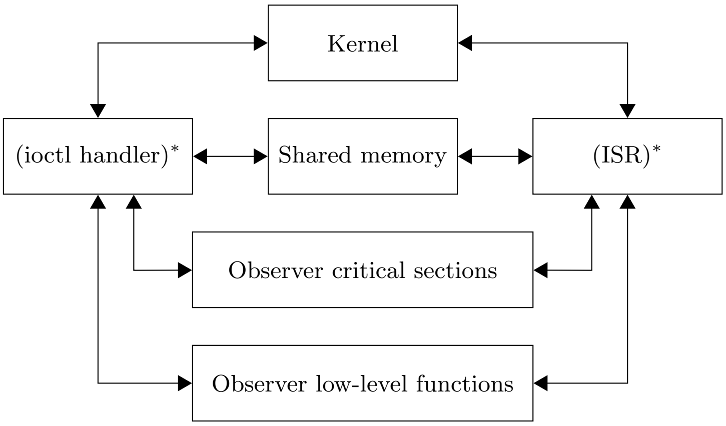


Fig. 5. Structure of the mCRL2 verification model of the I^2C bus driver.

Here, the ioctl handler is started (`begin_ioctl_handler(1, I2C_REQUEST_i)`), enters its critical section (`begin_cs(1)`), and modifies a shared memory variable (`shm_write(1, shmem_var, val)`). At the same time the interrupt service routine is started (`begin_isr(0)`). Since this could potentially modify the same variable, an error is issued. Note that the actions are all parameterised by a number to indicate the component to which they belong.

The other observer for auditing mutual exclusion of low-level function calls revealed one violation. The following error trace was generated:

```
begin_ioctl_handler(1, I2C_LL_REQUEST_i)
begin_I2C_LL_REQUEST_i(1)
begin_isr(0)
shm_read(0, shmem_var_j, val_j)
begin_I2C_LL_REQUEST_k(0)
error(0, LL_REENTRANT_CALL)
```

Here the ioctl handler calls a low-level function (`I2C_LL_REQUEST_i`). This call is interrupted by a call from the interrupt service routine to another low-level function (`I2C_LL_REQUEST_k`).

4.4 Resolving the violations

Single ioctl thread We have identified the causes of the above mentioned violations. They all had to do with misplaced or absent calls to functions that disable and enable interrupts. We fixed this by making 23 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, i.e., no mutual exclusion access violations have been found. The state space of the fixed model comprises 24.875.779 states and 34.803.025 transitions, which is less than for the previous model since some parallel behaviour has been blocked.

Multiple ioctl threads Both the original and the fixed model have only treated one thread making calls to the ioctl handler. It is also possible to analyse the driver in a multi-threaded setting. For this purpose, we have adapted the fixed model by running two instances of the process that represents the ioctl handler. In addition to interrupt disabling, in this model also spin locks are used to establish mutual exclusion between these two processes (threads). Verification of all 163.258.596 states and 231.643.626 transitions revealed no further errors.

On three instances, verification of the correctness using the techniques implemented in the mCRL2 toolset turned out to be prohibitively large to fully explore. To resolve this, we have employed symbolic techniques as implemented in the LTSmin toolset [2]. As for the two instances, this revealed no further errors.

5 Static Analysis Results

5.1 Static Analysis via Model Checking and UNO

The idea of static analysis by model checking, in [14] also called *syntactic* model checking, can be traced back to [31]. Instead of checking the property on the state space of the system, this approach works by exploring control flow and function call graphs of the programs. In this way one avoids the problem of state space explosion, which is the usual bottleneck in practical applications of model checking. The basic principle is the same as in standard model checking: the negation of the property is expressed as an automaton which accepts the erroneous traces.

