Master's Thesis

Formal Verification of Linux Device Drivers

Thomas Witkowski

May 2007

Supervisors:
Prof. Dr. Daniel Kroening and Nicolas Blanc

Contents

1. Introduction **7**
- 1.1. Related Work . . . 8
- 1.2. Outline . . . 10

2. Device Drivers **13**
- 2.1. Introduction . . . 13
  - 2.1.1. Types of Devices Drivers . . . 14
- 2.2. Linux Kernel API . . . 15
  - 2.2.1. Mutual Exclusion . . . 15
  - 2.2.2. Wait Queues . . . 17
  - 2.2.3. Tasklets and Work Queues . . . 18
  - 2.2.4. Interrupts . . . 18
  - 2.2.5. I/O Ports and I/O Memory . . . 20
  - 2.2.6. Memory Usage . . . 21
- 2.3. General Structure of a Device Driver's Source Code . . . 21
  - 2.3.1. Character Device Drivers . . . 21
  - 2.3.2. Block Device Drivers . . . 22
- 2.4. Bugs in Device Drivers . . . 23

3. Theoretical Background **27**
- 3.1. Calculus of Communicating Systems . . . 27
  - 3.1.1. CCS with Priorities . . . 28
  - 3.1.2. Sorts . . . 31
  - 3.1.3. Value Passing Extension . . . 31
- 3.2. LTL . . . 33

4. Operational Semantics **35**
- 4.1. The Programming Language $\mathcal{C}_0$ . . . 35
- 4.2. Operational Semantics of $\mathcal{C}_0$ . . . 37
  - 4.2.1. Sequential Composition . . . 37
  - 4.2.2. Variables . . . 37
  - 4.2.3. Commands and Expressions . . . 38
  - 4.2.4. Translating $\mathcal{C}_0$ Programs . . . 39
- 4.3. CCS based Semantics of the Linux Kernel API . . . 39
  - 4.3.1. Semaphores and Spinlocks . . . 40
  - 4.3.2. Wait Queues . . . 42

4.3.3. Tasklets . . . 44
4.3.4. Work Queues . . . 45
4.3.5. Interrupts . . . 46
4.3.6. Translating Device Drivers . . . 48
4.3.7. Execution Models for Device Drivers . . . 48
4.3.8. Example . . . 50
4.4. Operational Model for the Linux Kernel API . . . 52
4.4.1. Semaphores and Spinlocks . . . 53
4.4.2. Wait Queues . . . 54
4.4.3. Tasklets . . . 54
4.4.4. Work Queues . . . 55
4.4.5. Interrupts . . . 56
4.4.6. I/O Ports and Memory Usage . . . 57
4.4.7. Sequential Execution Model . . . 57
4.5. Proof of Over-Approximation . . . 57

5. Model Checking with SATABS **61**
5.1. CEGAR-Framework . . . 61
5.1.1. Example . . . 63
5.2. Using SATABS . . . 65

6. Driver Verification with DDVerify **67**
6.1. Implementation . . . 67
6.2. Usage . . . 68
6.3. Case Studies . . . 70

7. Discussion **73**
7.1. Future Work . . . 73
7.2. Acknowledgments . . . 74

A. Supported Linux Kernel API Functions **77**

B. Result of the Translation **81**

List of Figures

2.1. A split view of the kernel 14
2.2. Exemplary code of a character driver 22
2.3. Statistics about errors in Linux device drivers 24

3.1. Initial action sets 29
3.2. Operational semantics for CCS with priorities 30
3.3. Definition of sort $\mathcal{L}(P)$ of a process expression P 31
3.4. Definition of the translation function $[\![\cdot]\!]$ from the value passing CCS to the classical calculus 32
3.5. Labeling function for CCS processes 34

4.1. Syntax of $\mathcal{C}_0$ 36
4.2. Transitions of awaking processes from a wait queue 42
4.3. $\mathcal{C}_0$ program of a simple device driver 51
4.4. The operational semantics for semaphores 53
4.5. The operational semantics for wait queues 54
4.6. The operational semantics for tasklets 55
4.7. The program $ExecTaskletFunction$ 55
4.8. The operational semantics for work queues 56
4.9. The operational semantics for interrupts 56
4.10. The program $ExecInterruptFunction$ 56
4.11. The program $SequentialDriverExec$ 57
4.12. General idea to prove the over-approximation 58

5.1. The counterexample guided abstraction refinement framework 63
5.2. Example program P and three boolean programs (B_1, B_2 and B_3) that abstract P 64

6.1. Overview of DDVerify 67
6.2. Parameters of DDVerify 69
6.3. Statistics of the case studies 70
6.4. Spinlock bug in device driver `char/ds1286.c` 71
6.5. I/O port bug in device driver `char/watchdog/machzwd.c` 72

1. Introduction

Reliability of complex hardware and software systems became increasingly important over the last decades. Most parts of our infrastructure, e.g. traffic systems or telecommunication, make use of highly integrated and connected software components. A single error in the source code and an associated malfunction of the system can cause high costs and might even endanger the health of people. Therefore, verification of software has been a central problem of computer science for many years.

There are several different techniques for software verification, such as type systems, programming logics, formal semantics, static analysis and model checking. They differ in their complexity to use and in what they are able to prove. Model checking [18] is a powerful verification method, which can be applied in a full automatic manner. Nonetheless, user intervention is typically required in two places. First, model checking can only be applied to finite models. But in general, software can have an infinite state space. Hence, the user is responsible to provide an abstracted model that is fine enough to capture the important details. To automatize the process of model extraction from source code, much research [4, 13, 19, 21, 22, 29, 31, 39, 40, 41, 49, 61] has been done. Second, a property must be given for which the model is verified. We distinguish between two classes of verification properties: either the property is directly given by the user or the model of the source code is verified for properties which must always hold, e.g. null pointer dereferencing, array bounds or the correct use of an API. A user defined specification can be a formula in a temporal logic, an automaton or just a line of code, which is not allowed to be reached on all execution paths of the program.

In this thesis, we present the tool DDVerify for full automatic verification of Linux device drivers. Device drivers are part of the Linux kernel and are responsible for the communication between hardware and other parts of the kernel and user software, respectively. We have chosen device drivers because of the following reasons:

- They are the biggest part of the Linux kernel. Driver code accounts for about 70% of the code size of the Linux kernel 2.4.1 [17].
- Bugs in device driver code are the main reason for errors in operating systems, also relative to the size of code, as reported in [6, 17].
- A bug in a device driver can crash the whole system because it runs in kernel mode with exclusive access to the hardware, memory and cpu. Hence it is hard to detect and fix a bug in driver code.

For the verification process, we make use of the model checking tool SatAbs [66]. SatAbs can be used to verify ANSI-C programs for both user-defined specifications and general properties, like buffer overflows, pointer safety and division by zero, which must hold in all programs. Because it is based on the CEGAR-framework [19], the user does not need to provide an abstraction. SatAbs creates an abstraction of the model, which is as simple as possible but precise enough to capture all details that are necessary for the verification of the given property.

DDVerify checks Linux device drivers for correct usage of the Linux kernel API. The kernel API provides a large set of concepts for writing device drivers. It does not have a well defined semantics, but is described by means of natural language in several places [9, 51, 28] instead.

1.1. Related Work

The first attempt to formalize a programming language using a process logic is done by Milner in [56]. He uses a simple concurrent programming language, which is very similar to C, to demonstrate how to reason about programs using the *Calculus of Communicating System* (CCS). [67] makes use of a real-time extension of CCS, called RtCCS [68], in order to define a formal semantics for a real-time object-oriented programming language. Similar but more general, [62] introduces a framework for defining semantics of arbitrary concurrent object-based languages. In contrast to these works, [16] does not deal with an exemplary language but formalizes a subset of Java. This work is very similar to our formalization part. It defines a translation for a restricted Java program to a set of CCS processes. The main focus is on the concurrent behaviour of Java-threads.

Cleaveland et al. [24] use CCS, extended with priorities for the transitions, to model and verify distributed systems. As an example, a safety-critical part of a network used in the British Rail's Solid State Interlocking, a system which controls railways signals and points, is translated to CCS and verified using the NCSU Concurrency Workbench [25].

The most closely related work to the practical part of this thesis is the Microsoft SLAM project [4, 5, 6]. SLAM's analysis engine is a tool for checking safety properties of sequential C programs. It is based on predicate abstraction and the CEGAR-framework. The SDV (Static Driver Verifier) tool uses SLAM's analysis engine to check Windows device drivers for correct usage of the kernel API. The API usage rules are encoded in the C-like language SLIC (Specification Language for Interface Checking) [3]. This make is possible to extend the set of rules without changing the source code of the tool. In contrast to our work, SDV has several restrictions: variables are modeled using unbounded integer numbers, bit-wise operations are not supported, pointer arithmetic is ignored and casting operations are not checked.

YASM [39] is a model checker for C programs based on the CEGAR framework. Hence, it is very similar to SLAM and our approach. Its main advantage is

that YASM is not restricted to safety properties but can verify arbitrary properties, which can be expressed as CTL formulas. Therefore, also liveness properties (something will eventually happen) and non-blocking properties (something may always happen) can be expressed.

SATURN (SATisfiability-bases failURe aNalysis) [72, 73] is a framework for static bug detection. It is based on the translation of C code to SAT formulas. A SAT solver is used to check a property, given in the form of a SAT query. The framework has several restrictions. It does not support unions, arrays, pointer arithmetic and function pointers. Furthermore, it is unsound in its analysis of loops because it is not able to construct a finite boolean formula representing a fully unrolled loop. Instead, SATURN has been shown to be applicable on large source code. Using this framework, a Linux lock checker is built [72] to find locking-related bugs. It is applied to the source code of the whole Linux kernel, version 2.6.5. As a result of the analysis, 179 previously unknown bugs were found.

The model checker BLAST (Berkeley Lazy Abstraction Software verification Tool) [42, 43, 58] is used to find errors in the Linux kernel related to memory safety and incorrect locking behaviour. To build a model from C source code, BLAST uses an abstraction algorithm, called "lazy abstraction" [44]. The model checking algorithm checks if a label in the source code is reachable. This label is either introduced by the user or it is the result of an automatic translation of a temporal safety property. An error is found if the model checker is able to find a path such that the label can be reached. The major limitation of BLAST is the absence of support for function pointers, which are widely used in device driver code. The case study [58] shows that BLAST is able to find locking-related bugs, but had problems with checking for memory-safety. In [43], BLAST is used to present the idea of *extreme verification*, in dependence on extreme programming, a widely used method in software engineering. Instead of verifying the final result of the programming process, the model checking algorithm is used to determine if a change in the source code affects the proof of a property in the previous version. This technique is demonstrated on a Windows device driver.

The static analyzer MOPS [69] is used to check six security related properties on the entire Red Hat Linux 9 distribution, containing around 60 million lines of code. MOPS is not sound because it does not analyze function pointers and signals. Nevertheless, MOPS was able to find 108 real exploitable software bugs. It also reported 2225 false positives. More than 150 man-hours were spent to check the error reports.

The CMC model checker [60] follows a different approach. Instead of trying to extract a model from given source code, it executes the program within the model checker. This way, CMC is responsible for scheduling and executing the program in such a fashion to explore as many states as possible. Due to the possibly infinite state space of a program, CMC cannot exhaustively explore it. Nevertheless, CMC has been shown to be applicable in practice. In [59], CMC is used to check the Linux kernel implementation of TCP/IP. It was able to detect four new bugs within it. In [33], this case study is analysed and compared to static analysis of source code.

SPIN [46] is one of the first verification tools based on model checking, which is being applied widely in research and industry. It is not only restricted to software verification but can also be used to verify generic finite models that are specified in PROMELA, the specification language of SPIN. It was used to verify software written in C [48, 49, 37, 36], C++ [13] and Java [31, 41]. Further approaches to verify real source code can be found in the VeriSoft project [38, 15] and the Bandera tool set [29, 40].

Most model checkers for C code are not able to deal with device drivers because driver functions make use of large data structures, which must be initialized appropriately. Our approach defines an operating system model for Linux that creates these data structures and passes them to the driver functions. In [63], a general approach is presented to construct data environments by initializing all pointers without an operating system model.

Besides model checking, there also are other approaches of static code analysis. UNO [74] is a tool for static analysis of ANSI-C code. It checks a program for using uninitialized variables, `NULL` pointer dereferencing and out-of-bound array indexing. Furthermore, UNO can check for user defined properties, which are written in ANSI-C. In [47], UNO is used to check parts of the Linux kernel 2.3.9 for correct locking usage. PREfix [12] is a static analyzer for C and C++ source code, based on a simulation technique finding specific error-causing conditions. Due to its memory model, PREfix is also able to find memory related errors. In [30], the partial program verification tool ESP is presented. To verify a temporal safety property, ESP restricts the path-sensitive analysis to paths which are relevant for this property. Since the runtime of the algorithm is polynomial in time and space, it is also applicable to large software.

In [34], a static code analyser is presented which is able to extract programmer's belief from the source code. The only user inputs are templates of the form "does lock l protect variable v?". The analyser conducts a statistical analysis of the source code and instantiates the templates. Contradictions to the instantiated templates are reported to be errors in the code. This approach was applied on the Linux kernel 2.4 and was able to find 14 errors.

1.2. Outline

Chapter 2 describes the basic concepts of Linux device drivers. We introduce all parts of the Linux kernel API, for which we give a formalization later and which are supported by our tool DDVerify. We then show the general structure of the source code of device drivers and explain the most common bugs appearing in them. In Chapter 3, we introduce the theoretical concepts we use for the formalization. We give a brief introduction to the Calculus of Communicating Systems (CCS), which also gets extended with priorities, and the Linear-time Temporal Logic (LTL). The Linux kernel API is formalized in Chapter 4. Here, we give an operational semantics for a subset of C and extend this semantics to the commands provided by the kernel

API. In Chapter 5, we introduce model checking with the CEGAR-framework, which is the verification technology SATABS is based on. In Chapter 6, DDVERIFY is introduced. We explain the way it is implemented and how it collaborates with SATABS to verify device drivers. That chapter concludes with an overview over the results of the case studies we made with DDVERIFY about the device drivers of the Linux kernel 2.6.19. Finally, we summarize and discuss the results of this thesis. We also discuss ideas for future works.

2. Device Drivers

In this chapter, we give a brief introduction to device drivers. More information can be found in: [28, 9, 51].

The Linux kernel API provides numerous concepts and functions for Linux device driver programming. We have chosen a subset of this API, which is formalized in the next chapter and which is supported by DDVerify. This subset contains the most important concepts, which are widely used in device drivers and most critical with respect to bug occurrence. After a general introduction to device drivers, Section 2.2 presents these parts of the Linux kernel API. In Section 2.3, the general structure of device drivers' source code is explained. The chapter concludes with an overview of the most common bugs in device driver code.

2.1. Introduction

Device drivers are the parts of the Linux kernel which are responsible for the intercommunication between hardware, e.g. the devices, and software (see Figure 2.1). They can be seen as a "black box" that makes a device respond to a well-defined internal programming interface. The details of how the device works are completely hidden. If a software wants to perform something on the device, it can make use of standardized function calls that are independent of the specific driver. Then, mapping those calls to device-specific operations that work on real hardware is the very role of the device driver.

Device drivers do not only interact with user space programs, but also with the other parts of the kernel. Although the distinction between the different subsystems of the kernel cannot always be clearly made, the kernel can be split into the following components:

- Process management: The process is one of the fundamental abstractions in Linux. A process is a program in execution together with a set of resources such as open files, signals, internal kernel data and global variables. The kernel's process management allows to create and destroy processes and communication among different processes. In addition, the scheduler, which controls how processes share the CPU, is part of process management.

- Memory management: Similar to user applications, programs running in kernel space require a possibility to allocate memory. The memory management subsystem provides a rich set of methods, ranging from the simple `kmalloc`/`kfree` pair to much more complex functionalities.

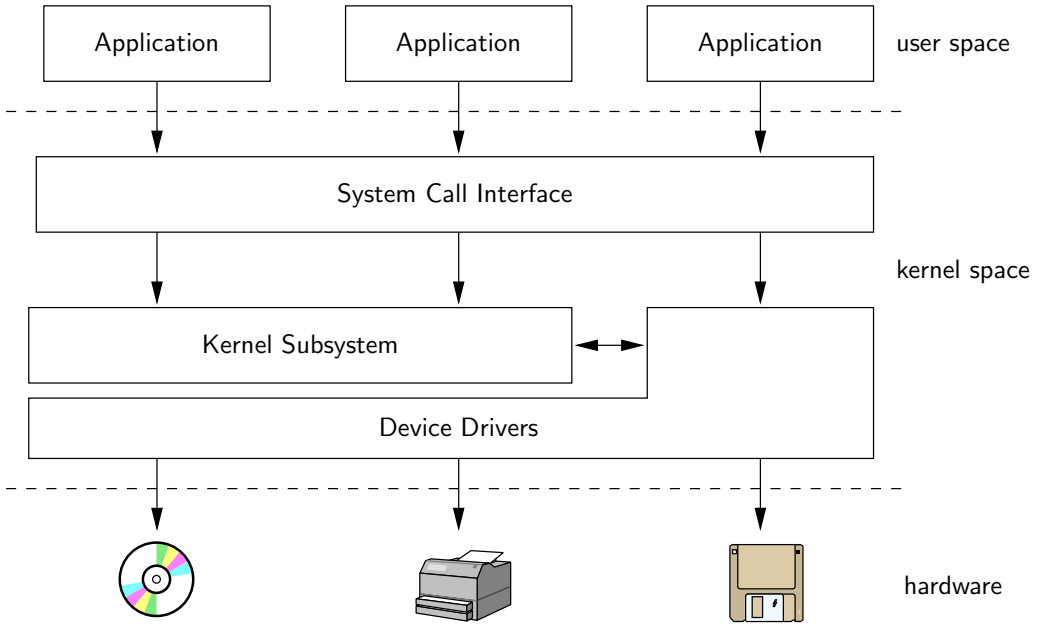

Figure 2.1.: A split view of the kernel

- Virtual filesystem: This subsystem of the kernel implements the filesystem-related interfaces provided to user space programs. In Linux, almost everything can be treated as a file.

- Networking: It is responsible for collecting, identifying, and dispatching incoming packets. It is in charge of delivering data packets across programs and network interfaces. Additionally, all routing and address resolution issues are implemented within the kernel.

We have used the terms *kernel space* and *user space* before. Both are different modes of execution. Device drivers always run in kernel space. That means they have a protected memory space and full access to the hardware. Conversely, user applications are executed in user space. They see a subset of the machine's available resources and are unable to call certain system functions, access hardware directly, or misbehave in other ways.