UNO [19] is a tool for source code analysis. Besides the standard static analysis checks for uninitialized variables, null pointer references and out of bound array indices, UNO can perform some advanced checking for user defined properties (more information can be found on the tool's web page [34]). Compared to model checking, the advantage of UNO is that it works directly on the C source code and therefore it does not require an explicit model. As added value to the project we decided to try to recover all possible errors that were discovered with mCRL2 and which were described in the previous sections. In what follows we use advanced analysis via user defined properties to check for dynamic behavioural defects, like race conditions and unsafe function calls.

5.2 Analysis of the I^{2}C bus driver

Instrumentation of the original C code. Since UNO cannot deal with assembler code and C extensions that are not according to the ANSI standard, the original C source code had to be modified such that those problems were avoided. Thus, we first preprocessed the original C source code with our in-house parser (which is part of the SQuAVisiT toolset) to eliminate some of the non-ANSI C construct `typeof`. After that we had to manually eliminate the remaining occurrences of `typeof`, add some `typedef`'s for the unknown types, and also remove the assembler code. All these changes did not affect the validity of the properties that we wanted to check.

Analysis. UNO was applied to the instrumented code and it was able to reproduce the possible defects that were discovered with mCRL2: the errors of accessing shared memory without previously disabling interrupts and unsafe function calls. As mentioned above, to obtain the errors we had to use UNO with user defined properties, which have to be encoded as property automata. The automaton monitors the traversal of the control flow graphs of the C functions. When presented with the corresponding property automaton, UNO produces an error trace, in case a violation of the property is found.

To capture references to shared memory with enabled interrupts we used the monitor (property automaton) given in Fig. 6. The monitor has two states represented with the variable `uno_state`. State 0 denotes that interrupts are enabled, and 1 that interrupts are disabled. In the C source code that is being checked the property automaton monitors for each function the traversal of all possible paths in its control flow graph (CFG). When along some of the paths of the CFG the name of the shared memory variable `shmem_var` is seen and the monitor is in state 0, an error message is issued. Each node of the CFG is labelled with symbol names extracted from the C program. Also, the node is marked with tags that describe how the symbol name is used, e.g., declared, invoked as a function, evaluated as variable, etc. Function `select` is a predefined query primitive that tries to match the variable name `shmem_var` with the symbol names in the CFG

```
void uno_check(void)
{
  if (uno_state == 0) { //interrupts enabled
     if (select("shmem_var", USE|DEF|REF0, NONE)) //shared memory access
       error("Shared memory reference with enabled interrupts");
     if (select("int_disable", FCALL, NONE)) //interrupts disabled
       uno_state = 1;
  }
  if (uno_state == 1){ //interrupts disabled
     if (select("int_enable", FCALL, NONE)) //interrupts enabled
       uno_state = 0;
  }
}
```

Fig. 6. UNO monitor: checking for shared memory access with interrupts enabled.

nodes. Besides the string corresponding to the variable name, `select` has two other parameters corresponding to the set of tags that must be attached to the symbol name and a set of tags that are forbidden, respectively. For example, tag `DEF` means that `shmem_var` is assigned a value in this occurrence and `NONE` means an empty set of tags, i.e., no tags are forbidden. The switches between states 0 and 1 happen when function calls to `int_disable` and `int_enable`, respectively, are encountered. In a similar way `select` matches the function names while using the tag `FCALL` to ensure that the concrete occurrence of the matched substring corresponds to a function call.

An example of UNO output is the trace given in Fig. 7. It indicates an assignment to the shared memory variable `shmem_var` without previously disabling interrupts. The very first line of the output contains the name of the tool, the number of the error trace, and the name of the function in which the error is found. The next line contains the error message printed by the monitor with the name of the matched symbol, in our case `shmem_var`. The remaining lines of the output are of the following form:

```
uno: 46: i2c_ioctl_handler()
'Shared memory reference with enabled interrupts' [shmem_var]
1: i2c_bus_driver_instr.c:7943:
<i2c_ioctl_handler(struct i2c_adapter *adapter,unsigned int cmd,
unsigned long arg)>;
2: i2c_bus_driver_instr.c:7947:
<i2c_phlm_bus_t *shmem_var=(i2c_phlm_bus_t *)adapter->algo_data;>;
3: i2c_bus_driver_instr.c:7948:
<int ret=0;>;
4: i2c_bus_driver_instr.c:7992:
<unsigned int timeout=(unsigned int )arg;>;
5: i2c_bus_driver_instr.c:7993:
<(timeout>=500)> == <_true_>;
6:  i2c_bus_driver_instr.c:7994:
<shmem_var->slv_timeout=((timeout*HZ)/1000)>;
```

Fig. 7. UNO error trace: reference to shared memory with enabled interrupts.

```
number: file name: program line number:
<program line text>;
```

The assignment to the member `slv_timeout` of the variable `shmem_var` takes place on the last line (labelled by 6). The line labelled by 1 contains the header, i.e., the entry point, of the function `i2c_ioctl_handler` in which the erroneous execution happens. None of the statements in the lines labelled by 1 to 6 contain a call to the interrupt disabling function `int_disable`, so one can conclude that the assignment on the last line is unprotected.

The error trace given above was obtained with the following command line:

```
uno_local -prop shmem.prop -allerr -fullpaths i2c_bus_driver_instr.c
```

where the call to `uno_local` means that only local properties are checked, i.e., intraprocedural analysis is performed, `i2c_bus_driver_instr.c` is the name of the C file which is checked, the property file is `shmem.prop` which is denoted by the switch `prop`, and the options `allerr` and `fullpaths`, respectively, instruct UNO to report all erroneous traces and that the traces should be fullpaths from the entry point of the function to the erroneous statement.

UNO produced 41 error traces. For 20 of them it was clear that they contained access to the shared memory without disabling interrupts. The other 21 traces were from auxiliary static functions, i.e., not from the critical functions, like `i2c_ioctl_handler`, and therefore an additional check was needed to establish whether they were valid or not.

Another type of possible errors that can be discovered in a similar way is a call to a nonreentrant low-level function with enabled interrupts. A property automaton analogous to the above can be used for this check. The essential difference is that the shared memory reference name `shmem_var` is replaced with the name of the function for which we try to detect unprotected calls.

6 Conclusions

In this paper we presented an analysis of the dynamics of an I^{2}C bus driver by means of model checking and static analysis. We were able to find possible non-trivial defects that were later confirmed by the developers. Furthermore, we have provided a verified fix for the found defects.

The time spent on this project is roughly 300 man hour. Most of it has been put into reverse engineering the structure and behaviour of the code, and the mCRL2 analysis.

6.1 Evaluating the two techniques

The results of the automated analysis using model checking with mCRL2 and advanced static analysis with UNO were quite encouraging. Originally, we intended to carry out the analysis of the driver only by using mCRL2. The use of UNO was considered more as an added value and a feasibility study about the usefulness of the advanced static analysis. However, since we were able to achieve similar results, we could compare the two approaches.

Comparing model checking and static analysis When comparing model checking and static analysis, it is important to note the differences in the counterexamples that are produced by both techniques. A model checking counterexample shows a precise path in the model, representing an interleaving of several parallel components. On the other hand, a static analysis counterexample shows a *projection* of such a path onto one component. So in general, model checking produces more precise counterexamples, but they correspond to the model, not to the source code.

In this project, the models have been extracted from the source code. This can be quite useful for checking existing source code. Although some general domain knowledge and expertise about the concrete instance of the problem will always be required, the overhead needed to integrate those in the verification process decreases with each new application of a similar kind, such as other device drivers.