2.1.1. Types of Devices Drivers

Linux distinguishes between three types of devices: *character devices* (or *char devices* for short), *block devices* and *network devices*. For each of these device classes, Linux provides a different driver programming interface. A character device is accessed as a stream of sequential data (like a file). Examples are keyboards, mice and the serial port. A driver for a character device usually implements at least the

open, close, read and write system calls. If a device is not accessed sequentially but rather randomly, it is called a block device. The most common block devices are hard disks, floppies, CD-ROMs and flash memories. Internally, a block device driver works with *blocks* of data, usually having a size of 4096 bytes. But in contrast to most Unix systems, Linux allows user software to perform I/O operations on an arbitrary number of bytes. Hence, block and character devices differ only in the way their data is managed internally in the kernel. Network device drivers implement the functionality of sending and receiving data packets of network interfaces. The most important difference between the previous device drivers and network drivers is that network drivers receive packets asynchronously from the outside, whereas the others operate only on requests from the kernel.

This is not the only way how devices and device drivers could be classified. One could also imagine to distinguish between USB devices, SCSI devices, PCI devices, and so on. In such a classification, a hard disk connected over USB would be classified as a USB device. In contrast to this, a device driver for a USB hard disk drive is implemented in Linux as a block device that works with the USB subsystem.

2.2. Linux Kernel API

In this section we give an introduction to the parts of the Linux kernel API which are formalized in Section 4.3 and which are supported by DDVerify. We explicitly mention what is modeled and which parts or which behaviour we do not care about. All supported API functions and their correct types are listed in Appendix A.

2.2.1. Mutual Exclusion

When writing device drivers, it must be considered that the code runs in parallel to itself most of the time. This may have two reasons: some parts of the code, like timers, tasklets or interrupt handlers are concurrently executed to the "main code" of the device driver. Second, more than one user space program can make use of a device. But not every part of the code should be executed by two threads at the same time. Consider the following example, which is taken from [28]:

```
if (!data[pos]) {
   data[pos] = kmalloc(...);
   if (!data[pos]) {
      goto out;
   }
}
```

This piece of code checks whether the required data is allocated or not. In the letter case, the memory will be allocated using `kmalloc`. Suppose that two processes A and B are independently attempting to write to the same position within the device. Each process reaches the `if` test in the first line of the fragment above at the same

time. If the pointer in question is `NULL`, each process will decide to allocate memory, and each will assign the resulting pointer to `data[pos]`. Since both processes are assigning to the same location, clearly only one of the assignments will prevail. What will happen, is that the process that completes the assignment second will "win". If process A assigns first, its assignment will be overwritten by process B. At that point, the device driver forgets entirely about the memory that A allocated. The memory is dropped and never returned to the system.

This sequence of events is a demonstration of a *race condition*. Race conditions are a result of uncontrolled access to shared data. When the wrong access pattern happens, something probably undesirable results. In the example above, the result is a memory leak. The parts of the code which manipulate shared data and therefore can be responsible for race conditions are called *critical sections*.

The usual technique for access management to shared resources is called *locking* or *mutual exclusion*, which ensures that only one process can have access to shared data at any time. The Linux kernel API provides several methods to do so, for example the so called *semaphores*, *spinlocks* and *mutexes*[1]. For the sake of simplicity, we can think of a semaphore to be a variable with only two values *locked* and *unlocked*. If a process wants to enter a critical section, a corresponding semaphore must be acquired for this purpose. If the semaphore is locked at this moment, the process must wait until the semaphore is unlocked by the process that owns it. Using semaphores, the source code of the previous example would look as follows:

```
lock(sem);
if (!data[pos]) {
   data[pos] = kmalloc(...);
   if (!data[pos]) {
      goto out;
   }
}
unlock(sem);
```

Mutexes [57] were introduced in the Linux kernel 2.6.17. From the point of use, they are equal to semaphores. In contrast to them, the mutex subsystem is faster, has a smaller data structure, has a smaller size of system code, and it provides an advanced debugging system. There are no special read/writer mutexes, which are provided by the semaphore subsystem.

Spinlocks are very similar to semaphores. They differ in how the waiting is done when a process must wait to enter a critical section. If a critical section is guarded with a semaphore, the process can go to sleep. That means, that another process will be executed and the scheduler will choose the waiting process in the future. In contrast to this, spinlocks are implemented with busy waiting. A process, when waiting for a spinlock, will not lose control of the execution.

[1] "Mutex" is also used as a general concept name for mutual exclusion methods. In the Linux kernel 2.6.17, mutexes are introduced as a self-contained subsystem.

We formalize both, semaphores and spinlocks. We omit the formalization of mutexes, because it would not differ from the former. DDVerify supports semaphores, spinlocks and mutexes. Device drivers can make use of arbitrary many locks. They may be initialized either statically or at runtime. Interruptible and noninterruptible sleep is possible. Furthermore, the functions `down_trylock`, `spin_trylock` and `mutex_trylock` are supported. We do not support the reader/writer semaphores and reader/writer spinlocks. Disabling interrupts is also not supported by DDVerify. Therefore, the following functions are supported but have no influence on the behaviour of interrupts: `spin_lock_irqsave`, `spin_lock_irq`, `spin_lock_bh` and the corresponding unlocking functions.

2.2.2. Wait Queues

Putting a process to sleep is an important technique in device driver programming. Assume a scenario where there is a process that wants to read data from a device, but there is no data present at this time. The process can go to sleep. This means, it is removed from the scheduler's run queue and will be woken up from another process when something of interest happens. For example, a writer process can awake the reader process to notify it that new data was written to the device and is now available to be read.

Wait queues are one of the available concepts for sleeping. A wait queue is a simple list of processes waiting for an event to occur. It is represented by a data structure of type `wait_queue_head_t`. After the initialization of a wait queue variable, a process can go to sleep on this wait queue by invoking one of the `wait_event*` macros. The following two macros are formalized and supported by DDVerify:

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
```

In both cases, `queue` is the wait queue data structure to use. The `condition` is an arbitrary boolean expression. A process goes to sleep only if this expression evaluates to true. In this case, the process deletes itself from the run queue of the scheduler and calls `schedule`. When using `wait_event`, the process is put into an uninterruptible sleep. In contrast to this, the sleeping of `wait_event_interruptible` can be interrupted by signals. If the waiting was interrupted, this function returns a nonzero integer.

Processes waiting for an event are awaken using one of the following functions:

```
wake_up(queue)
wake_up_interruptible(queue)
```

If a process is put to sleep with `wait_event`, it can be only awaken with the function `wake_up`. Correspondingly, if the function `wait_event_interruptible` is used to wait for an event, the function `wake_up_interruptible` must be used to wake up the process. When one of these functions is called, the waiting condition is checked again. If it is false, the process is put to sleep again. Nevertheless, an awakened

process cannot make any assumption about this condition. After it has successfully been awakened, another process could be scheduled which changes the condition. Therefore in most cases, putting a process to sleep is implemented as a while loop. Whenever the process is awakened and the condition does not hold, it is put to sleep again.

2.2.3. Tasklets and Work Queues

There are many situations where a process needs to defer work to happen later. Interrupt handlers, which are explained in the next section, must terminate very fast. Therefore, they only do the important work. Most of the data management operations are delayed to a later time. The Linux kernel API provides several concepts for deferred work: kernel timers, softirqs, tasklets and work queues. The latter two are used in most situations. Therefore, we formalize them and they are supported by DDVerify. Softirqs are used in very special situations only, such as, when the scheduling has to be very fast. In contrast to the other mechanisms, kernel timers allow to schedule a task until at least a specific time span has elapsed.

Tasklets, which have nothing to do with tasks, are represented by the data structure `tasklet_struct`. A variable of this type is initialized together with a so called *handler function* of the tasklet. To schedule the tasklet, `tasklet_schedule` is used. The tasklet's handler function may be executed immediately if the system is not under heavy load, but never later than the next timer tick. A tasklet can be executed in parallel with other tasklets, but is strictly serialized with respect to itself. DDVerify also supports the possibility to disable and enable tasklets, but it does not care about the higher prioritized tasklets, which are scheduled with `tasklet_hi_schedule`.

Work queues are very similar to tasklets. The major difference is that tasklets run in interrupt context, with the result that tasklet code is not allowed to sleep. Instead, work queue functions run in context of a special kernel process. Therefore, they are more flexible. In particular, work queue functions can sleep and have access to user space memory. A work queue is represented by the structure `workqueue_struct`. It is possible to declare and initialize arbitrary many work queues. A work, i.e. a variable of type `work_struct` saving a pointer to a handler function, can be added to a work queue. Alternatively, there is a shared work queue provided by the kernel. Since most device drivers in the Linux kernel just use the shared work queue, we formalize and support only this one. A work can be added to the shared work queue using the function `schedule_work(work)`, where the `work` variable must be initialized using either `INIT_WORK` or `PREPARE_WORK` before.

2.2.4. Interrupts

In general, an *interrupt* is a signal that is sent by either a software function or a hardware device. In what follows, we only deal with hardware interrupts. They are sent by devices to notify the corresponding device drivers that something has

happened. For that reason, the device driver must register an *interrupt handler function*, which receives interrupt signals on a given *interrupt line* number. The number of available interrupt lines is very restricted. On the x86 architecture, there are only 16 of them, which are shared by all the hardware inside the computer and the external devices connected to it. Therefore, one interrupt line can be requested by more than one device. It is possible to request a non-shared interrupt line, but in most cases this is not a good idea due to the very restricted resources. We only formalize shared interrupts, but it is not hard to extend the formalization to exclusive interrupt line requesting. Because DDVerify can only check one device driver at the time, there is no difference between shared and non-shared interrupt requests for it.

An interrupt line is requested using the function `request_irq` with the following parameters:

- `irq`: number of the requested interrupt line. In general, it can be a hard task to find out which interrupt line is free and can be used by the device driver. More information about the possible techniques can be found in [28].
- `function`: the handler function. If an interrupt request arrives on the requested interrupt line, this function will be called.
- `flags`: a set of flags to indicate a "fast" interrupt handler, a shared or an exclusive interrupt request, and several other things. Because we do not make a difference between fast and slow interrupt handlers, and we only allow shared interrupt requests, we do not care about these flags in the following.
- `dev_name`: a string used for the information output in `/proc/interrupts`. All owners of the interrupt lines are shown in this file.
- `dev_id`: a unique identifier used to identify the corresponding device driver of an interrupt request. When an interrupt request arrives on a shared interrupt line, this identifier is used by the device driver to find out whether the interrupt request was generated by the appropriate hardware device or by some others.

The value returned by the requesting function is either 0 to indicate success or a negative error code. A correct requested interrupt line can be released using the function `free_irq`.

Interrupt handler functions must have a fixed type:

```
irqreturn_t handler(int irq, void *dev_id, struct pt_regs *regs)
```

The second argument is used to identify whether the interrupt request was sent by the hardware corresponding to the device driver or not. In the first case, the function must return the value `IRQ_HANDLED` to indicate that this interrupt request is handled by this handler function. Otherwise, the value `IRQ_NONE` must be returned. Interrupt handlers are not executed in the context of a process. Therefore, they have

several restrictions. An interrupt handler function cannot transfer data to or from user space, it cannot do anything that could cause the execution to go to sleep, such as calling `wait_event`, allocating memory with anything other than `GFP_ATOMIC`, or locking a semaphore. Finally, interrupt handlers may not call `schedule`. Therefore, interrupt handlers are usually restricted to doing the most important work, such as saving new data. Most of the work is done by a deferred function. This splitting of work is known as the *top and bottom halves* of interrupt handlers. The top half is the handler function that actually responds to the interrupt and the bottom half is a function that is scheduled by the top half to be executed later, at a safer time. Those functions are usually realized using work queues or tasklets, which we have presented before.

Requesting and releasing shared interrupt lines are the only methods which are formalized and supported by DDVerify. We do not distinguish between fast and slow interrupt handlers. Exclusive interrupt line requesting, as well as enabling and disabling interrupts, are not supported.

2.2.5. I/O Ports and I/O Memory

Most devices have registers that are used to control the device or to read its status. The registers are mapped to I/O addresses, which are usually called *I/O ports*. In the x86 architecture the I/O address space provides up to 65,536 8-bit I/O ports. Before a device driver may access an I/O port, it has to allocate it first. This is done with the function `request_region`. If the requesting succeeds, the device driver has exclusive access to those ports. To release a chunk of ports, the function `release_region` is used. A rich set of functions to read and write to I/O ports is provided by the Linux kernel API. The complete list can be found in Appendix A. DDVerify supports all of these functions.

I/O memory is another concept to make memory on a device available to device drivers. I/O ports are only used if several bytes must be read or written from a control or status register. In contrast to this, I/O memory is mostly used to handle larger data sets, like video data or ethernet packets. I/O memory regions must be allocated prior to use. This is done with the function `request_mem_region`. Correspondingly, `release_mem_region` is used to free memory regions. After a memory region has been requested, it must be made available to the kernel before use. The function `ioremap` takes the physical address of I/O memory and the size of the requested region and returns a pointer. This pointer is not allowed to be dereferenced directly. The I/O memory, to which the pointer points to, can be used with a set of functions provided for that purpose. DDVerify supports all functions for I/O memory usage.

The Linux kernel API provides the possibility to map I/O ports to I/O memory. This is done with this function:

```
void *ioport_map(unsigned long port, unsigned int count)
```

It remaps `count` I/O ports, starting from port `port`, and makes them appear to be

I/O memory. The mapping can be canceled using the function `ioport_unmap`. I/O port mapping is also supported by DDVerify.

2.2.6. Memory Usage

In user space programs, the functions `malloc` and `free` are used to allocate and to free memory. Correspondingly, `kmalloc` and `kfree` are used for kernel space memory, but they have to be used more carefully. The behaviour of the memory allocation process can be controlled using flags, which are provided to `kmalloc` as the second argument. The most commonly used flags are `GFP_KERNEL` and `GFP_ATOMIC`. The first one means that the allocation is done by executing a system call function on behalf of a process. Consequently, `kmalloc` can put the current process to sleep, waiting for free memory. When a function uses the flag `GFP_KERNEL` to allocate memory, it must be reentrant and may not be running in atomic context.

Whenever the current process is not allowed to sleep, the flag `GFP_ATOMIC` must be used to allocate memory. This is the case, for instance, in interrupt handlers, tasklets, kernel timers or when a lock is held.

DDVerify supports allocating memory using `kmalloc`, and freeing it with `kfree`. It is checked that the flag `GFP_KERNEL` is not used in interrupt context where sleeping is not allowed. All other flags are ignored. Other methods of allocating memory are not supported.

2.3. General Structure of a Device Driver's Source Code

The general structure of device drivers' code is similar most of the time: Each device driver has an initialization function which is called when the device driver is loaded into memory. One of its duties is to inform the kernel which functions are provided to user space applications by the device driver. When unloaded from the system, the device driver's exit function is invoked, which does all the cleanup work. The macros `module_init` and `module_exit` are used in the source code of a device driver to notify the initialization and the exit function. In the following, we give a short overview over the general structure of character and block device drivers.

2.3.1. Character Device Drivers

At first, a character device driver has to register a set of device numbers. A *device number* is split into a *major number* and a *minor number*. Major numbers are used to identify the registered drivers. Minor numbers exactly determine which device is being referred to. Device numbers are represented by the data type `dev_t`. There are two different ways for a device driver to obtain one or more device numbers. Either a given set of device numbers is statically registered, or the allocation is done dynamically. Static allocation only succeeds if all the device numbers are free. Otherwise, the registration fails. Hence, in most cases, dynamic allocation of device

```
#include <...>

dev_t number;
struct cdev cdev;

int mydev_open(...)
{
  return 0;
}

int mydev_release(...)
{
  return 0;
}

ssize_t mydev_read(...)
{
  return 0;
}

struct file_operations fops = {
  .owner = THIS_MODULE,
  .open = mydev_open,
  .read = mydev_read,
  .release = mydev_release,
};
```

```
int driver_init()
{
  int err;

  err = alloc_chrdev_region(
        &number, 0, 1, "mydev");
  if (err < 0) {
    return err;
  }

  err = cdev_add(&cdev, number, 1);
  if (err) {
    return err;
  }

  return 0;
}

void driver_exit()
{
  unregister_chrdev_region(&number, 0);
  cdev_del(&cdev);
}

module_init(driver_init);
module_exit(driver_exit);
```

Figure 2.2.: Exemplary code of a character driver

numbers is the better choice. The functions for static and dynamic allocation of device numbers are `register_chrdev_region` and `alloc_chrdev_region`.

Once the device driver has successfully obtained the required device numbers, it may register the devices. A character device is represented by a structure of type `struct cdev`. A valid `cdev` structure is allocated using the function `cdev_alloc`. One of the most important fields of the `cdev` structure is `ops`, a structure of type `struct file_operations`. It is a collection of function pointers and defines the provided functionality of the device driver. If the `cdev` structure is statically defined, the function `cdev_init` is used to initialize it along with a `file_operations` structure. The last thing to do is to inform the kernel about the device by using the function `cdev_add`. After calling `cdev_add`, the device is registered in the system and can immediately be used by user space programs. To remove a char device from the system, the function `cdev_del` is used.

2.3.2. Block Device Drivers

Similar to character device drivers, block device drivers must use a registration interface to make their devices available to the kernel. The concepts are similar,

but the details of block device registration are different.

The first step is to execute the function `register_blkdev` in order to register the driver to the kernel and to obtain a major number for the driver. The corresponding function for canceling a block device driver registration is `unregister_blkdev`. After the registration, the kernel is informed about the driver but there are no devices alive. Block devices, which are also called *disk devices*, are represented in the kernel by the structure `struct gendisk`. A structure of this type is dynamically allocated using the function `alloc_disk`. The function takes one argument, which should be the number of minor numbers of the disk device. After allocating a gendisk, several fields of this structure must be initialized. The most important ones are `fops` and `queue`. The first one is a pointer to a structure of type `struct block_device_operations`. Like the file operations structure in character device drivers, this structure is a collection of function pointers defining the implemented functionalities of the device driver. The queue field is a pointer to a request queue.

The core of every block device driver is the *request queue*, represented by a variable of type `struct request_queue`, and its *request function*. The request queue stores all block read and write requests, which are then performed by the request function. A request queue variable is allocated using the function `blk_init_queue`, which takes two arguments: a pointer to the driver's request function and an initialized spinlock. This spinlock is used to restrict concurrent calls of the request function.

After allocating a gendisk structure and initializing its fields, the device can be registered to the kernel. This is done using `add_disk`. From this point, the device is registered and can immediately be used by user programs. In the driver's cleanup function, `del_gendisk` should be called to release the disk devices. It is also necessary to release the driver's request queue using `blk_cleanup_queue`.

2.4. Bugs in Device Drivers

In software development, most of the bugs follow a small set of patterns, e.g. out-of-bounds while accessing an array or use of non initialized variables. In this section, we present the most typical bugs that appear in device drivers.

In [17], different versions of the Linux kernel are checked with twelve different checkers in order to prepare a study about bugs in operating systems. Figure 2.3 shows the proportional distribution of device driver bugs, which were found with eight of these checkers in the Linux kernel 2.4.1. The block-checker checks that blocking functions are not called while interrupts are disabled or a spinlock is held. These errors may cause a deadlock of the whole system. An exemplary bug [11] can be found in the driver `sound/isa/sb/sb16_csp.c` of the Linux kernel 2.6.3:

```
spin_lock_irqsave(&p->chip->reg_lock, flags);
...
unsigned char *kbuf, *_kbuf;
_kbuf = kbuf = kmalloc(size, GFP_KERNEL);
```

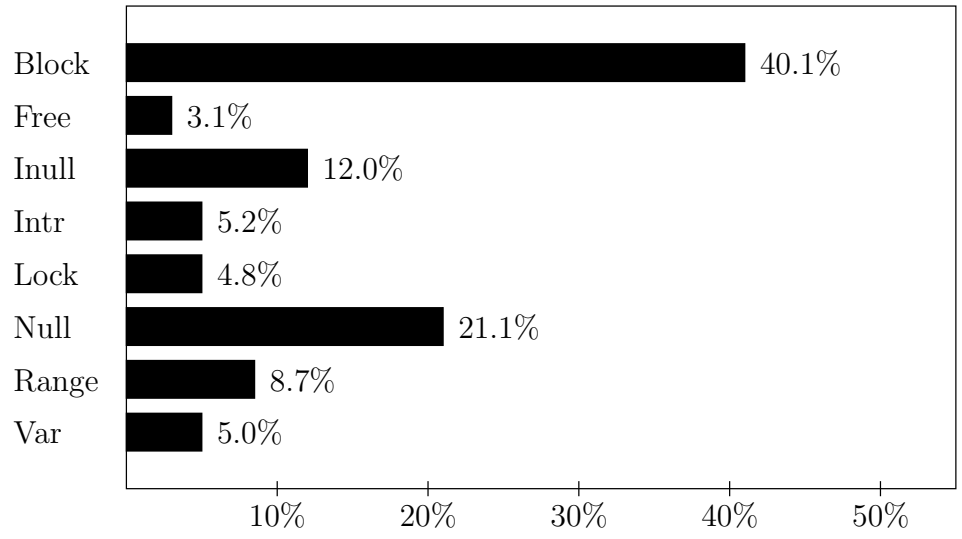

Figure 2.3.: Statistics about errors in Linux device drivers

First, a spinlock is acquired. Several lines later, `kmalloc` is called with the parameter `GFP_KERNEL`. Hence, it is allowed to go to sleep (see Section 2.2.6), possibly causing a deadlock.

The free-checker checks that freed memory is not used any more. An example [58] for a use-after-free error is shown in the following snippet of the device driver `drivers/message/i2o/pci.c` of the Linux kernel 2.6.13:

```
static int i2o_pci_probe(struct pci_dev *pdev,
                         const struct pci_device_id *id)
{
   struct i2o_controller *c;
   ...
   c = i2o_iop_alloc();
   ...
   i2o_iop_free(c);
   put_device(c->device.parent);
   ...
}
```

The function `i2o_iop_alloc` allocates memory for an `i2o_controller` structure using `kmalloc`. At the end of the listing, this memory is freed through `i2o_iop_free`. The bug in this piece of code arises from the call of `put_device`, since its parameter `c->device.parent` causes an already freed pointer to be dereferenced.

The Inull-checker checks if inconsistent assumptions are made about whether a pointer is `NULL`. The following part of the source code of the device driver

`drivers/isdn/avmb1/capidrv.c`, taken from Linux kernel 2.4.1, is an example for this kind of errors [34]:

```
if (card == NULL) {
   printk(KERN_ERR "capidrv-%d: ... %d!\n", card->contrnr, id);
}
```

The `printk` function is only executed if `card` is a `NULL`-pointer. But this pointer is dereferenced by providing `card->contrnr` as a parameter to `printk`.

The Intr-checker checks that disabled interrupts are restored. The Lock-checker checks whether acquired locks are released and that acquired locks are not acquired again. An exemplary locking related bug, which we have found in the current Linux kernel 2.6.19 using DDVerify, is presented in Section 6.3. The Range-checker checks for bounds of array indices and loop bounds derived from user data. The Null-checker is used to find bugs related to dereferencing `NULL`-pointers. Finally, the Var-checker checks that large stack variables (> 1 KByte) are not allocated on the fixed-size kernel stack.

3. Theoretical Background

In this chapter, we introduce the theoretical concepts which are used for the formalization of the Linux kernel API in the next chapter. The first section gives an introduction to the Calculus of Communicating Systems. It is used to define concurrent systems formally. In Section 3.2, we give a brief introduction to the LTL, a temporal logic used to specify properties.

3.1. Calculus of Communicating Systems

The *Calculus of Communicating Systems* (CCS) [55, 56] is a process algebra developed for the analytical modeling of concurrent and distributed systems. We present an extended version of CCS with priorities and the value passing extension. The formal notation used in this introduction is based on [56, 24].

The main concepts of CCS are *processes*[1] and *channel communication*. A concurrent system is described by a set of CCS processes. Their communication over channels corresponds to the system's behaviour. For example, consider a simple vending machine that sells tea and coffee. A person may throw one coin into the vending machine and gets a cup of tea after pushing a button. For another coin, the person gets a cup of coffee. The vending machine can be described with one CCS process:

$$Vending \stackrel{\text{def}}{=} coin.(button.\overline{tea}.Vending + coin.button.\overline{coffee}.Vending)$$

Here, *coin*, *button*, *tea* and $coffee$ are channel labels. Channels are used for the communication between processes. An action can have the name of either the label of a channel or its co-label, like $\overline{tea}$. The former are defined to be input actions over channels, whereas the co-labels define an output action. The $Vending$ process makes an input transition over the channel *coin* and results in a sum of two processes. The first summand proceeds with an input transition over *button*, an output transition over *tea* and restores the process of the vending machine. The second summand is similar, but requires another input transition over *coin*.

A person handling the vending machine can be defined by the following CCS process:

$$Person \stackrel{\text{def}}{=} \overline{coin}.(\overline{button}.tea.Person + \overline{coin}.\overline{button}.coffee.Person)$$

The person's process definition is very similar to the process of the vending machine, just the labels are changed by the corresponding co-labels, and vice versa.

[1] In several publications, processes are also called *agents*.

This is quite natural, because the person outputs on the channels *coin* and *button* and expects an output of the vending machine on the channels *tea* and $coffee$, which are inputs from the persons's point of view.

Both processes can be composed into one process describing a system of a vending machine and a person using it:

$$Sys \stackrel{\text{def}}{=} (Vending \mid Person) \setminus \{coin, button, tea, coffee\}$$

The system's process definition is a parallel composition of the processes $Vending$ and $Person$. This composition is restricted to all four channels. Intuitively this means that these channels are not available for other processes outside of Sys. The corresponding actions are invisible, because they happen inside of the system's process. Furthermore, the vending machine's process and the person's process are allowed to perform handshake communications over the restricted channels only. That means, if an action in one process and a corresponding co-action in the other process is possible, both continue simultaneously with an invisible handshake-communication over the channel.

3.1.1. CCS with Priorities

After the previous informal introduction, we now define CCS with priorities in a formal way. Let $\{\Lambda_k \mid k \in \mathbb{N}\}$ denote a family of pairwise-disjoint, countably infinite sets of *labels*. Λ_k contains the channels with priority k that processes may synchronize over. The set of *co-labels* with priority k is defined by $\overline{\Lambda}_k = \{\overline{\alpha} \mid \alpha \in \Lambda_k\}$. We then define the set of all labels by $\mathcal{L} = \bigcup\{\Lambda_k \mid k \in \mathbb{N}\}$. For better readability we write α_k if $\alpha \in \Lambda_k$. We use $\alpha_k, \beta_k, \gamma_k, \ldots$ to range over labels, and $\overline{\alpha}_k, \overline{\beta}_k, \overline{\gamma_k}, \ldots$ to range over co-labels. The handshake communication with priority k over channel α is denoted with τ_k^α. The set of actions with priority k is defined by $Act_k = \Lambda_k \cup \overline{\Lambda}_k \cup \{\tau_k^\alpha \mid \alpha \in \Lambda_k\}$. Accordingly, the set $Act = \bigcup\{Act_k \mid k \in \mathbb{N}\}$ contains all actions.

The syntax of CCS with priorities is defined by the following Backus-Naur-form:

$$P ::= nil \quad | \quad \alpha_k.P \quad | \quad \sum_{i \in I} P_i \quad | \quad P \mid Q \quad |$$
$$P \rhd Q \quad | \quad P \setminus L \quad | \quad P[f] \quad | \quad C \stackrel{\text{def}}{=} P$$

where P_i, P and Q range over the set of processes, $L \subseteq \mathcal{L}$, and I is a possibly infinite indexing set. $f : \mathcal{L} \mapsto \mathcal{L}$ is a relabeling function for labels. The intuitive definitions of the processes are as follows:

- *nil* represents the process which cannot perform any action.
- $\alpha_k.P$ denotes a process which can perform an action $\alpha \in Act$ with priority k and then behaves like process P.

$$
\begin{aligned}
I_k(\alpha_l.E) &= \begin{cases} \{a_l\} & \text{if } l = k \\ \emptyset & \text{otherwise} \end{cases} \\
I_k(\sum_{i \in I} P_i) &= \bigcup_{i \in I} I_k(P_i) \\
I_k(P \setminus L) &= I_k(P) \setminus (L \cup \overline{L}) \\
I_k(P[f]) &= \{f(\alpha_k) \mid \alpha_k \in I_k(P)\} \\
I_k(P \mid Q) &= I_k(P) \cup I_k(Q) \cup \{\tau_k^\alpha \mid \alpha_k \in I_k(P) \cap \overline{I}_k(Q)\} \\
I_k(P \rhd Q) &= I_k(P) \cup I_k(Q) \\
I_k(C) &= I_k(P) \text{ where } C \stackrel{\text{def}}{=} P \\
I_{<k}(P) &= \bigcup \{I_l(P) \mid 0 \leq l < k\}
\end{aligned}
$$

Figure 3.1.: Initial action sets

- $\sum_{i \in I} P_i$ is the summation of processes over an indexing set I. This process continues as one of the processes P_i. The choice is done nondeterministically, but can be restricted by the priorities of handshake actions. For the binary summation we also write $P + Q$ for two processes P and Q.
- $P \mid Q$ is the parallel composition of the processes P and Q.
- $P \rhd Q$ behaves like P and is capable of disabling P by engaging in Q.
- $P \setminus L$ restricts the actions contained in L to be handshake actions in the process P. For processes outside of $P \setminus L$, the channels defined in L are not available to communicate with P.
- $P[f]$ behaves like P but with the action relabeled by function f.
- $C \stackrel{\text{def}}{=} P$ defines an abbreviation C for a process P.

The operational semantics is given by a labeled transition system $\langle \mathcal{P}, \mathcal{A}, \rightarrow \rangle$, where $\mathcal{P}$ is the set of processes, $\mathcal{A}$ is the set of actions and $\rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ is the transition relation. We write $P \xrightarrow{\alpha_k} P'$ instead of $\langle P, \alpha_k, P' \rangle \in \rightarrow$.

The definition for the operational rules of the transition systems requires the concept of *initial action sets* which are inductively defined on the syntax of processes, as shown in Figure 3.1. $I_k(P)$ denotes the set of all initial actions of a process P with the priority k. $I_{<k}(P)$ is the set of all initial action with a higher priority than k. We write $\alpha \in I_{<k}(P)$ if there exists an l such that $\alpha_l \in I_l(P)$ and $l < k$.

The only axiom **Act** and the rules for the transition relation $\rightarrow$ are shown in Figure 3.2. Note that the rules $\mathbf{Sum_1}$ and $\mathbf{Sum_2}$ for the binary summation are only a specialization of the general summation rule $\mathbf{Sum_j}$. Both are explicitly given because the binary summation is used very often in what follows.

$$\textbf{Act}\quad \frac{}{\alpha_k.P \xrightarrow{\alpha_k} P}$$

$$\textbf{Sum}_\textbf{j}\quad \frac{P_j \xrightarrow{\alpha_k} P'_j}{\sum_{i\in I} P_i \xrightarrow{\alpha_k} P'_j}\ \forall i \in I.i \neq j \implies \tau^\beta \notin I_{<k}(P_i)$$

$$\textbf{Sum}_\textbf{1}\quad \frac{P_1 \xrightarrow{\alpha_k} P'_1}{P_1 + P_2 \xrightarrow{\alpha_k} P'_1}\ \tau^\beta \notin I_{<k}(P_2) \qquad \textbf{Sum}_\textbf{2}\quad \frac{P_2 \xrightarrow{\alpha_k} P'_2}{P_1 + P_2 \xrightarrow{\alpha_k} P'_2}\ \tau^\beta \notin I_{<k}(P_1)$$

$$\textbf{Com}_\textbf{1}\quad \frac{P \xrightarrow{\alpha_k} P'}{P \mid Q \xrightarrow{\alpha_k} P' \mid Q}\ \tau^\beta \notin I_{<k}(P \mid Q)$$

$$\textbf{Com}_\textbf{2}\quad \frac{Q \xrightarrow{\alpha_k} Q'}{P \mid Q \xrightarrow{\alpha_k} P \mid Q'}\ \tau^\beta \notin I_{<k}(P \mid Q)$$

$$\textbf{Com}_\textbf{3}\quad \frac{P \xrightarrow{\alpha_k} P' \quad Q \xrightarrow{\overline{\alpha}_k} Q'}{P \mid Q \xrightarrow{\tau_k^\alpha} P' \mid Q'}\ \tau^\beta \notin I_{<k}(P \mid Q)$$

$$\textbf{Dis}_\textbf{1}\quad \frac{P \xrightarrow{\alpha_k} P'}{P \rhd Q \xrightarrow{\alpha_k} P' \rhd Q}\ \tau^\beta \notin I_{<k}(Q) \quad \textbf{Dis}_\textbf{2}\quad \frac{Q \xrightarrow{\alpha_k} Q'}{P \rhd Q \xrightarrow{\alpha_k} Q'}\ \tau^\beta \notin I_{<k}(P)$$

$$\textbf{Res}\quad \frac{P \xrightarrow{\alpha_k} P'}{P \setminus L \xrightarrow{\alpha_k} P' \setminus L}\ \alpha_k \notin (L \cup \overline{L}) \qquad \textbf{Con}\quad \frac{P \xrightarrow{\alpha_k} P'}{C \xrightarrow{\alpha_k} P'}\ C \stackrel{\text{def}}{=} P$$

$$\textbf{Rel}\quad \frac{P \xrightarrow{\alpha_k} P'}{P[f] \xrightarrow{\beta_k} P'[f]}\ f(\alpha) = \beta$$

Figure 3.2.: Operational semantics for CCS with priorities

The question may arise whether the extension of the classical CCS with priorities is really necessary or if it is syntactic sugar. We show an example, taken from [23], to illustrate the necessity of priorities in process algebras in general. Consider the following system:

$$C \stackrel{\text{def}}{=} up.C^1 + i.nil$$
$$C^{n+1} \stackrel{\text{def}}{=} up.C^{n+2} + down.C^n + i.nil$$
$$Irp \stackrel{\text{def}}{=} shutDown.\bar{i}.nil$$
$$Sys \stackrel{\text{def}}{=} (C \mid Irp) \setminus \{i\}$$

The process C acts as an infinite counter, while Irp is designed to halt C when the environment issues a *shutDown* request. After the arrival of a signal on this

$$
\begin{aligned}
\mathcal{L}(nil.P) &= \emptyset \\
\mathcal{L}(\alpha_k.P) &= \{\alpha\} \cup \mathcal{L}(P) \\
\mathcal{L}(\overline{\alpha}_k.P) &= \{\alpha\} \cup \mathcal{L}(P) \\
\mathcal{L}(\tau_k^{\alpha}.P) &= \mathcal{L}(P) \\
\mathcal{L}(\sum_i P_i) &= \bigcup_i \mathcal{L}(P_i) \\
\mathcal{L}(P \mid Q) &= \mathcal{L}(P) \cup \mathcal{L}(Q) \\
\mathcal{L}(P \rhd Q) &= \mathcal{L}(P) \cup \mathcal{L}(Q) \\
\mathcal{L}(P \setminus L) &= \mathcal{L}(P) - (L \cup \overline{L}) \\
\mathcal{L}(P[f]) &= \{f(l) \mid l \in \mathcal{L}(P)\}
\end{aligned}
$$

Figure 3.3.: Definition of sort $\mathcal{L}(P)$ of a process expression P

channel, the internal handshake communication over channel i would be available. After this, the counter would end up in a deadlock process. But nothing forces the counter process to do the handshake. It may also continue with either *up* or *down*. The system cannot be described correctly using CCS only. When priorities are added to the formalization, we can achieve the expected behaviour. To do so, we have to assign a higher priority to the action i than to the actions *up* and *down*. If the handshake communication over i is available, the counter process must do this and will be stopped afterwards. This kind of interrupting systems is the basis of several concepts in operating systems. Therefore, we have added the concept of priorities to our formalization.

3.1.2. Sorts

A *sort* L is a set of labels. We say that a process P has sort L, and write $P : L$, if all the actions which P may perform at any time in the future have labels in L. Clearly, if L is a sort of P, and $L \subseteq L'$, then L' is a sort of P, too. Normally, we are interested in the smallest sort L of P such that $P : L$. The definition of a sort for a process expression E is shown in Figure 3.3.

In the following, we will define so called *access sorts* for CCS process definitions. An access sort for a CCS process P defines the labels by which other processes may communicate with P. We denote access sorts with ACC.

3.1.3. Value Passing Extension

The value passing extension of CCS allows to define processes which can send and receive values over channels. We define the values to be natural numbers, but in general every countably infinite domain may be chosen.

$$
\begin{aligned}
[\![nil]\!] &\mapsto nil \\
[\![\alpha_k(x).P]\!] &\mapsto \sum_{v \in V} \alpha_k^v.[\![P\{x := v\}]\!] \\
[\![\overline{\alpha}_k(e).P]\!] &\mapsto \overline{\alpha}_k^e.[\![P]\!] \\
[\![\sum_{i \in I} P_i]\!] &\mapsto \sum_{i \in I} [\![P_i]\!] \\
[\![P \mid Q]\!] &\mapsto [\![P]\!] \mid [\![Q]\!] \\
[\![P \rhd Q]\!] &\mapsto [\![P]\!] \rhd [\![Q]\!] \\
[\![P \setminus L]\!] &\mapsto [\![P]\!] \setminus \{\alpha^v \mid \alpha \in L,\, v \in V\} \\
[\![P[f]]\!] &\mapsto [\![P]\!][f'] \text{ where } f'(l^v) = f(l)^v \\
[\![\textbf{if } b \textbf{ then } P \textbf{ else } Q]\!] &\mapsto \begin{cases} [\![P]\!] & \text{if } b = true \\ [\![Q]\!] & \text{otherwise} \end{cases} \\
[\![C\langle e_1, \ldots, e_n\rangle]\!] &\mapsto C_{e_1,\ldots,e_n}
\end{aligned}
$$

Figure 3.4.: Definition of the translation function $[\![\cdot]\!]$ from the value passing CCS to the classical calculus

The set of CCS processes is extended in the following way:

$$
\begin{aligned}
VP ::= {} & \alpha_k(x).P \quad | \quad \overline{\alpha}_k(e).P \quad | \\
& C(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P \quad | \quad C\langle e_1, \ldots, e_n\rangle \quad | \\
& \textbf{if } b \textbf{ then } P \textbf{ else } Q
\end{aligned}
$$

Where $x, x_1, \ldots, x_n$ are variables, $e, e_1, \ldots, e_n$ are value expressions, i.e. representing natural numbers, and b is a boolean expression. The value passing extension also allows the parametrized definition and instantiation of processes, and the conditional branching between two processes.

Milner shows in [56] that this extension of CCS is syntactic sugar only. Therefore, it is possible to give a translation function from process definitions denoted in the value passing calculus to the classical one. The fundamental idea of this translation function $[\![\cdot]\!]$ from VP to P is, to consider a channel α_k together with its actual content v, as a new channel name α_k^v. Therefore, for each data value there is a separate channel name. The translating equations for the function $[\![]\!]$ are shown in Figure 3.4. Furthermore, a single defining process equation $C(\vec{x}) \stackrel{\text{def}}{=} P$ is translated into the indexed set of defining process equations

$$\{C_{\vec{v}} \stackrel{\text{def}}{=} [\![P\{\vec{x} := \vec{v}\}]\!] \mid \vec{v} \in V^n\}$$

where n is given by $\vec{x} = x_1, \ldots, x_n$.

3.2. LTL

In the previous section, we defined the Calculus of Communicating Systems in order to describe concurrent systems in a formal way. A temporal logic is then used to provide *specifications* (also called properties) for those systems and to prove them. *Linear-time temporal logic* [50, 18], or LTL for short, is a temporal logic to reason about states on future paths. An LTL formula ϕ is satisfied in a state s, if all paths starting in that state satisfy ϕ. Thus, LTL implicitly quantifies universally over paths.

LTL has the following syntax given in Backus-Naur-form:

$$\phi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \mathrm{X}\ \phi \mid \mathrm{F}\ \phi \mid \mathrm{G}\ \phi \mid \phi\ \mathrm{U}\ \phi \mid \phi\ \mathrm{R}\ \phi$$

where p is any proposition atom from some set $\mathcal{A}$. Thus, the symbols $\top$ and $\bot$ are LTL formulas, as are all atoms from $\mathcal{A}$, and $\neg\phi$ is an LTL formula if ϕ is one, etc. The connectives X, F, G, U and R are called *temporal connectives*. X means "neXt state", F means "some Feature state", and G means "Globally", i.e. all feature states. The connectives U and R are called "Until" and "Release" respectively. Usually, $\mathcal{A}$ defines a set of atomic formulas denoting atomic facts which may hold in a system. Because we want to reason about systems defined by CCS processes, we define $\mathcal{A}$ to be Act, i.e. the set of all actions, in the following. An atomic formula $p \in \mathcal{A}$ is then true in a state s, if the action p can be executed in this state. This brings us to the semantics of LTL.

A model for an LTL formula is defined by a transition system $\mathcal{M} = (S, \rightarrow, L)$, where S is a set of states, $\rightarrow \subseteq S \times S$ is a binary relation on S, and $L : S \mapsto \mathcal{P}(\mathcal{A})$ is a labelling function for states. The latter one defines the atomic formulas which are true in the states. To reason about the future of a state $s \in S$, we need the definition of a path in a transition system $\mathcal{M}$: A path is an infinite sequence of states s_1,s_2,s_3, ... in S such that, for each $i \geq 1$, $s_i \rightarrow s_{i+1}$ holds. We write paths as $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$. The suffix of a path π, starting at state s_i, is denoted by π^i. Whether a path $\pi = s_1 \rightarrow \ldots$ of a transition system $\mathcal{M}$ satisfies an LTL formula is defined by the satisfaction relation $\models$ as follows:

1. $\pi \models \top$
2. $\pi \not\models \bot$
3. $\pi \models p$ iff $p \in L(s_1)$
4. $\pi \models \neg\phi$ iff $\pi \not\models \phi$
5. $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$
6. $\pi \models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$
7. $\pi \models \phi_1 \rightarrow \phi_2$ iff $\pi \models \phi_2$ whenever $\pi \models \phi_1$
8. $\pi \models X\ \phi$ iff $\pi^2 \models \phi$

9. $\pi \models G\ \phi$ iff $\forall i \geq 1.\pi^i \models \phi$
10. $\pi \models F\ \phi$ iff $\exists i \geq 1.\pi^i \models \phi$
11. $\pi \models \phi\ U\ \psi$ iff $\exists i \geq 1.(\pi^i \models \psi \wedge \forall j.1 \leq j < i \rightarrow \pi^j \models \phi)$
12. $\pi \models \phi\ R\ \psi$ iff $\exists i \geq 1.(\pi^i \models \phi \wedge \forall j.1 \leq j \leq i \rightarrow \pi^j \models \psi) \vee \forall k \geq 1.\pi^k \models \psi$

For a given transition system $\mathcal{M} = (S, \rightarrow, L)$, a state $s \in S$ fulfills an LTL formula ϕ if and only if ϕ is satisfied by every path π in $\mathcal{M}$ starting at s. We then write $\mathcal{M}, s \models \phi$.

$$
\begin{aligned}
L(nil) &= \emptyset \\
L(\alpha_k.P) &= \{\alpha\} \cup L(P) \\
L(\sum_{i \in I} P_i) &= \bigcup_{i \in I} L(P_i) \\
L(P \mid Q) &= L(P) \cup L(Q) \\
L(P \triangleright Q) &= L(P) \cup L(Q) \\
L(P \setminus L) &= L(P) \\
L(P[f]) &= \{f(a) \mid a \in L(P)\} \\
L(C) &= L(P) \text{ if } C \stackrel{\text{def}}{=} P
\end{aligned}
$$

Figure 3.5.: Labeling function for CCS processes

The operational semantics of CCS is also defined by a transition system that can be used to specify LTL formulas for it. It does not matter that this transition system is labeled. The labeling function L for CCS processes is defined in Figure 3.5.

4. Operational Semantics

In this chapter we give an operational semantics for concurrent C programs. We consider a subset of C, called C_0. This restriction comes without loss of generality, since it can easily be shown that every C program can be translated into C_0. For this language, we give an semantics using CCS. This approach makes it more easier to reason about concurrency. Furthermore, we given the semantics of several parts of the Linux kernel API. This makes it possible to reason about C_0 programs which makes use of them. Because the operational semantics of CCS is given by a labeled transition system, it is not trivial to implement a corresponding model in C. Therefore, in the last step we give an operational semantics for concurrent C programs and the Linux kernel API in a way, which is closer to C.

The approach to define the semantics of a parallel imperative programming language in CCS roots back to Milner's book [56], where an operational semantics for the concurrent programming language $\mathcal{M}$ was defined. In [67], CCS was used to define an operational semantics for a real-time object-oriented programming language. Our semantics for C_0 is based on Milner's approach, but is simplified in many ways to make the concepts more clearer without loosing in details. To our best knowledge, there are no other approaches to formalize parts of the Linux API with a process algebra.

4.1. The Programming Language $\mathcal{C}_0$

The programming language C is very expressive. In the following, we only consider a very restricted subset of C, named $\mathcal{C}_0$. While-loops are the only supported loop constructs. Furthermore, recursion is not allowed. Both are no real restriction, because while-programs are Turing complete [35]. Hence, all other constructions can be simulated using while-loops only. In theory, it is possible to translate a C program into $\mathcal{C}_0$.

On the syntactical level, we distinguish between two different function calls: function calls to functions defined in the $\mathcal{C}_0$ program, and function calls to external functions, e.g. functions of the Linux kernel API. The first ones are restricted in that neither parameters nor return values are allowed. Calls to external function may have arbitrary many parameters, but they are required to return an integer value. In $\mathcal{C}_0$, only global variables may be declared. It is allowed to declare variables of arbitrary type, but only integer variables are directly supported by $\mathcal{C}_0$. As soon as the model of the Linux kernel API is presented, we introduce further types of variables. Expressions over integers are also very restricted, only the binary sum of

⟨Statement⟩ ::= ⟨VarName⟩ `=` ⟨IntExpr⟩
| ⟨Statement⟩`;`⟨Statement⟩
| `if` ⟨IntExpr⟩ { ⟨Statement⟩ }
| `if` ⟨IntExpr⟩ { ⟨Statement⟩ } `else` { ⟨Statement⟩ }
| `while` ⟨IntExpr⟩ { ⟨Statement⟩ }
| `void` ⟨FuncName⟩`()`
| ⟨FuncCall⟩
| ⟨VarName⟩ `=` ⟨FuncCall⟩

IntExpr ::= ⟨IntNumber⟩
| ⟨VarName⟩
| ⟨VarName⟩ `+` ⟨IntNumber⟩
| ⟨VarName⟩ `==` ⟨IntNumber⟩

FuncDecl ::= `void` ⟨FuncName⟩`()` { ⟨Statement⟩ }
| ⟨FuncDecl⟩`;`⟨FuncDecl⟩

VarDecl ::= ⟨VarType⟩ ⟨VarName⟩
| `int` ⟨VarName⟩
| `int` ⟨VarName⟩ `=` ⟨IntNumber⟩
| ⟨VarDecl⟩`;`⟨VarDecl⟩

Prog ::= ⟨VarDecl⟩⟨FuncDecl⟩

Figure 4.1.: Syntax of $\mathcal{C}_0$

a variable and a constant, and the comparison of two variables are allowed. Multiplication and division can be implemented using this basic operations and loops. The syntax of $\mathcal{C}_0$ is given by its Backus-Naur-form in Figure 4.1. ⟨VarName⟩ and ⟨FuncName⟩ represent the syntax of correct variable and functions names in C. A syntactically correct function call, i.e. the function name and a set of parameters, is defined by ⟨FuncCall⟩. $\mathcal{C}_0$ supports the instantiation of arbitrary variable types. The syntax of a correct type name in C is defined by ⟨TypeName⟩.

$\mathcal{C}_0$ is a strict sequential programming language. Concurrency is introduced when we present the formalization of the Linux kernel API. We use the parallel composition of CCS to run several processes, like tasklet functions or interrupt handlers, in parallel to the main program execution. For sake of simplicity, we do not allow concurrent calls to one function. That means that a function, which is declared in a $\mathcal{C}_0$ program, cannot be executed by two processes at the same time. In this case, one of these processes must wait until the other has left the function's body. In [56], a general approach to the formalization of program functions is presented, which also allows parallel and recursive function calls.

4.2. Operational Semantics of $\mathcal{C}_0$

In this section we give an operation semantics for the programming language $\mathcal{C}_0$. The semantics is based on CCS with priorities and value passing. To simplify the following notations, we define the standard priority of transitions to be 2. This means that whenever a priority is not given explicitly, we assume the transition to have this standard priority.

4.2.1. Sequential Composition

There is no basic notion of sequential composition of processes in CCS. To define a combinator $Before$ for sequential composition, we assume that processes must indicate their termination by a $\overline{done}$ transition. The combinator is then defined as follows:

$$P\ Before\ Q \stackrel{\text{def}}{=} (P[b/done] \mid b.Q) \setminus \{b\}$$
$$Done \stackrel{\text{def}}{=} \overline{done}.nil$$

where b is a new label.

4.2.2. Variables

$\mathcal{C}_0$ provides basic support for integer variables. An integer variable i is represented by a CCS process which stores the value of the variable and provides two channels for reading and writing this value:

$$Int^i(Val) \stackrel{\text{def}}{=} write^i(x).Int^i x\rangle + \overline{read}^i(val).Int^i\langle val\rangle$$

The $\overline{read}$ transition is used to read the value of the variable, and via the $write$ transition a new value is set to the variable. There are two possibilities how a variable can obtain a new value: either through the evaluation of an assignment or through the result of an external function call. Both expressions are translated to CCS processes, which must terminate with a $\overline{res}(x)$ transition providing their result as parameter x. In order to write this result to the variable, we define the following combinator:

$$P\ Into(x)\ Q \stackrel{\text{def}}{=} (P \mid res(x).Q) \setminus \{res\}$$

where Q is a process expression with a free variable x and no further res transitions. This combinator is then used with a process P, which evaluates an integer expression or an external function call and returns the value as an output transition over res, and a process Q, which takes the results and writes it to a variable.

Once we introduce the semantics of the Linux kernel API, more types of variables are possible. For a variable type τ and an instance χ of τ, we define an access sort $ACC(\tau, \chi)$. This set contains all labels that can be used to interact with the CCS process of the variable τ. For an integer variable i, the access sort is defined by:

$$ACC(\texttt{int}, i) = \{read^i, write^i\}$$

4.2.3. Commands and Expressions

The function $[\![\cdot]\!]$ translates a $\mathcal{C}_0$ program into a set of CCS processes. It is defined over the inductive structure of $\mathcal{C}_0$. Using the auxiliary combinators $Before$ and $Into(x)$, the translation of statements is pretty straightforward:

$$
\begin{aligned}
[\![X \texttt{ = } E]\!] &= [\![E]\!]\ Into(x)\ (\overline{write}^X(x).Done) \\
[\![S \texttt{ ; } S']\!] &= [\![S]\!]\ Before\ [\![S']\!] \\
[\![\texttt{if } E \texttt{ \{ } S \texttt{ \}}]\!] &= [\![E]\!]\ Into(x)\ (\text{if } x = 1 \text{ then } [\![\text{S}]\!] \text{ else } Done) \\
[\![\texttt{if } E \texttt{ \{ } S \texttt{ \} else \{ } S' \texttt{ \}}]\!] &= [\![E]\!]\ Into(x)\ (\text{if } x = 1 \text{ then } [\![\text{S}]\!] \text{ else } [\![\text{S'}]\!]) \\
[\![\texttt{while E \{ S \}}]\!] &= W, \text{ where } W \stackrel{\text{def}}{=} [\![E]\!]\ Into(x) \\
&\qquad\qquad (\text{if } x = 1 \text{ then } [\![S]\!]\ Before\ W \text{ else } Done) \\
[\![\texttt{void G()}]\!] &= \overline{call}^{\texttt{G}}.return^{\texttt{G}}.Done \\
[\![X \texttt{ = } F]\!] &= [\![F]\!]\ Into(x)\ (\overline{write}^X(x).Done)
\end{aligned}
$$

Where S and S' are statements, X is an integer variable, E is an integer expression, G is a function name and F is an external function call. For external function calls, the translation must be defined explicitly for each function. Note the main difference between both kinds of function calls: while an internal function call must terminate with a *done* transition, external function calls must return an integer value. Therefore, they terminate with a *res* transition to provide the return result. The translation of function declarations is defined as follows:

$$
\begin{aligned}
[\![\texttt{void G() \{ } S \texttt{ \}}]\!] &= W_{\text{G}}, \text{ where} \\
W_{\text{G}} &\stackrel{\text{def}}{=} call^{\text{G}}.[\![S]\!]\ Before\ \overline{return}^{\text{G}}.W_{\text{G}} \\
[\![D \texttt{ ; } D']\!] &= [\![D]\!] \mid [\![D']\!]
\end{aligned}
$$

Where D and D' are function declarations. A declaration of a function G is translated into a process that is activated by a handshake communication over the channel $call^{\text{G}}$. The process which calls this function, must then wait until it receives a signal on the channel $return^{\text{G}}$. The translation of integer expressions is defined in the following way:

$$
\begin{aligned}
[\![C]\!] &= \overline{res}(C) \\
[\![X]\!] &= read^X(x).\overline{res}(x) \\
[\![X \texttt{ + } C]\!] &= read^X(x).\overline{res}(x + C) \\
[\![X \texttt{ == } C]\!] &= read^X(x).(\text{if } x = C \text{ then } \overline{res}(1) \text{ else } \overline{res}(0))
\end{aligned}
$$

Here, C is an integer constant. The comparison of a variable and a constant is translated to a function that returns 1 if both values are equal and 0 otherwise.

Next, we define the translation of variable declarations:

$$\begin{aligned} [\![\texttt{int } X]\!] &= Int^X\langle 0\rangle \\ [\![\texttt{int } X = C]\!] &= Int^X\langle C\rangle \\ [\![D \ ; \ D']\!] &= [\![D]\!] \mid [\![D']\!] \end{aligned}$$