We gained more confidence in the idea that the degree of parallelism in the system and issues at hand are an important factor in deciding whether model checking or advanced static analysis is the most effective choice. We think that the main reason why we could effectively employ advanced static analysis to find concurrency issues in this case study, is because the number of parallel components involved in the properties we wanted to check was rather low. Potential concurrency issues could be identified manually, and translated directly to an UNO property automaton. However, when the number of parallel components increases, and when issues are caused by the interplay of multiple components, finding these potential issues may become much harder, if not impossible.

Combining model checking and static analysis Instead of choosing between model checking and static analysis, we can also use them in tandem. For instance, one could first use advanced static analysis to find possible defects that are traditionally its domain, like uninitialized variables, null pointers, out of bound arrays, memory leaks, possible security flaws like tainted input, etc. Since no model is required, this can be applied to the whole code base. Besides that, advanced static analysis for dynamic properties, as presented in this paper, can be employed as a kind of light-weight model checking to locate possible problems. Then standard model checking techniques can be employed to weed out spurious counterexamples and possibly discover additional errors. Once the possible defects are located, one can restrict fully-fledged model checking only to the critical modules in the code base. In this way one can avoid producing a model of the whole system, which could be time and resource consuming, and which, together with state space explosion, is usually the main obstacle in practical applications of model checking.

Another way to combine the two techniques could be to use static analysis to generate counterexamples as sequences of statements of the source code. One of the classical problems in model checking is how to map the counterexamples of the abstract model into the concrete code. By teaming up model checking and static analysis in such a semi-automated manner we obtain a much easier way to generate the counterexamples. We are not aware of these combined frameworks being used before in the literature. It is an interesting research challenge to couple these two techniques even tighter in the future.

6.2 Related Work

The idea to perform static analysis via model checking, as we use it in this paper, was considered for the first time in [30,31]. This has been later developed and served as a framework for several tools. Probably the best known in that direction is the work of Engler et al.(e.g. [13]). In [19], which introduces the static analysis tool UNO, case studies of verification of Linux device drivers with the tool are mentioned, although no details of the results are presented. This work uses only static analysis without traditional model checking. An important feature of the tool is that it allows user defined properties which can be used to check dynamic (behavioural) characteristics of the system. The ideas and concepts of UNO are further developed in the Orion tool [10], which puts emphasis on eliminating spurious counterexamples. For the time being the tool does not feature user defined properties and as such it cannot be used for the type of analysis that we have in this paper. Goanna [14] is a C/C++ source code analyzer which employs the model checker NuSMV [6]. The model checking is used only in the context of advanced static analysis and not in a traditional way as a standalone technique. They use as a case study the Open Secure Sockets Layer package. RacerX [12] is another static tool that can detect deadlocks and race conditions. To this end it computes sets of locks that are used in the program. One of the strengths of RacerX is that the locks are inferred using statistical methods. In this way, at least in theory, the user is releaved of the task to indicate the locks in the programs. In [12] several case studies are reported and among them are checks of two versions of the Linux kernel, with confirmed errors found. A predicate abstraction approach to concurrent Linux device drivers analysis is presented in [36]. The tool DDVerify described there can generate a driver harness from a driver source code, which can further be analysed by a pre-/post condition checking tool SatAbs. There are other commercial tools, like Coverity [8], KlockWork [21], and Sonnar [32], and academic tools, like Calysto [1] and MyGcc [35], that can use advanced static analysis methods. For a more complete overview see [33].

It is interesting that Approver [18], probably the very first ever tool for automated formal verification, written in the end of the 70s by J. Hajek at the Technische Universiteit Eindhoven, used as input Algol programs instead of some modeling language. Later this approach was abandoned and the verification tools used as a rule some modeling language. Model checking directly source code ("model checking without a model") was pioneered by Godefroid with his tool VeriSoft (e.g. [15]). Since then model checking of source code of languages like C, C++, and Java became quite a fashionable trend in the verification community. The most influential work was the one by Ball and Rajamani that resulted with the SLAM model checker [3] which became an important part of Microsoft's Static Driver Verifier. Another tool derived from this line later is BLAST [4]. This tool was applied in [25] to check Linux kernel code, where also some of its weaknesses were indicated (see [25] also for a comprehensive list of other applications of BLAST). CMBC [7] is a tool for bounded model checking of C code which uses SAT solvers. There exist also tools based on other techniques, like automated theorem proving (Eau Claire [5]) and abstract interpretation (PolySpace [27], Astrée [9]) which also work directly with source code.