Where D and D' are arbitrary variable declarations. Due to the ANSI-C standard, an uninitialized global integer variable has the value 0.

4.2.4. Translating $\mathcal{C}_0$ Programs

Finally, we define the translation of a whole $\mathcal{C}_0$ program. Assume that a $\mathcal{C}_0$ program $\mathcal{P}$ consists of a nonempty sequence $\mathcal{P}_\mathcal{D}$ of variable declarations and a nonempty sequence $\mathcal{P}_\mathcal{F}$ of function declarations. We write $\langle \tau, \chi\rangle \in \mathcal{P}_\mathcal{D}$, if variable χ of type τ is defined in $\mathcal{P}_\mathcal{D}$. Furthermore, we write $f_{foo} \in \mathcal{P}_\mathcal{F}$, if foo is the name of a function declaration in $\mathcal{P}_\mathcal{F}$.

Let a $\mathcal{C}_0$ program be given. $\mathcal{P}_\mathcal{D}$ consists of m variable declarations, and $\mathcal{P}_\mathcal{F}$ consists of n function declarations. Furthermore, $f_{main} \in \mathcal{P}_\mathcal{F}$, which is the entry point of the program. The translation of $\mathcal{P}$ is then defined as follows:

$$[\![\mathcal{P}]\!] = ([\![\mathcal{P}_\mathcal{D}]\!] \mid [\![\mathcal{P}_\mathcal{F}]\!] \mid \overline{call}^{main}.nil) \setminus \Delta$$

with

$$\Delta = \bigcup_{1\leq i\leq m} ACC(\tau_i, \chi_i) \cup \{call^{f_i}, return^{f_i} \mid 1 \leq i \leq n\}$$

The label restriction set Δ restricts all variable access labels and all function call and return labels to be internal handshake actions.

4.3. CCS based Semantics of the Linux Kernel API

In this section, we introduce a semantics, based on CCS, for a subset of the Linux kernel API. This subset is explained in Section 2.2. The parts of the API which are concurrency related, are of major interest. Here, we omit the formalization of memory management and I/O communication. A semantics of these parts is presented in the next section, where we introduce an operational semantics for all supported parts of the Linux kernel API.

For each supported subsystem of the Linux kernel API, we define a translation function. This function translates the C-commands of the API to CCS process definitions. Furthermore, we specify LTL formulas defining the behaviour of programs that make correct use of the Linux kernel API. If one of these formulas is not true for a transition, then the corresponding program has a bug.

4.3.1. Semaphores and Spinlocks

The way semaphores can be modeled as a CCS process is very simple. For a semaphore x, we define a process Sem^x as follows:

$$Sem^x \stackrel{\text{def}}{=} down^x.up^x.Sem^x$$

Assume that there are two CCS processes P and Q that run in parallel to the semaphore process Sem^x. If P wants to lock the semaphore x it must perform the transition $\overline{down}^x$. This results in a handshake communication with the semaphore process. If the process Q now tries to acquire the same semaphore, it cannot continue because there is no counterpart for the locking transition. Only when P unlocks the semaphore using the transition $\overline{lock}^x$, the semaphore process will be restored and Q may continue with locking the semaphore.

The translations of functions for initializing, locking and unlocking semaphores are defined as follows:

$$\begin{aligned}
[\![\texttt{struct semaphore sem}]\!] &= init^{\texttt{sem}}.Sem^{\texttt{sem}} \\
[\![\texttt{DECLARE_MUTEX(sem)}]\!] &= Sem^{\texttt{sem}} \\
[\![\texttt{init_MUTEX(\&sem)}]\!] &= \overline{init}^{\texttt{sem}}.Done \\
[\![\texttt{down(\&sem)}]\!] &= \overline{down}^{\texttt{sem}}.Done \\
[\![\texttt{down_interruptible(\&sem)}]\!] &= (\overline{down}^{\texttt{sem}}.\overline{res}(0) + \overline{res}(-1)).Done \\
[\![\texttt{down_trylock(\&sem)}]\!] &= (\overline{down}^{\texttt{sem}}.\overline{res}(0) + \tau_3.\overline{res}(-1)).Done \\
[\![\texttt{up(\&sem)}]\!] &= \overline{up}^{\texttt{sem}}.Done
\end{aligned}$$

If a semaphore variable is not declared with the macro `DECLARE_MUTEX`, it must be initialized with the function `init_MUTEX` before use. Therefore, the corresponding semaphore process is guarded with an *init* transition. The locking function `down_interruptible` may fail when waiting for the semaphore. `down_trylock` tries to lock the semaphore and fails immediately if it is not available. This is modeled using a τ_3 transition with a lower priority than the transition for locking the semaphore has. If a handshake communication over the channel $down^x$ is possible, the process has to perform this one. In this case, it returns 0 to indicate the successful locking of the semaphore. Otherwise, the process performs the τ_3 transition and returns -1, because the semaphore has been locked by another process.

The main difference between semaphores and spinlocks is the way how the waiting is done if the semaphore or spinlock is not available. Because we do not formalize the difference between sleeping or busy waiting, semaphores and spinlocks are quite similar in their formalization. The CCS process for a spinlock x is defined in the same way like semaphores are defined:

$$Spin^x \stackrel{\text{def}}{=} down^x.up^x.Spin^x$$

The translations of spinlock related functions are also quite similar:

$$\begin{aligned}
[\![\texttt{spinlock_t lock}]\!] &= init^{\texttt{lock}}.Spin^{\texttt{lock}} \\
[\![\texttt{spinlock_t lock} = \texttt{SPIN_LOCK_UNLOCKED}]\!] &= Spin^{\texttt{lock}} \\
[\![\texttt{spin_lock_init(\&lock)}]\!] &= \overline{init}^{\texttt{lock}}.Done \\
[\![\texttt{spin_lock(\&lock))}]\!] &= \overline{down}^{\texttt{lock}}.Done \\
[\![\texttt{spin_unlock(\&lock))}]\!] &= \overline{up}^{\texttt{lock}}.Done
\end{aligned}$$

The access sorts for semaphores and spinlocks are defined by:

$$ACC(\texttt{struct semaphore}, x) = ACC(\texttt{spinlock\_t}, x) = \{init^x, down^x, up^x\}$$

We define three specifications of correct semaphore and spinlock usage:

- A semaphore or a spinlock, respectively, always has to be initialized before it is used. This is expressed by the following formula:

  $$\mathrm{G}\ \neg(init^x \wedge \overline{down}^x)$$

  If an output transition on channel $down^x$ is available, there is a process trying to acquire the lock x. If, at the same time, the input transition $init^x$ is available, then the lock x has not been initialized. Therefore, both transitions are not allowed to occur simultaneously.

- Unlocking a locked semaphore or spinlock is not allowed:

  $$\mathrm{G}\ \neg(down^x \wedge \overline{up}^x)$$

  If both, $down^x$ and $\overline{up}^x$ are available at the same time, there is a process which wants to unlock a lock x, which has not been locked before (otherwise the input transition $down^x$ would not be available).

- A process it not allowed to lock a semaphore or spinlock if it has already been locked by the same process. For a general formula defining this property, we would need to store the information about which process has locked the semaphore or spinlock, respectively. If we restrict the situation to the case of only one execution process (hence there is no concurrency), we can define the following formula:

  $$\mathrm{G}\ \neg(up^x \wedge \overline{down}^x)$$

  If the transition up^x is available, we know that the lock x is locked in this state. If there is only one process running and it makes an output transition on channel $down^x$, then we know that this process tries to lock x, which has been locked before by the same process.

$$StartWakeup^{\mathtt{x}} \mid wakeupQueue_0^{\mathtt{x}}.P \mid wakeupQueue_0^{\mathtt{x}}.Q \mid \overline{startWakeup}^{\mathtt{x}}.R$$
$$\downarrow \tau_2^{startWakeup^{\mathtt{x}}}$$
$$WakeupQueue^{\mathtt{x}} \mid wakeupQueue_0^{\mathtt{x}}.P \mid wakeupQueue_0^{\mathtt{x}}.Q \mid R$$
$$\downarrow \tau_0^{wakeupQueue^{\mathtt{x}}}$$
$$WakeupQueue^{\mathtt{x}} \mid P \mid wakeupQueue_0^{\mathtt{x}}.Q \mid R$$
$$\downarrow \tau_0^{wakeupQueue^{\mathtt{x}}}$$
$$WakeupQueue^{\mathtt{x}} \mid P \mid Q \mid R$$
$$\downarrow \tau_1$$
$$StartWakeup^{\mathtt{x}} \mid P \mid Q \mid R$$

Figure 4.2.: Transitions of awaking processes from a wait queue

4.3.2. Wait Queues

A process may use a wait queue to sleep on until a condition holds. Another process must then request the wait queue to awake all waiting processes. We model the waiting on a wait queue x with a special transition $wakeupQueue^x$. If a process wants to sleep on the wait queue x, it just needs to wait for the complementary transition. The input transition over this channel is sent by a wake up process. When this process is started, it sends the $\overline{wakeupQueue}^x$ transition as many times as there are processes waiting on the wait queue x. This way, the waiting processes perform a handshake communication over this channel and may continue. The wake-up method for a wait queue x consists of the following two processes:

$$WakeupQueue^{\mathtt{x}} \stackrel{\text{def}}{=} \overline{wakeupQueue_0}^{\mathtt{x}}.WakeupQueue^{\mathtt{x}} + \tau_1.StartWakeup^{\mathtt{x}}$$
$$StartWakeup^{\mathtt{x}} \stackrel{\text{def}}{=} startWakeup^{\mathtt{x}}.WakeupQueue^{\mathtt{x}}$$

In order to start waking up on a wait queue x, a process has to perform the transition $\overline{startWakeup}^x$. After this, the process $WakeupQueue^x$ is started. It implements a loop, creating $\overline{wakeupQueue}_0^x$ transitions until there is no further complementary transition. This wakes up all waiting processes. If there are no more waiting processes, $WakeupQueue^x$ is forced to make a τ_1 transition. This way, the process $StartWakeup^x$ is restored. An exemplary transition for two waiting processes is shown in Figure 4.2.

When defining the translation function for wait queues, we make the assumption that the waiting condition only consists of an integer variable. If this variable evaluates to 1, the condition is fulfilled. We use this assumption to simplify the

formalization of wait queues. This simplification does not reduce the applicability of wait queues, since the full condition can be evaluated before the wake up function is called. The result is then stored in a temporary variable, which is the only waiting condition of the wait queue. The translation function for the supported wait queue functions is then defined as follows:

$$\begin{aligned}
[\![\texttt{wait_queue_head_t queue}]\!] &= init^{\texttt{queue}}.StartWakeup^{\texttt{queue}} \\
[\![\texttt{DECLARE_WAIT_QUEUE_HEAD(queue)}]\!] &= StartWakeup^{\texttt{queue}} \\
[\![\texttt{init_waitqueue_head(\&queue)}]\!] &= \overline{init}^{\texttt{queue}}.Done \\
[\![\texttt{wait_event(queue, var)}]\!] &= W' \\
[\![\texttt{wait_event_interruptible(queue, var)}]\!] &= W'' \\
[\![\texttt{wake_up(\&queue)}]\!] &= \overline{startWakeup}^{\texttt{queue}}.Done \\
[\![\texttt{wake_up_interruptible(\&queue)}]\!] &= \overline{startWakeup}^{\texttt{queue}}.Done
\end{aligned}$$

where W' and W'' are new process names and defined in the following way:

$$\begin{aligned}
W' &\stackrel{\text{def}}{=} read^{\texttt{var}}(x)(\text{if } x = 1 \text{ then } Done \text{ else } wakeupQueue_0^{\texttt{queue}}.W') \\
W'' &\stackrel{\text{def}}{=} read^{\texttt{var}}(x)(\text{if } x = 1 \text{ then } \overline{res}(0) \text{ else } wakeupQueue_0^{\texttt{queue}}.W'' + \overline{res}(-1))
\end{aligned}$$

A wait queue variable, which is not declared with the `DECLARE_WAIT_QUEUE_HEAD` macro, must be initialized with the function `init_waitqueue_head` before use. Therefore, the corresponding wait queue process is guarded with an *init* transition. After a handshake communication over this channel, the process $StartWakeup^{\texttt{queue}}$ is available. Whenever a process wants to go to sleep, the condition is checked first. If the condition evaluates to true, nothing will happen. Otherwise, the process waits for the wake up transition $\overline{wakeupQueue}$. When this transition is available, the condition is checked again. The function `wait_event_interruptible` might also fail. In this case, the process does not wait and returns the value -1 to indicate the failure. The access sort for wait queues is defined as follows:

$$ACC(\texttt{wait\_queue\_head\_t}, x) = \{init^x, wakeupQueue^x, startWakeup^x\}$$

We define through an LTL formula, that a wait queue x is never allowed to be used if it has not been initialized correctly before:

$$\mathrm{G}\ \neg(init^x \wedge (\overline{wakeupQueue}^x \vee \overline{startWakeup}^x))$$

This formula expresses that in a correct state, it is not allowed to have both the initialization transition $init^x$, denoting that wait queue x is not initialized, and other transitions available on the same wait queue.

4.3.3. Tasklets

The tasklet subsystem is a method for deferring work to happen later. When starting a tasklet, its handler function is scheduled to be called in the near future. A feature of tasklets is that they can be disabled and re-enabled. A tasklet won't be executed until it is enabled as many times as it has been disabled. Furthermore, a tasklet can reschedule itself.

For better readability, the formalization of tasklets is split into several CCS processes. $Tasklet^t$ is the main process for a tasklet variable t:

$$Tasklet^t \stackrel{\text{def}}{=} setFunc^t(x).Tasklet^t\langle x\rangle$$
$$Tasklet^t(f) \stackrel{\text{def}}{=} TaskletFunc^t\langle f\rangle \mid TaskletCount^t\langle 0\rangle \mid TaskletWait^t$$

Before a tasklet can be used, it has to be initialized with a handler function. In the formalization, this is done via the *setFunc* transition. If a handler function is set to the process $Tasklet^t$, it continues as the parallel composition of the three processes $TaskletFunc^t$, $TaskletCount^t$ and $TaskletWait^t$:

$$TaskletFunc^t(f) \stackrel{\text{def}}{=} setFunc^t(x).Tasklet^t\langle x\rangle + \overline{getFunc}^t(f).Tasklet^t\langle f\rangle$$
$$TaskletCount^t(c) \stackrel{\text{def}}{=} incCount^t.TaskletCount^t\langle c'\rangle + decCount^t.(\text{if } c = 0 \text{ then } TaskletCount^t\langle 0\rangle \text{ else } TaskletCount^t\langle c''\rangle) \text{ with } c' = c+1 \text{ and } c'' = c-1$$
$$TaskletWait^t \stackrel{\text{def}}{=} scheduleTasklet^t.TaskletSchedule^t$$

The first process saves the name of the tasklet's handler function. Using the channels *setFunc* and *getFunc*, it is possible to read the function name or to set a new one. *TaskletCount* is used to store the number of times the tasklet has been disabled. Whenever the process can make a handshake over the channel *incCount*, the number is increased. When decreasing the number over the channel *decCount*, it must first be checked if the number is not zero. If this is the case, the decrease is ignored. *TaskletWait* waits for an input signal on the channel *scheduleTasklet* in order to start the process *TaskletSchedule*:

$$TaskletSchedule^t \stackrel{\text{def}}{=} getCount^t(x).(\text{if } x = 0 \text{ then } TaskletExec^t \text{ else } TaskletSchedule^t)$$
$$TaskletExec^t \stackrel{\text{def}}{=} getFunc^t(f).\overline{call}^f.return^f.TaskletWait^t$$

TaskletSchedule checks if the tasklet has been enabled as many times as it has been disabled. If this is the case, the handler function of the tasklet is executed

within process $TaskletExec$. Otherwise, $TaskletSchedule$ waits until the counter is down to zero to start the tasklet's handler function.

The translation function for the supported tasklet functions is defined in the following way:

$$\begin{aligned}
[\![\texttt{struct tasklet_struct tasklet}]\!] &= Tasklet^{\texttt{tasklet}} \\
[\![\texttt{DECLARE_TASKLET(tasklet, func, data)}]\!] &= Tasklet^{\texttt{tasklet}}\langle\texttt{func}\rangle \\
[\![\texttt{tasklet_init(\&tasklet, func, data)}]\!] &= \overline{setFunc}^{\texttt{tasklet}}(\texttt{func}).Done \\
[\![\texttt{tasklet_schedule(\&tasklet)}]\!] &= \overline{scheduleTasklet}^{\texttt{tasklet}}.Done \\
[\![\texttt{tasklet_enable(\&tasklet)}]\!] &= \overline{decCount}^{\texttt{tasklet}}.Done \\
[\![\texttt{tasklet_disable(\&tasklet)}]\!] &= \overline{incCount}^{\texttt{tasklet}}.Done
\end{aligned}$$

The access sort for tasklets is defined as follows:

$$\begin{aligned}
ACC(\texttt{struct tasklet_struct}, x) = \{&setFunc^x, getFunc^x, incCount^x, \\
&decCount^x, scheduleTasklet^x\}
\end{aligned}$$

We define through an LTL formula, that a tasklet structure x is never allowed to be used if it has not already been initialized correctly:

$$\mathrm{G}\ \neg(\overline{scheduleTasklet}^x \wedge \neg\exists f.\overline{getFunc}^x(f))$$

If an output transition on channel $scheduleTasklet^x$ is available, then there is a process using the tasklet x. If, at this time, there is no transition available on the channel $getFunc^x$, then tasklet x has not been initialized. For an initialized tasklet, it is always possible to read its handler function.

4.3.4. Work Queues

We only model the shared work queue, which is provided by the Linux kernel API. In general, a device driver may create a private work queue, but this is used in very few device drivers only.

There is no explicit CCS process definition for the shared work queue. Instead, we directly model the work structures, which are waiting for the signal $\overline{scheduleWork}$ to execute their handler function. For a work structure w, the following processes are defined:

$$\begin{aligned}
Work^w(f) &\stackrel{\text{def}}{=} setFunc^w(x).Work^w\langle x\rangle + \\
&\qquad scheduleWork^w.WorkExec^w\langle f\rangle \\
WorkExec^w(f) &\stackrel{\text{def}}{=} (\overline{call}^f.return^f.\overline{exitCatch_1}^w.Work^w\langle f\rangle) \mid \\
&\qquad WorkCatchCalls^w \\
WorkCatchCalls^w &\stackrel{\text{def}}{=} scheduleWork^w.CatchWorkCalls^w + \\
&\qquad exitCatch_1^w.nil
\end{aligned}$$

The main process $Work$ makes it possible to set a new handler function via the $setFunc$ channel or to schedule it via $scheduleWork$. In the latter case, the work process continues as process $WorkExec$. It executes the handler function and, in parallel, catches and ignores all further scheduling requests of the work structure. This is due to the semantics of the Linux kernel API. It defines that whenever a work structure is scheduled but running at the moment, the scheduling request terminates immediately without causing any effect.

The translation function for the supported work queue functions is defined in the following way:

$$\begin{aligned}
[\![\texttt{struct work_struct work}]\!] &= setFunc(x)^{\texttt{work}}.Work^{\texttt{work}}\langle x\rangle \\
[\![\texttt{DECLARE_WORK(work, func, data)}]\!] &= Work^{\texttt{work}}\langle\texttt{func}\rangle \\
[\![\texttt{INIT_WORK(\&work, func, data)}]\!] &= setFunc^{\texttt{work}}(\texttt{func}).Done \\
[\![\texttt{PREPARE_WORK(\&work, func, data)}]\!] &= setFunc^{\texttt{work}}(\texttt{func}).Done \\
[\![\texttt{schedule_work(\&work)}]\!] &= (\overline{scheduleWork}^{\texttt{work}}.\overline{res}(1) + \overline{res}(0)).\\
&\qquad Done
\end{aligned}$$

Similar to tasklets, the macro `DECLARE_WORK` can be used to initialize a work data structure together with the work handler function. The handler function can be changed with the macros `INIT_WORK` and `PREPARE_WORK`. `schedule_work` is used to schedule the handler function of a work structure. In contrast to tasklets, this function may fail. Therefore, it has an integer return value. The access sort for work queues is defined as follows:

$$ACC(\texttt{struct work\_struct}, x) = \{setFunc^x, scheduleWork^x, exitCatch^x\}$$

Similar to tasklets, we define an LTL formula denoting that a work structure x is never allowed to be used if it has not been initialized correctly before:

$$\mathrm{G}\ \neg(\overline{scheduleWork}^x \wedge \neg scheduleWork^x)$$

The only situation when both $\overline{scheduleWork}^x$ and $schedukeWork^x$ are available, is when a work structure x is used without having been initialized before. Note that whenever a work structure x has been scheduled, there is the CCS process $WorkCatchCalls$, providing the $scheduleWork^x$ transition.

4.3.5. Interrupts

The formalization of the interrupt handling process can be simplified if two assumptions are made: first, we formalize only one device driver at a time. Therefore, we do not care about the concurrent behaviour of different device drivers. Second, we assume that a device driver may register at most one interrupt handler function for each interrupt line. The Linux kernel API does not prohibit to install more than one interrupt handler for one interrupt line, but from our point of view, this would

not make much sense. With these simplifications, it is clear that we do not need to care about the difference between shared and private interrupt lines.

In this scenario, requesting an interrupt line can be reduced to providing a function handler to a special process using the transition $requestIrqHandler$:

$$\begin{aligned} RequestIrq^i &\stackrel{\text{def}}{=} requestIrqHandler^i(f).IrqHandler^i\langle f\rangle \\ IrqHandler^i(f) &\stackrel{\text{def}}{=} startIrq_0^i.\overline{call}^f.return^f.\overline{finishedIrq}^i.IrqHandler^i\langle f\rangle + \\ &\qquad releaseIrqHandler^i.RequestIrq^i \end{aligned}$$

After requesting the interrupt line, the process $IrqHandler$ is available. It waits for an input transition on channel $startIrq^i$ in order to start the interrupt function handler of the i-th interrupt line. The transitions over this channel have the highest priority 0 to indicate that every other computations are interrupted when an interrupt request is generated. If $IrqHandler$ receives the signal $releaseIrqHandler$, it continues as process $RequestIrq$. Hence, the interrupt line is freed.

The $startIrq$ transitions, which start the execution of the corresponding interrupt handler function, are generated by an interrupt generating process:

$$IrqGenerator^i \stackrel{\text{def}}{=} \overline{startIrq_0}^i.finishedIrq^i.IrqGenerator^i$$

This process reflects the intuition that interrupts are generated by some hardware. Now we can combine all the interrupt related processes into one general interrupt control process:

$$\begin{aligned} IrqHandling \stackrel{\text{def}}{=}\ & RequestIrq^0 \mid \cdots \mid RequestIrq^{15} \mid \\ & IrqGenerator^0 \mid \cdots \mid IrqGenerator^{15} \end{aligned}$$

There are only two functions in the Linux kernel API for which we need to define a translation: `request_irq` for requesting an interrupt line, and `free_irq` for releasing it. Both translations are quite simple and results in sending a transition over the corresponding channel. Note that requesting an interrupt line may also fail:

$$\begin{aligned} [\![\texttt{request\_irq(irq, handler, ...)}]\!] &= \overline{requestIrqHandler}^{\texttt{irq}}(\texttt{handler}).\overline{res}(0) + \\ &\qquad \overline{res}(-1) \\ [\![\texttt{free\_irq(irq, devid)}]\!] &= \overline{releaseIrqHandler}^{\texttt{irq}}.Done \end{aligned}$$

The access sort for the interrupt handling processes is defined as follows:

$$\begin{aligned} ACC_{irq} = \{&requestIrqHandler^i, releaseIrqHandler^i, \\ &startIrq^i, finishedIrq^i \mid 0 \leq i \leq 15\} \end{aligned}$$

4.3.6. Translating Device Drivers

In Section 4.2.4, the translation of $\mathcal{C}_0$ programs is shown. This translation function cannot be used directly on device driver's code because of the following reasons:

- Device drivers do not have a main function like stand-alone C programs. Instead, they provide an initialization function and a cleanup function.
- The translation of $\mathcal{C}_0$ programs restricts the function calls to the program where the functions are defined. But device drives are written to provide a set of functions to other programs.
- The main interrupt handling process must run in parallel to the translation result of a device driver.

Therefore, we redefine the translation function $[\![\cdot]\!]$ for device drivers written in $\mathcal{C}_0$.

A device driver $\mathcal{C}_0$ program $\mathcal{P}$ consists of a possibly empty sequence $\mathcal{P}_D$ of variable declarations and a nonempty sequence $\mathcal{P}_\mathcal{F}$ of function declarations. Assume that the number of variable declarations in $\mathcal{P}_\mathcal{D}$ is m, and the number of function declarations in $\mathcal{P}_\mathcal{F}$ is n. The translation of $\mathcal{P}$ is then defined as follows:

$$[\![\mathcal{P}]\!] = ([\![\mathcal{P}_\mathcal{D}]\!] \mid [\![\mathcal{P}_\mathcal{F}]\!] \mid IrqHandling) \setminus \Delta$$

with

$$\Delta = \bigcup_{1 \leq i \leq m} ACC(\tau_i, \chi_i) \cup ACC_{irq}$$

In contrast to the definition of $[\![\cdot]\!]$ for general $\mathcal{C}_0$ programs, the labels are only restricted to the variable and interrupt access labels. The function call and return transitions are available to processes outside of $[\![\mathcal{P}]\!]$.

4.3.7. Execution Models for Device Drivers

The translation definition for device drivers is not complete in terms of no execution model having been defined for it yet. An execution model simulates a user space program using a device driver. In this section, we introduce two different execution models for the device driver translation. The first execution model allows a restricted concurrency, whereas the second execution model is strictly sequential.

Concurrent Model

A full concurrent model, where the interleaving between all functions executed in parallel is allowed, would be closest to the reality. But it would have the disadvantage that, because of the state explosion in model checking, it would not be applicable for further reasoning. Therefore, we restrict the possibility of process interleaving in the following way: All provided driver functions are executed in sequential order. This simulates a scenario where just one user space program makes

use of the device driver. The driver's functions can run in parallel to tasklet, work queues, and interrupt handler functions.

The concurrent model consists of five processes. The initialization process $Init$ calls the init function of the device driver and starts the function execution process $Exec$. This process calls the process $Func$ arbitrary times. $Func$ chooses one of the available driver's functions, executes it and waits until the function terminates. Afterwards, the $Exit$ process is called, which just executes the cleanup function of the device driver. Finally, $Model$ is the parallel composition of all of these processes:

$$
\begin{aligned}
Init &\stackrel{\text{def}}{=} \overline{call}^{f_{init}}.return^{f_{init}}.\overline{startExec}.nil \\
Exec &\stackrel{\text{def}}{=} startExec.(\overline{callFunc}.Exec + \overline{exit}.nil) \\
Func &\stackrel{\text{def}}{=} callFunc.(\\
&\qquad \overline{call}^{f_3}.return^{f_3}.\overline{startExec}.Func + \\
&\qquad\qquad \vdots \\
&\qquad \overline{call}^{f_i}.return^{f_i}.\overline{startExec}.Func) \\
Exit &\stackrel{\text{def}}{=} exit.\overline{call}^{f_{exit}}.return^{f_{exit}}.nil \\
Model &\stackrel{\text{def}}{=} (Init \mid Exec \mid Func \mid Exit) \setminus \{startExec, callFunc, exit\}
\end{aligned}
$$

Without loss of generality, we assume that $f_{init} = f_1$ and $f_{exit} = f_2$ holds. Then, we finally combine the translation of a device driver $\mathcal{P}$ and the concurrent execution model process $Model$ to one CCS process:

$$ConcurrentDriverExec \stackrel{\text{def}}{=} (\llbracket \mathcal{P} \rrbracket \mid Model) \setminus \Delta$$

with

$$\Delta = \{call^{f_i}, return^{f_i} \mid 1 \leq i \leq n\}$$

Sequential Model

In the sequential model, we need a way to forbid the concurrent execution of driver's functions and the handler functions of tasklets, wait queues and interrupts. In order to accomplish this, we introduce a global lock that must be held by a process to execute one of these functions:

$$Lock \stackrel{\text{def}}{=} downLock.upLock.Lock$$

The process definitions of the sequential model are changed correspondingly. Before a function of the device driver may be called, the global lock has to be requested

successfully:

$$
\begin{aligned}
Init &\stackrel{\text{def}}{=} \overline{downLock}.\overline{call}^{f_{init}}.return^{f_{init}}.\overline{upLock}.\overline{startExec}.nil \\
Exec &\stackrel{\text{def}}{=} startExec.(\overline{callFunc}.Exec + \overline{exit}.nil) \\
Func &\stackrel{\text{def}}{=} callFunc.(\\
&\qquad \overline{downLock}.\overline{call}^{f_3}.return^{f_3}.\overline{upLock}.\overline{startExec}.Func + \\
&\qquad\qquad \vdots \\
&\qquad \overline{downLock}.\overline{call}^{f_i}.return^{f_i}.\overline{upLock}.\overline{startExec}.Func) \\
Exit &\stackrel{\text{def}}{=} exit.\overline{downLock}.\overline{call}^{f_{exit}}.return^{f_{exit}}.\overline{upLock}.nil \\
Model &\stackrel{\text{def}}{=} (Init \mid Exec \mid Func \mid Exit) \setminus \{startExec, callFunc, exit\}
\end{aligned}
$$

Correspondingly, the main CCS processes for tasklets, work queues and interrupt handlers must be redefined in the following way:

$$
\begin{aligned}
TaskletExec^t &\stackrel{\text{def}}{=} getFunc^t(f).\overline{downLock}.\overline{call}^f.return^f.\overline{upLock}.TaskletWait^f \\
WorkExec^{\mathtt{w}}(f) &\stackrel{\text{def}}{=} (\overline{downLock}.\overline{call}^f.return^f.\overline{exitCatch}_1^{\mathtt{w}}.\overline{upLock}.Work^{\mathtt{w}}\langle f\rangle) \mid \\
&\qquad\qquad WorkCatchCalls^{\mathtt{t}} \\
IrqHandler^i(f) &\stackrel{\text{def}}{=} (startIrq_0^i.\overline{downLock}.\overline{call}^f.return^f.\overline{finishedIrq}^i.\overline{upLock}. \\
&\qquad\qquad IrqHandler^i\langle f\rangle) + \\
&\qquad releaseIrqHandler^i.RequestIrq^i
\end{aligned}
$$

Finally, we combine the translation of a device driver $\mathcal{P}$ and the sequential execution model process $Model$ to one CCS process:

$$SequentialDriverExec \stackrel{\text{def}}{=} (\llbracket \mathcal{P} \rrbracket \mid Model) \setminus \Delta$$

with

$$\Delta = \{call^{f_i}, return^{f_i} \mid 1 \leq i \leq n\} \cup \{downLock, upLock\}$$

4.3.8. Example

In this section, we show an example of translating a device driver's code to a set of CCS processes. Showing the translation on a real device driver's source code would not be possible, because the translation of a program written in C to a $\mathcal{C}_0$ program would produce a result which would be far too large to be translated manually. Therefore, we only consider an exemplary skeleton of a device driver here. Its $\mathcal{C}_0$ source code is shown in Figure 4.3. This device driver provides a small set of functions, which real device drivers usually implement, too. The function

```
int result1 = -1;
int result2 = -1;
int open = 0;
int readdata = 0;
int choice = 0;
spinlock_t lock = SPIN_LOCK_UNLOCKED;
DECLARE_WAIT_QUEUE_HEAD(queue);

void devopen() {
  if (result2 == 0) {
    spin_lock(&lock);
    open();
    open = 1;
    spin_unlock(&lock);
  }
}

void devclose() {
  if (open == 1) {
    spin_lock(&lock);
    close();
    open = 0;
    spin_unlock(&lock);
  }
}

void devread() {
  if (open == 1) {
    wait_event(queue, readdata);
    read();
    readdata = 0;
  }
}

void devwrite() {
  if (open == 1)
    write();
}

void irqhandler() {
  getdata();
  readdata = 1;
  wake_up(&queue);
}

void init() {
  result1 = register_dev();
  if (result1 == 0) {
    result2 = request_irq(7,
       irqhandler, FLAG, NAME, 0);
    if (result2 == -1)
      cleanup();
  }
}

void cleanup() {
  if (result1 == 0) {
    unregister_dev();
    if (result2 == 0)
      free_irq(7, 0);
  }
}
```

Figure 4.3.: $\mathcal{C}_0$ program of a simple device driver

`init` registers the device driver and requests interrupt line 7. If `result1` is 0, the interrupt line has been requested successfully and can be used by the device driver. Removing the device driver from the system and releasing the interrupt line is done through function `cleanup`. The functions `devopen` and `devclose` make use of a spinlock to guard the opening and closing functions of the device. The presented device driver allows processes to read and write on the device. If a process starts the reading function `devread`, it is possible that there is no data to read at that time. This is the case if `readdata` is zero. A wait queue is used to wait as long as there is no new data available. The interrupt handler is executed as soon as new data arrives. Its only job is to receive the data and to wake up all processes waiting to read data.

The translation of the driver's source code and a corresponding concurrent execution model are shown in Appendix B. To make the results more legible, we assume that the function `register_dev` returns either 0 or -1. In general, if the function

fails, there are several negative return values possible. The functions `open`, `close`, `read` and `write` are not further specified and therefore translated to an atomic instruction. The device driver declares eight variables. Their translation is as follows:

$$\begin{aligned} Var \stackrel{\text{def}}{=} & \; Int^{\texttt{result1}}\langle -1\rangle \mid Int^{\texttt{result2}}\langle -1\rangle \mid Int^{\texttt{open}}\langle 0\rangle \mid \\ & Int^{\texttt{readdata}}\langle 0\rangle \mid Int^{\texttt{choice}}\langle 0\rangle \mid \\ & Spin^{\texttt{lock}} \mid WakeupQueue^{\texttt{queue}} \mid StartWakeup^{\texttt{queue}} \end{aligned}$$

4.4. Operational Model for the Linux Kernel API

In this section, we define a sequential operational model for the Linux kernel API. In the following, let $\mathcal{B} = \{\bot, \top\}$ be the boolean set, where $\bot$ corresponds to *false* and $\top$ corresponds to *true*. Furthermore, let Sem, $Spin$, $Wait$, $Tasklet$, $Work$ and Irq denote the values of semaphores, spinlocks, wait queues, tasklets, work queues and interrupts. By convention, we assume $p \in Sem$, $l \in Spin$, $q \in Wait$, $\{t, t_1, t_2, \dots\} \subset Tasklet$, and $\{w, w_1, w_2, \dots\} \subset Work$. They are defined in the following way:

$$\begin{aligned} Sem &= \mathcal{B} \times \mathcal{B} && \text{with } (init, lock) \in Sem \\ Spin &= \mathcal{B} \times \mathcal{B} && \text{with } (init, lock) \in Spin \\ Wait &= \mathcal{B} && \text{with } (init) \in Wait \\ Tasklet &= \mathbb{N} \times \mathbb{N} \times \mathcal{B} && \text{with } (pc, cnt, sched) \in Tasklet \\ Work &= \mathbb{N} \times \mathcal{B} && \text{with } (pc, sched) \in Work \\ Irq &= \mathbb{N} \times \mathcal{B} && \text{with } (pc, req) \in Irq \end{aligned}$$

The set of variables of a specific type X is denoted with V_X. The set of states is defined by:

$$\begin{aligned} \Sigma = & (V_{\mathbb{Z}} \mapsto \mathbb{Z}) \times (V_{Sem} \mapsto Sem) \times (V_{Spin} \mapsto Spin) \times \\ & (V_{Wait} \mapsto Wait) \times (V_{Tasklet} \mapsto Tasklet) \times (V_{Work} \mapsto Work) \times \\ & (\mathbb{N} \mapsto Irq) \times (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \times (\mathbb{N} \mapsto \mathbb{N}) \times \{proc, irp\} \end{aligned}$$

with

$$(\sigma_{\mathbb{Z}}, \sigma_{sem}, \sigma_{spin}, \sigma_{wait}, \sigma_{tasklet}, \sigma_{work}, irq, ioport, pc, reg, context) \in \Sigma$$

$\sigma_{\mathbb{Z}}$ is a mapping from integer variables to their values. irq maps an interrupt line number to its value. This function is only defined for the values 0 to 15. $ioport$ is a pair of numbers representing the requested I/O ports. pc is the program counter of the next instruction. In the following, we use the function PC for defining for each

$$\frac{}{s[\![\texttt{struct semaphore p}]\!]_{cmd} s\langle p.init, \bot \mid p.lock, \bot\rangle} \quad \text{(sem-decl-1)}$$

$$\frac{}{s[\![\texttt{DECLARE\_MUTEX(p)}]\!]_{cmd} s\langle p.init, \top \mid p.lock, \bot\rangle} \quad \text{(sem-decl-2)}$$

$$\frac{s \models p.init = \bot}{s[\![\texttt{init\_MUTEX(\&p)}]\!]_{cmd} s\langle p.init, \top\rangle} \quad \text{(sem-init)}$$

$$\frac{s \models p.init = \bot \wedge p.lock = \bot \wedge context = proc}{s[\![\texttt{down(\&p)}]\!]_{cmd} s\langle p.lock, \top\rangle} \quad \text{(sem-down)}$$

$$\frac{s \models p.init = \bot \wedge p.lock = \bot \wedge context = proc}{s[\![\texttt{i := down\_interruptible(\&p)}]\!]_{cmd} s\langle p.lock, \top \mid i, 0\rangle} \quad \text{(sem-down-irp-1)}$$

$$\frac{s \models p.init = \bot \wedge p.lock = \bot \wedge context = proc}{s[\![\texttt{i := down\_interruptible(\&p)}]\!]_{cmd} s\langle i, -1\rangle} \quad \text{(sem-down-irp-2)}$$

$$\frac{s \models p.lock = \top}{s[\![\texttt{up(\&p)}]\!]_{cmd} s\langle p.lock, \bot\rangle} \quad \text{(sem-up)}$$

Figure 4.4.: The operational semantics for semaphores

function name its corresponding program counter. *reg* is a set of further registers. They can be used to store other global settings, like a return program counter when a function is called. In what follows, we omit the usage of the registers to make the definitions more legible. *context* denotes the context of the current execution. This can either be the context of a process or the interrupt context.

In the notation of the rules, we make use of the symbols $[\![]\!]_X$ and $[\![]\!]_{cmd}$. Symbol $[\![]\!]_X$ denotes a function from states and expressions to values of type X. Symbol $[\![]\!]_{cmd}$ denotes a function from states and expressions to states. Both are formally defined as follows:

$$[\![]\!]_X : (XExpression \times \Sigma) \mapsto X$$
$$[\![]\!]_{cmd} : (Command \times \Sigma) \mapsto \Sigma$$

We use $s\langle a, x\rangle$ to denote the state equal to s except for variable a having value x. If a has a field named b, then $s\langle a.b, x\rangle$ denotes the state that is equal to s except for $a.b$ being x.

4.4.1. Semaphores and Spinlocks

We model semaphores and spinlocks, respectively, as pairs of boolean variables. The first element denotes whether the data structure is initialized. The second element models the lock. If that element is true, then the semaphore or spinlock

is locked. Before a semaphore or spinlock can be locked, it must be checked if it is initialized and not already locked. Furthermore, semaphores are only allowed to be used in the context of a process. The operational semantics for semaphores is shown in Figure 4.4. Spinlocks are defined in the same way.

4.4.2. Wait Queues

As explained in Section 4.3.7, in a sequential model, we cannot make use of wait queues. To be able to reason about device drivers that make use of them, we assume that the wake-up condition holds the moment the process should go to sleep on the queue. For this case, the Linux kernel API defines that the wait command does not have any effects. In the C implementation of the operational model, we make use of the possibility of the model checker SatAbs to ignore all execution paths which lead to wait commands but where the condition does not hold. The operational semantics of the supported wait queue commands is shown in Figure 4.5. The possibilities for declaring and initializing wait queues are not mentioned because they are trivial. The static declaration of a wait queue sets the *init* field to false. If the wait queue is declared using the macro `DECLARE_WAIT_QUEUE_HEAD`, or the function `init_waitqueue_head` is used, then the *init* field is set to true.

$$\frac{s \models q.init = \top \wedge v = 1}{s[\![\texttt{wait\_event(q, v)}]\!]_{cmd} s} \quad \text{(wq-wait)}$$

$$\frac{s \models q.init = \top \wedge v = 1}{s[\![\texttt{i := wait\_event\_interruptible(q, v)}]\!]_{cmd} s\langle i, 0\rangle} \quad \text{(wq-wait-irp-1)}$$

$$\frac{s \models q.init = \top}{s[\![\texttt{i := wait\_event\_interruptible(q, v)}]\!]_{cmd} s\langle i, -1\rangle} \quad \text{(wq-wait-irp-2)}$$

$$\frac{s \models q.init = \top}{s[\![\texttt{wake\_up(q)}]\!]_{cmd} s} \quad \text{(wq-wakeup)}$$

$$\frac{s \models q.init = \top}{s[\![\texttt{wake\_up\_interruptible(q)}]\!]_{cmd} s} \quad \text{(wq-wakeup-irp)}$$

Figure 4.5.: The operational semantics for wait queues

4.4.3. Tasklets

Tasklets are modeled by a triple $(pc, cnt, sched)$. pc is the program counter of the tasklet handler function. If the handler function of the tasklet is not given, pc is zero. cnt counts the number of "open" disables. Hence, each disabling of the tasklet increases this number, while each enabling decreases it. A tasklet is only executed

if the counter is zero. Finally, *sched* denotes if the tasklet should be scheduled. The operations semantics for tasklets is shown in Figure 4.6. Here, we also omit the declaration and initialization functions. The static declaration sets the fields *pc* and *cnt* to zero, and *sched* to false. If the tasklet is declared using the macro `DECLARE_TASKLET`, then *pc* is set to the program counter of the handler function.

In Figure 4.7, the program *ExecTasckletFunction* is shown. If it is called, it searches for a registered tasklet which was scheduled before and which disabling counter is zero. If there exists such a tasklet, the execution context is set to the interrupt context and the tasklet's handler function is executed. Note that the rules allow a tasklet handler function to schedule itself.

$$\frac{s \models t.pc \neq 0}{s[\![\texttt{tasklet\_schedule(\&t)}]\!]_{cmd} s\langle t.sched, \top\rangle} \quad \text{(tsk-sched)}$$

$$\frac{}{s[\![\texttt{tasklet\_disable(\&t)}]\!]_{cmd} s\langle t.cnt, t.cnt + 1\rangle} \quad \text{(tsk-disable)}$$

$$\frac{s \models t.cnt > 0}{s[\![\texttt{tasklet\_enable(\&t)}]\!]_{cmd} s\langle t.cnt, t.cnt - 1\rangle} \quad \text{(tsk-enable)}$$

Figure 4.6.: The operational semantics for tasklets