6.3 Future Work

In general, although there is ample room for improvement, one can conclude that model checking and static analysis is sufficiently mature to cope with industrial size problems and can contribute to the improvement of the software products. This opens promising perspectives for future projects of this type.

Depending on the future collaboration with the client, we intend to extend our analysis on the other layers of the protocol stack. Considering that some defects remained unresolved, it could be interesting to see if they are caused by erroneous behavioural patterns which do not fall in the categories that we considered in this work. Then, hopefully one could generalize these patterns for later use with other software systems. Another natural avenue for future work would be to apply the combination of model checking and static analysis to other types of software, which again can be an inspiration for improvements in both techniques as well as their combination.

The most important thing our model checking part can benefit from is automating the extraction of models from source code. This consists of two tasks: translating the source code to a model, and abstraction from irrelevant details. Automated translation of the source code to a model would bring us most of the benefits, since this translation can be specified for a large class of programs, while abstraction needs to take place on a case-by-case basis. A good starting point in that direction could be [20] which presents an attempt to combine these two aspects.

Acknowledgements We thank Serguei Roubtsov for his help with the instrumentation of the source code, and Mark van den Brand and Harold Weffers for their feedback on previous versions of the text.

References

1. D. Babić, A.J. Hu, *Structural Abstraction of Software Verification Conditions*, Computer Aided Verification (CAV '07), LNCS 4590, pp. 371–383, Springer, 2007.
2. S. Blom, and J. van de Pol. *Symbolic Reachability for Process Algebras with Recursive Data Types*, Proc. ICTAC 2008, LNCS 5160, pp.81–95. Springer, 2008.
3. T. Ball, S.K. Rajamani, *The SLAM Toolkit*, Computer Aided Verification (CAV '01), pp. 260-264, Springer, 2001.
4. D. Beyer, T.A. Henzinger, R. Jhala, R. Majumdar, *The Software Model Checker BLAST*, Int. J. Softw. Tools Technol. Transfer, vol. 9, pp. 505-525, Springer, 2007.
5. B. Chess, *Improving Computer Security Using Extended Static Checking*, IEEE Symposium on Security and Privacy SP '02, Wahsington DC, USA, p. 160, IEEE Computer Society, 2002.
6. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, *NuSMV Version 2: An Open Source Tool for Symbolic Model Checking*, Computer-Aided Verification (CAV '02), LNCS 2404, pp. 359–364, Springer, 2002.

7. E. Clarke, D. Kroening, F. Lerda, *A Tool for Checking ANSI-C Programs*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), LNCS 2988, pp. 168-176, Springer, 2004.
8. http://www.coverity.com
9. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Moniiaux, X. Rival, *Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software*, The Essence of Computation, LNCS 2566, pp. 85-108, 2002.
10. D. Dams, K.S. Namjoshi, *Orion: High-precision methods for static error anaalysis of C and C++ programs*, Formal Methods for Components and Objects, 4th Int'l Symposium FEMCO 2005, LNCS 4111, pp. 138–160, 2006.
11. E.A. Emerson, E.M. Clarke, *Characterizing Correctness Properties of Parallel Programs Using Fixpoints*, Automata, Languages and Programming, pp. 169-181, LNCS 85, Springer, 1980.
12. D. Engler, K. Ashcraft, *RacerX: Effective, Static Detection of Race Conditions and Deadlocks*, Proc. of the 19th ACM Symposium on Operating Systems Principles SOSP 2003, pp. 237–252, ACM, 2003.
13. D. Engler, B. Chelf, A. Chou, S. Hallem, *Checking System Rules Using System-Specific, Programmer Written Compiler Extensions*, Proc. Symposium on Operating Systems Design and Implementation, San Diego, California, 2000.
14. A. Fehnker, R. Huuck, P. Jayet, F. Lussenburg, F. Rauch, *Model Checking Software at Compile Time*, IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering TASE '07, pp.45–56, IEEE, 2007.
15. P. Godefroid, B.Hanmer, L. Hagadeesan, *Model Checking Without a Model: An Analysis of Heart-Beat Monitor of a Telephone Switch using VeriSoft*, ISSTA '98 (ACM SIGSOFT Int'l. Symposium on Software Testing and Analysis), 1998.
16. J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. *Analysis of distributed systems with mCRL2*, In: M. Alexander, W. Gardner, editors, Process Algebra for Parallel and Distributed Processing, pp.99–128. Chapman and Hall, 2008.
17. J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. *The formal specification language mCRL2*, In: E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, Methods for Modelling Software Systems (MMOSS), nr 06351 in Dagstuhl Seminar Proceedings, Internationales Begegnungsund Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
18. J. Hajek. *Automatically verified data transfer protocols*, Proc. 4th Int'l Computer Communications Conference, pp. 749–756, 1978.
19. G.J. Holzmann, *Static Source Code Checking for User-Defined Properties*, Proc. World Conference on Integrated Design & Process Technology (IDPT), 2002.
20. G.J. Holzmann, M.H. Smith, *A Practical Method for the Verification of Event-Driven Software*, Proc. Int'l. Conf. on Software Eng. (ICSE 99), pp.597–607, 1999.
21. http://www.klockwork.com
22. G. Kroah-Hartman, *I2C Drivers, Part I*, Linux Journal, December 2003.
23. G. Kroah-Hartman, *I2C Drivers, Part II*, Linux Journal, February 2004.
24. T. Kropf, *Software Bugs Seen from an Industrial Perspective or Can Formal Methods Help on Automotive Software Development?*, Computer Aided Verification (CAV '07), LNCS 4590, p.3, Springer, 2007.
25. J.T. Mühlberg, G. Lüttgen, *BLASTing Linux Code*, FMICS and PDMC 2006, LNCS 4346, pp. 211–226, Springer, 2007.
26. F. Nielson, H.R. Nielson, C.L. Hankin, *Principles of Program Analysis*, Springer, 1999.

27. http://www.polysapce.com/cpp.htm
28. S. Roubtsov, A. Telea, D. Holten *SQuAVisiT: A Software Quality Assessment and Visualisation Toolset*, In: Seventh IEEE Int'l Working Conference on Source Code Analysis and Manipulation (SCAM 2007). pp. 155–166, IEEE, 2007.
29. J.P. Quelle, J. Sifakis, *Specification and Verification of Concurrent Systems in CESAR*, 5th Int'l Symposium on Programming, pp. 337–350, 1981.
30. D.A. Schmidt, *Data Flow Analysis is Model Checking of Abstract Interpretations*, ACM SIGPLAN-SIGACT Symposium on Principle of Programming Languages (POPL '98), New York, NY, USA, ACM Press, 1998.
31. D.A. Schmidt, B. Steffen, *Program Analysis as Model Checking of Abstract Interpretations*, Intl. Symposium on Static Analysis (SAS '98), pp 351–380, Springer, 1998.
32. http://www.grammatech.com/products/codesonar/overview.html
33. http://spinroot.com/static
34. http://spinroot.com/uno
35. N. Volanschi, *A portable compiler-integrated approach to permanent checking*, Journal Automated Software Engineering, 15 (1), Springer, 2008.
36. N. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher, *Model Checking Concurrent Linux Device Drivers*, Proc. ACE'07, November 1-9, 2007, Atlanta, Georgia, USA.