```
1:  procedure ExecTaskletFunction
2:      if  ∃i : t_i.pc ≠ 0 ∧ t_i.cnt = 0 ∧ t_i.sched = ⊤  then
3:          t_i.sched = ⊥;
4:          context := irp;
5:          call  t_i.pc;
6:          context := proc;
```

Figure 4.7.: The program *ExecTasckletFunction*

4.4.4. Work Queues

Because we do not model work queues in general, but only the shared work queue, we may omit the queue formalization and just model the work elements themselves. A work is modeled, similar to tasklets but without the disabling counter, as a tuple of a program counter and a scheduling flag. The operational semantics for the shared work queue is shown in Figure 4.8. If a work structure is statically declared, its program counter is set to zero and its field *sched* to false. When using the macro `DECLARE_WORK`, the program counter is set to the entry point of the handler function.

Furthermore, we also define a program *ExecWorkQueueFunction*, similar to the tasklet executing program shown in Figure 4.7. The only differences between them

are that the work queue program does not need to care about a disabling counter and that it is run in the context of a process.

$$\frac{s \models w.sched = \bot}{s[\![\texttt{INIT\_WORK(\&w, f, d)}]\!]_{cmd} s\langle w.pc, PC(f)\rangle} \quad \text{(work-init)}$$

$$\frac{s \models w.sched = \bot}{s[\![\texttt{PREPARE\_WORK(\&w, f, d)}]\!]_{cmd} s\langle w.pc, PC(f)\rangle} \quad \text{(work-prepare)}$$

$$\frac{s \models w.pc \neq 0}{s[\![\texttt{schedule(\&w)}]\!]_{cmd} s\langle w.sched, \top\rangle} \quad \text{(work-sched)}$$

Figure 4.8.: The operational semantics for work queues

4.4.5. Interrupts

Interrupts are modeled as a function *irp*, which maps integers to interrupt values. Because the x86 architecture has a fixed number of 16 interrupts, *irp* is only defined for the values 0 to 15. An interrupt value is a tuple of a program counter and a boolean value which denotes whether the interrupt has been requested correctly before. The operational semantics for requesting and releasing interrupts is shown in Figure 4.9. Calling one of the requested interrupt handler functions is done by the program *ExecInterruptFunction*. It is shown in Figure 4.10.

$$\frac{s \models irq(n).req = \bot}{s[\![\texttt{request\_irq(n, f, ...)}]\!]_{cmd} s\langle irq(n).req, \top \mid irq(n).pc, PC(f)\rangle} \quad \text{(irq-req)}$$

$$\frac{s \models irq(n).req = \top}{s[\![\texttt{free\_irq(n, ...)}]\!]_{cmd} s\langle irq(n).req, \bot \mid irq(n).pc, 0\rangle} \quad \text{(irq-free)}$$

Figure 4.9.: The operational semantics for interrupts

```
1:  procedure ExecInterruptFunction
2:      if  ∃i : 0 ≤ i ≤ 15 ∧ irq(i).req = ⊤  then
3:          context := irp;
4:          call  irq(i).pc;
5:          context := proc;
```

Figure 4.10.: The program *ExecInterruptFunction*

4.4.6. I/O Ports and Memory Usage

The requested I/O ports are represented by the pair $ioport = (\mathbb{N} \times \mathbb{N})$ of two natural numbers. The first element denotes the first requested I/O port, and the second elements denotes the number of requested I/O ports. The functions `request_region` and `release_region`, for requesting and releasing I/O ports, are the only functions that change the value of *ioport*. In the operational model, all API functions which access I/O ports, have the short precondition that the port must be within the requested region.

The rules of the operational model for the memory usage functions do not change the state. Instead, there is the only precondition that the memory function is called in the correct context.

4.4.7. Sequential Execution Model

Finally, we present the program *SequentialDriverExec*, as shown in Figure 4.11. This program corresponds to the sequential execution model presented in Section 4.3.7. First, it properly initializes the interrupt values and calls the init function f_{init} of the device driver. The main loop executes either one of the driver's functions, a tasklet handler function, a work handler function or an interrupt handler function arbitrary times and in arbitrary order. Note that the function nondet_int() returns an arbitrary integer value. Eventually, the driver's cleanup function f_{exit} is called.

```
 1: procedure SequentialDriverExec
 2:     irq = λx.(0, ⊥);
 3:     call f_init;
 4:     do {
 5:         r := nondet_int();
 6:         switch (r) {
 7:             case 1: call f_i with f_i ≠ f_init and f_i ≠ f_exit;
 8:             case 2: call ExecTaskletFunction();
 9:             case 3: call ExecWorkQueueFunction();
10:             case 3: call ExecInterruptHandler();
11:         }
12:     } while(r);
13:     call f_exit;
```

Figure 4.11.: The program *SequentialDriverExec*

4.5. Proof of Over-Approximation

In the previous sections, we have introduced both a CCS based semantics of the Linux kernel API and an operational model of it. The operational model is defined in a way that it is an over-approximation of the CCS based semantics. Informally,

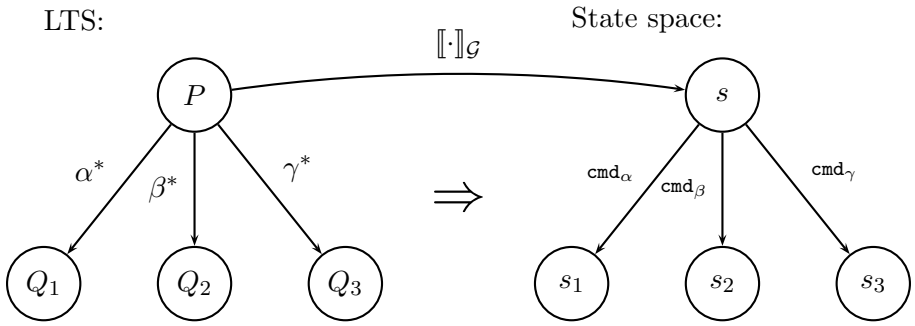

Figure 4.12.: General idea to prove the over-approximation

this means that the operational model defines more behaviour than the CCS based semantics. This makes the verification process more efficient. On the other hand, it can cause false negatives. Note that because of the over-approximation, we do not have false positives, i.e. the verifier never incorrectly reports that a program is correct.

In this section, we give the general idea of how it can be proven that the operational model is an over-approximation of the CCS based semantics. We do not want to give the proof in full detail, because this would be very technical. The general idea of the proof is shown in Figure 4.12. We assume that both models can be defined as graphs. This is trivial for the CCS based semantics because of the operational semantics of CCS. This defines a labeled transition system that can directly be translated to a cyclic graph. For the operational model, we can define a graph in the following way. Each state of the operational model corresponds to a node in its graph representation. Two graph states s and s' are connected if there is a rule *cmd* in the operational model, such that s fulfils the preconditions of *cmd* and the resulting state of executing *cmd* in s is s'. The edge between s and s' is then labeled with the name of rule *cmd*.

Let the graph of the CCS based semantics be denoted with $\mathcal{G}_1$ and the graph of the operational model be denoted with $\mathcal{G}_2$. To relate the graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, we have to define a function $[\![\cdot]\!]_{\mathcal{G}} : \mathcal{G}_1 \mapsto \mathcal{G}_2$, mapping states defined by CCS processes to states in the state space of the operational model. This mapping function is inductively defined over the structure of the set of CCS processes that are the result of translating a device driver, as defined in Section 4.3. We omit the whole definition of $[\![\cdot]\!]_{\mathcal{G}}$ and instead just explain the general idea with the example of semaphores. When a semaphore variable x is translated to CCS, the general result is the following process definition:

$$Sem^x \stackrel{\text{def}}{=} down^x.up^x.Sem^x$$

where the translation can be guarded by an $init^x$ transition if the variable is not initialized. Therefore, the partial definition of $[\![\cdot]\!]_{\mathcal{G}}$ for semaphores is defined in the

following way:

$$\begin{aligned}
[\![init^x.Sem^x]\!]_{\mathcal{G}} &= \{\sigma \mid \sigma \in \Sigma \wedge \sigma_{sem}(x) = (\bot, \bot)\} \\
[\![down^x.up^x.Sem^x]\!]_{\mathcal{G}} &= \{\sigma \mid \sigma \in \Sigma \wedge \sigma_{sem}(x) = (\top, \bot)\} \\
[\![up^x.Sem^x]\!]_{\mathcal{G}} &= \{\sigma \mid \sigma \in \Sigma \wedge \sigma_{sem}(x) = (\top, \top)\}
\end{aligned}$$

Let $[\![\cdot]\!]_{\mathcal{G}}$ be defined for all parts of the Linux kernel API. Furthermore, let $P = \{p_1, \ldots, p_n\}$, a set of CCS processes, be given, resulting from the translation of a device driver. For P, the function $[\![\cdot]\!]_{\mathcal{G}}$ is then defined by:

$$[\![P]\!]_{\mathcal{G}} = \bigcap_{1 \leq i \leq n} [\![p_i]\!]_{\mathcal{G}}$$

Using the definition of $[\![\cdot]\!]_{\mathcal{G}}$, we can relate states in the CCS based semantics to states of the operational model. Now we have to prove that the operational model captures at least the behaviour of the CCS based semantics. This is defined by the following formula:

$$Q \xrightarrow{\alpha^*} Q' \wedge [\![Q]\!]_{\mathcal{G}} = s \Rightarrow \exists s'.(s \xrightarrow{cmd_{\alpha^*}} s' \wedge [\![Q']\!]_{\mathcal{G}} = s')$$

where Q and Q' are CCS processes and Q' is derived from Q on a transition path $\alpha^* : Q \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} Q'$. If this is the case and Q corresponds to the state s in the operational model, then we have to prove that there exists a corresponding rule cmd_{α^*} such that it can be executed in state s and its resulting state s' corresponds to Q'. Therefore, each behaviour of the CCS based semantics is captured by the operational model, but not vice versa. Note that we have to reason about paths in the CCS based semantics, because in the general case, more than one CCS transition may correspond to a command of the Linux kernel API. Because all the subsystems of the Linux kernel API are translated to different CCS processes that do not influence each other, it is quite straightforward, but very technical, to give the proof in full detail, which is why we refrained from providing it in the first place.

5. Model Checking with SATABS

Model checking [18] is a formal verification technique for finding bugs in complex system descriptions. In particular, these systems include hardware, protocols and software. A model checker answers the question whether a given model $\mathcal{M}$ fulfills a property ϕ, i.e. the model checker checks whether $\mathcal{M} \models \phi$ holds. In model checking, the models are usually given as transition systems, for example as *Kripke structures*. The properties are encoded in temporal logics, like CTL, LTL or the very general μ-calculus [7, 18, 50]. If the model fulfills the property, the model checker answers 'yes' and 'no' otherwise. In the latter case, most model checkers also produce a counterexample given as a trace of system behaviour which causes the failure.

The very first approaches of model checking directly checked the property on the state space of the model. This causes the problem of *state explosion*. In verification of integrated circuits, for example, the number of states is quite often exponentially related to the size of the circuit [54]. The use of OBDDs (ordered binary decision diagrams) in model checking results in a significant breakthrough in verification, because they allow systems with a much larger state space to be verified. Model checking using OBDDs is called *symbolic model checking*.

Even when using symbolic model checking, state explosion is still a serious problem for the verification of large systems, especially of software. Software tends to be less structured than hardware, its state space may be infinite and it is usually asynchronous. Many successful techniques for dealing with the state explosion problem in software verification are based on the *partial order reduction*. These techniques exploit the independence of concurrently executed events. Two events are independent of each other when executing them in either order results in the same global state. More information about partial order reduction and a good overview of other approaches to the state explosion problem can be found in [18].

When research on software verification has been changed from dealing with toy examples to dealing with real world programs, it emerges that the mentioned techniques to the state explosion problem are still not sufficient. In the following section, we describe how SATABS handles this problem. Section 5.2 introduces some practical aspects of SATABS that are necessary to understand the implementation of DDVERIFY.

5.1. CEGAR-Framework

Another method for reducing the state space of models of software systems in the verification process is *abstraction*. Abstraction techniques reduce the program state

space by mapping the set of states of the actual system to a smaller abstract set of states in a way that preserves the relevant behaviours of the system. The drawback of abstractions is that they have to be done manually. That needs the knowledge of an expert and a huge amount of time. Therefore, it is not applicable for building full automatic model checking tools.

Predicate abstraction [22] is one of the most popular and widely applied methods for dealing with the state explosion problem in software model checking. It abstracts a program by only keeping track of certain predicates on the data. Each predicate is represented by a boolean variable in the abstract program, while the original data variables are eliminated. Verification of a software system with predicate abstraction consists of constructing and evaluating a finite-state system that is an abstraction of the original system with respect to a set of predicates.

In practice, a conservative over-approximation is used when abstracting the original program. This makes, in contrast to an exact abstraction, the verification process feasible because of the larger reduction of the state space. Instead, model checking the abstracted program may cause a false negative produced by a counterexample that cannot be simulated in the original program. This is usually called a *spurious counterexample*. Spurious counterexamples can be eliminated by changing the set of predicates that was used for the last abstraction process. This step is called *predicate refinement*.

The predicate abstraction process has been automated by the *counterexample guided abstraction refinement* framework, or *CEGAR* for short [14, 20, 22, 66]. This framework is shown in Figure 5.1. The steps in CEGAR based model checking are the following:

1. *Program abstraction:* The abstraction of the program is generated with respect to a given set of predicates.

2. *Verification:* The model checking algorithm is run in order to check if a property ϕ holds in the abstracted model. If the model checker succeeds, the CEGAR loop terminates. Otherwise, the resulting counterexample is used in the next step.

3. *Counterexample validation:* The counterexample is simulated on the original program in order to determine whether it is spurious. If it is not the case, the counterexample corresponds to an execution path of the original program such that ϕ is violated. The CEGAR loop terminates. Otherwise, it proceeds to the next step.

4. *Predicate refinement:* The set of predicates is changed in order to eliminate the detected spurious counterexample. The CEGAR loop proceeds to the first step with the updated set of predicates.

SatAbs generates boolean programs as the result of the program abstraction. Boolean programs [1, 2, 22] are programs with the usual control-flow constructs but

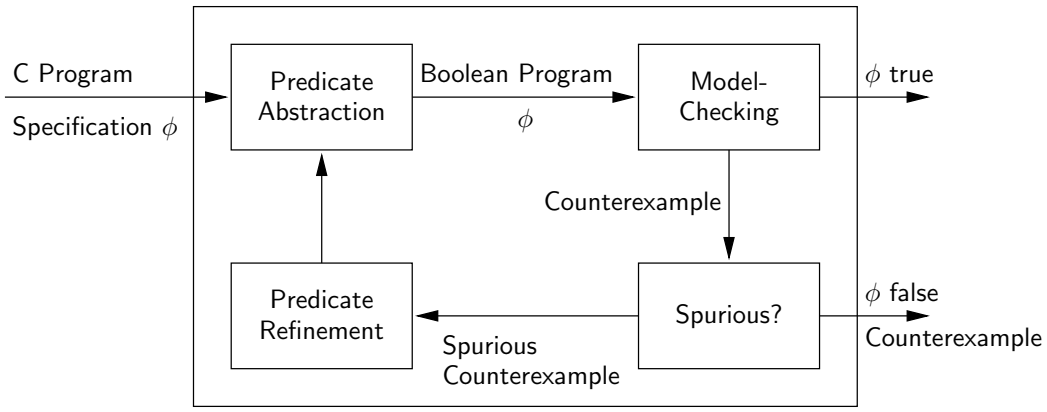

Figure 5.1.: The counterexample guided abstraction refinement framework

in which all variables are of type boolean. Boolean programs are useful for model checking because they have a finite state space, and reachability and termination are decidable (which is not the case for imperative programming languages in general). Either the model checker Cadence SMV [53] or Boppo [27] is then used to model check the boolean programs.

In SATABS, there is no possibility to define a property as a temporal logic formula. Instead of this, assertions can be introduced into the source code that are then checked to be true on all execution paths of the program. The use of assertions is explained in more detail in Section 5.2.

5.1.1. Example

To make the idea behind the CEGAR-framework clearer, we present an example, which is taken from [1]. Consider program P in Figure 5.2. In this program, `T` denotes the boolean value true and `F` denotes false. We want to check if the statement `assert(F)` at line 15 is reachable. We start with a boolean program that coarsely abstracts P and refines it incrementally, driven by the goal of answering the reachability query:

- Step 1: P is abstracted to a boolean program B_1 that only retains the controlflow structure of P. Every variable declaration of P has been removed, every assignment statement has been replaced by `skip` (denoted by "..." for readability), and every boolean expression has been replaced by "?". The latter one is the non-deterministic boolean choice.

- Step 2: It is checked, whether the assertion is reachable. The answer is positive because there exists a path in B_1 which leads to line 15, if the non-deterministic choices in lines 13 and 14 evaluate to true.

	P	B_1	B_2	B_3
1:	`numUnits: int;`		`nU0: bool;`	`nU0: bool;`
2:	`level: int;`			
3:	`void getUnit() {`	`void getUnit() {`	`void getUnit() {`	`void getUnit() {`
4:	`  canEnter: bool := F;`	`  ...;`	`  ...;`	`  cE: bool := F;`
5:	`  if (numUnits = 0) {`	`  if (?)  {`	`  if (nU0) {`	`  if (nU0) {`
6:	`  if (level > 10) {`	`  if (?)  {`	`  if (?)  {`	`  if (?)  {`
7:	`  NewUnit();`	`  ...;`	`  ...;`	`  ...;`
8:	`  numUnits := 1;`	`  ...;`	`  nU0 := F;`	`  nU0 := F;`
9:	`  canEnter := T;`	`  ...;`	`  ...;`	`  cE := T;`
10:	`  }`	`  }`	`  }`	`  }`
11:	`  } else`	`  } else`	`  } else`	`  } else`
12:	`  canEnter := T;`	`  ...;`	`  ...;`	`  cE := T;`
13:	`  if (canEnter)`	`  if (?)`	`  if (?)`	`  if (cE)`
14:	`  if (numUnits = 0)`	`  if (?)`	`  if (nU0)`	`  if (nU0)`
15:	`  assert(F);`	`  ...;`	`  ...;`	`  ...;`
16:	`  else`	`  else`	`  else`	`  else`
17:	`  gotUnit();`	`  ...;`	`  ...;`	`  ...;`
18:	`}`	`}`	`}`	`}`

Figure 5.2.: Example program P and three boolean programs (B_1, B_2 and B_3) that abstract P

- Step 3: This counterexample is spurious because it cannot be simulated in P. The variable $numUnits$ is constrained to be both equal to 0 (because of the evaluation of "?" in line 14) and not 0 (because of the lines 5 - 11 in P).
- Step 4: To refine the abstracted program B_1, the predicate $numUnits = 0$ is added.
- Step 1: P is abstracted to a boolean program B_2 with the only predicate $numUnits = 0$. A new boolean variable $nU0$ is added to keep track of the information if the variable `numUnits` is equal to 0 or not.
- Step 2: It is checked once more if line 15 is reachable in B_2. This is already the case, due to the following path: we assume `nU0` to be true at the beginning, "?" in line 6 evaluates to false and "?" in line 13 evaluates to true.
- Step 3: This counterexample is also spurious. In P, the variable `canEnter` is initially set to false and not updated up to line 13, where it is assumed to be true. This is a contradiction.
- Step 4: To eliminate this spurious counterexample, a boolean variable is added to model the `canEnter` condition, namely cE.
- Step 1: The boolean program B_3 is the result of abstracting the source program P with the predicates $numUnits = 0$ and $canEnter$.
- Step 2: Line 15 is not reachable in B_3, so it is not reachable in P. Note that B_3 contains no mention of the variable `level`, as its value does not impact the reachability of line 15.

5.2. Using SATABS

In this section, we describe how SATABS handles verification properties and how it deals with functions without bodies. Both are important in order to understand the implementation of DDVERIFY. A complete introduction for using SATABS can be found in [66].

SATABS does not allow to define verification properties in a temporal logic or other formal description languages. In contrast, assertions can be introduced in the code that should be verified. An assertion is an arbitrary predicate that is defined by a C boolean expression, e.g. `assert(i < MAX)` or `assert(cnt == 10 || err = -1)`. SATABS then checks that the assertions are fulfilled on all possible execution paths of the program. Furthermore, SATABS allows the verification of the following properties:

- Buffer overflows: For each array, SATABS checks whether the upper or lower bounds are violated whenever the array is accessed.
- Pointer safety: SATABS searches for null-pointer dereferences.
- Division by zero: SATABS checks whether there is a path in the program that executes a division by zero.

All the properties described above are *reachability* properties. They are of the form "Is there a path such that property ... is violated?". Liveness properties (something will eventually happen) are not supported by SATABS.

SATABS handles functions without bodies in the following way. It simply assumes that such a function returns an arbitrary value, but that no other locations than the one on the left hand side of the assignment are changed. We make use of this assumption in the following ways: First, if a kernel function does not change anything in our model, we do not need to implement a body for it. This is especially the case in most of the memory related functions, because we do not model the memory state. Second, we can easily define functions to get nondeterministic values of a special type. Assume that we need a function that returns an arbitrary integer value. We just have to define the following function in a global header file:

```
int nondet_int();
```

Because we do not define a body for this function, it will always return an arbitrary integer value during the verification process. Nondeterministic functions are defined for most data types in DDVERIFY.

6. Driver Verification with DDVerify

In this chapter, we present our tool DDVerify. The first section gives an introduction to the implementation of DDVerify. In the second section, we show how DDVerify can be used to verify Linux device drivers. And finally, we present the results of the case studies.

6.1. Implementation

DDVerify is a tool written in C. Its general structure is shown in Figure 6.1. A major part of DDVerify is the reimplementation of the Linux kernel source tree. Each source and header file of the kernel is replaced by its implementation of the operational model. When DDVerify is used on a device driver, it takes the driver's source code, the implementation of the operational model and further user input and collaborates with SatAbs to verify the device driver.

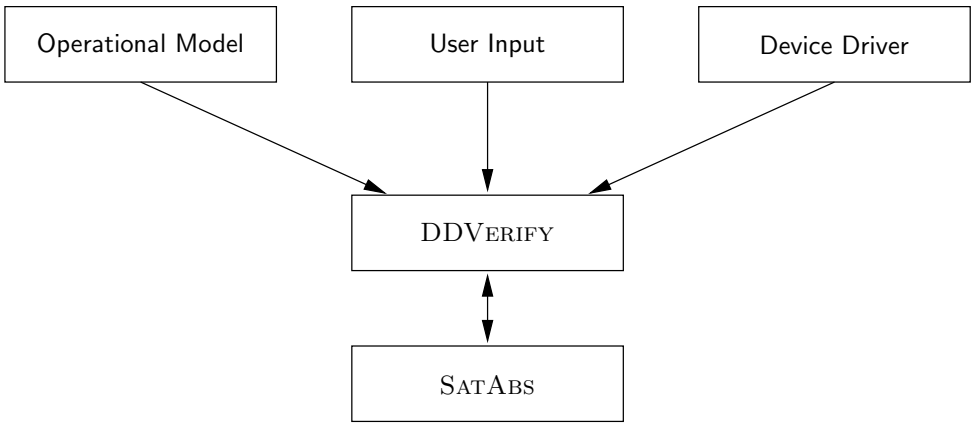

Figure 6.1.: Overview of DDVerify

At first, DDVerify extracts some information from the driver's source code. Several awk scripts [32] are used to parse the main source file of the device driver in order to find out the type of the driver (char, block or net), the used interfaces (e.g. PCI) and the driver's initialization and cleanup function. After this, DDVerify creates a new C source file named `__main.c`:

```
#include <ddverify/ddverify.h>
#include "dd.c"
```

```
int main()
{
   _ddv_module_init = dd_init;
   _ddv_module_exit = dd_exit;
   call_ddv();

   return 0;
}
```

Here, `dd.c` is the main source file of the device driver which should be verified with DDVerify. `dd_init` and `dd_exit` are the driver's initialization and its cleanup function. Both are stored in function pointers. The function `call_ddv` starts the execution model. In the standard configuration, DDVerify executes the device driver function with an sequential execution model. But it is also possible to verify a device driver with a concurrent execution model, which is still experimental.

A major work of the initialization function is to register the provided functions of the device driver to the kernel (see Section 2.3). The registration functions for character and block device drivers are also rewritten in the reimplementation of the kernel sources. In this way, DDVerify is able to extract the information about the available functions of a device driver and to use them when running the execution model.

By default, all verification checks are disabled. Therefore, the user has to provide the information which properties should be verified, e.g. spinlock or context related properties. In the implementation of the operational model, the assertions are guarded by a preprocessor macro in the following way:

```
#ifdef DDV_ASSERT_SPINLOCK
   __CPROVER_assert(lock->init, "Spinlock is initialized");
   __CPROVER_assert(!lock->locked, "Spinlock is not locked");
#endif
```

In this example, the precondition check for locking a spinlock is guarded with the preprocessor macro `DDV_ASSERT_SPINLOCK`. If the user wants to verify the spinlock usage, he has to provide the parameter `--check-spinlock` to DDVerify. Then, the corresponding preprocessor macro is enabled when running SatAbs.

6.2. Usage

To use DDVerify on a device driver, the user has to provide the source files of the driver and some further parameters. All possible parameters are shown in Figure 6.2. The most important ones are the `--check`-parameters, which define the properties that should be checked. The parameters `--module-init`, `--module-exit` and `--driver-type` are used if DDVerify is not able to find out either the initialization function, the cleanup function or the type of the driver automatically.

```
-D macro                          --check-pointer
--16, --32, --64                  --check-spinlock
--modelchecker #                  --check-semaphore
--iterations #                    --check-mutex
--ddv-path #                      --check-io
--module-init #                   --check-wait-queue
--module-exit #                   --check-tasklet
--driver-type #                   --check-work-queue
--model #                         --check-timer
--prepare-only                    --check-context
--check-bounds
```

Figure 6.2.: Parameters of DDVerify

Using the parameter `--model`, the execution model can be chosen. In the current version of DDVerify, one can choose between `seq` for the sequential model and `con` for the concurrent model.

After starting DDVerify on a device driver, SatAbs is called to prove all related claims. Each claim is proven by its own. For example, here is the output for proving that the device driver `char/efirtc.c` makes correct use of spinlocks:

```
$ ddverify efirtc.c --check-spinlock
Module initialization function: efi_rtc_init
Module cleanup function: efi_rtc_exit
Device driver type: char
PCI interface: no
Run satabs ...
Parse satabs output ...
12 Claims
Prove claim 1: + Time: 2.91 sec
Prove claim 2: + Time: 2.794 sec
Prove claim 3: + Time: 0 sec
Prove claim 4: + Time: 2.802 sec
Prove claim 5: + Time: 2.803 sec
Prove claim 6: + Time: 0 sec
Prove claim 7: + Time: 2.8 sec
Prove claim 8: + Time: 2.789 sec
Prove claim 9: + Time: 0 sec
Prove claim 10: + Time: 2.772 sec
Prove claim 11: + Time: 2.79 sec
Prove claim 12: + Time: 0 sec
Time to prove all positive claims: 22.459999
```

For each claim, a plus symbol indicates that the claim has been proven successfully.

	Number	Time	Failure	sec/claim	Prove Rate
Spinlock	513	1258.1	43	2.68	91.62%
Semaphore	4	28.89	2	14.45	50.00%
Mutex	7	0	7	0	0.00%
I/O	836	8164.41	69	10.64	91.75%
Wait Queue	44	2012.01	6	52.95	86.36%
Tasklet	1	0	1	0	0.00%
Work Queue	3	0	2	0	33.33%
Timer	41	5374.56	4	145.26	90.24 %
Context	193	0	0	0	100.00%
All	1642	16837.97	134	11.17	91.84%

Figure 6.3.: Statistics of the case studies

Furthermore, the time spent to prove the claim is reported. This number does not include the time for parsing and preparing the input files. The output of SatAbs for the nth claim is stored in the file `.claim_out_nth`.

6.3. Case Studies

We have used DDVerify to check 31 device drivers from Linux kernel 2.6.19.[1] SatAbs was used to verify a total of 27,926 lines of code. The smallest device driver verified had 148 LOC, the largest one 4,587 LOC, and the average was 901 LOC per device driver. All case studies were made on a Pentium-IV computer with 2.54 GHz and 1 GB RAM. As for the operating system, Debian 3.1 Linux was used.

Figure 6.3 shows the statistics of the case studies. Most of the claims are related to either spinlocks or I/O communication. The context check verifies that certain functions, like semaphores or everything that may access user space, are not allowed to be executed in interrupt context. Altogether, 1,642 claims were checked. 91.84% of them were proven successfully. SatAbs had problems with two device drivers, which by chance contained all of the mutex and tasklet claims. Thus, all corresponding checks failed. Consequently, all of these proofs failed. All context related checks could be proven by SatAbs's reachability analysis which is executed before the model checker is run. Hence, the time to prove the claims is zero.

The time to prove a claim highly depends upon the number of functions that are involved in its verification. Most of the spinlock claims are related to correct locking and unlocking behaviour. Because we make use of the sequential execution model, we have no interleaving between functions. In most cases, a function locks a spinlock at the beginning and releases it at the end. There are no inter-procedural checks required. In contrast to this, most of the timer checks involve at least two

[1] taken from `http://www.kernel.org`

different functions. A timer check verifies that a timer has always been initialized before it is used the first time. The initialization of a timer is typically done in the initialization function of the device driver, whereas the timer itself is used later in other functions of the driver. Therefore, SATABS has to prove that on all possible execution paths, whenever the initialization function of the driver succeeds, every used timer is initialized correctly. This makes these proofs much more time intensive than spinlock checks.

134 claims failed the verification. Two of these failures are bugs in the device drivers. 57 proofs did not terminate within 20 iterations of the CEGAR-loop. SATABS ends up with an internal error during 58 proofs. For the remaining 14 proofs, SATABS reports a counterexample that is a false negative. Two of these counterexamples are related to a bug in SATABS. The remaining 12 counterexamples are caused by the over-approximation of the operational model. The I/O model saves just one requested I/O port region, but it is possible to request arbitrary many I/O port regions in general. If a driver requests two I/O port regions, the model overwrites the first request. Hence, it reports an error if the driver reads or writes to a port of the I/O region which has been requested first. We have also defined and implemented an alternative I/O model that is more correct, but using it, the time to prove the I/O related claims rose by a factor ranging from 20 to 30. Therefore, we have omitted it from this presentation. More effort has to be invested in order to define an I/O model that is both correct with respect to multiple I/O port region requests and fast enough in practice.

```
1:  int ds1286_open(struct inode *inode, struct file *file)
2:  {
3:     spin_lock_irq(&ds1286_lock);
4:     if (ds1286_status & RTC_IS_OPEN)
5:       goto out_busy;
6:     ds1286_status |= RTC_IS_OPEN;
7:     spin_unlock_irq(&ds1286_lock);
8:     return 0;
9:  out_busy:
10:    spin_lock_irq(&ds1286_lock);
11:    return -EBUSY;
12: }
```

Figure 6.4.: Spinlock bug in device driver `char/ds1286.c`

DDVERIFY has found two bugs in two of the 31 device drivers. The first bug appears in the device opening function of the device driver `char/ds1286.c`. Its source code is shown in Figure 6.4. The device driver uses the variable `ds1286_status` to ensure that only one device is opened at a time. To avoid racing conditions on this variable, the check is guarded with the spinlock `ds1286_lock`. In line 3, this spinlock is locked. If the device is already open, the function jumps to the label

`out_busy` in line 9. But instead of unlocking the spinlock, it is locked again. This violates the rule that a locked spinlock is not allowed to be locked again by the same process. The second bug appears in the initialization function of the device driver `char/watchdog/machzwd.c`. The relevant part of its source code is shown in Figure 6.5. In lines 4 and 5, the device driver accesses the ports 0x218 and 0x21A, respectively. But these ports has not been requested at this point in the execution of the code. Instead, the request function is called in line 7.

```
1: static int zf_init(void)
2: {
3:    int ret;
4:    outb(0x02, 0x218);
5:    ret = inw(0x21A);
6:    ...
7:    request_region(0x218, 3, ``...'');
8:    ...
9: }
```

Figure 6.5.: I/O port bug in device driver `char/watchdog/machzwd.c`

7. Discussion

In this thesis, we have shown that formal methods and formal verification techniques can indeed be applied to real world software. We have chosen Linux device drivers because they are responsible for most errors in operating systems and because it is hard to find and remove them with conventional methods like testing. In Section 4, we gave a semantics for the Linux kernel API. To our best knowledge, this is the first approach to formally define the Linux kernel API. We have only defined a semantics for a small subset of the API, but it shows a good way of defining such semantics, which can easily be applied to the rest of the API, too. Although we do not belief that it is necessary or meaningful for defining the whole API in CCS. This process algebra is very suitable to define the behaviour of concurrent processes, but it is not easy and not very legible to define highly data dependent behaviour in it.

In Section 6, DDVerify was introduced. This tool is based on the previous formalization. It consists of several independent elements: the implementation of the operational model, two execution models and a central controlling part, which is responsible of managing all these elements altogether, in order to run the verification process with SatAbs. In the cases studies, which are presented in Section 6.3, we have shown that DDVerify is able to verify real world device drivers. We have verified 31 device drivers from the Linux kernel 2.6.19. DDVerify is now able to verify char and lock device drivers with a code size of up to 3,000 lines of code. We believe that the supported size of the drivers can be improved easily by extending the set of supported API functions.

7.1. Future Work

Both parts of this thesis, the theoretical and the practical one, can be extended in multiple ways. The formalization can be extended to support more parts of the Linux kernel API. It would be interesting to examine if other formal description methods are more suitable for this purpose. The use of CCS has the disadvantage in that it is hard to define highly data dependent behaviour.

We have also started some work on checking the CCS definitions with "The Concurrency Workbench of the New Century" [25, 26]. Using this program, systems can be defined with CCS, also potentially extended with priorities, and checked for properties defined in the μ-calculus. This work is not finished yet, but it is worthwhile to continue it. In this way, the definitions can be verified to be consistent with themselves.

SatAbs is able to verify safety properties only. It is shown in [39], that it is possible to extend the specifications for model checking based on the CEGAR

framework to the full power of CTL. If we could extend SATABS in the same way, we would be able to verify liveness properties, e.g. termination, too. Other methods are known to either approximate liveness properties by safety properties [71] or to translate liveness properties into equivalent safety properties [8].

The concurrent execution model of DDVERIFY is still experimental due to the verification time of concurrent software in SATABS. More research in this direction should be done in order to be able to deal with a potentially restricted number of threads in a manageable time. In [70], an approach is presented to combine sequential assignments into parallel assignment blocks. When this technique is used in model checking based on the CEGAR-framework, the assignment combination procedure can lead to significant speed-ups. This was also shown in several case studies, among other things for a set of windows device drivers. In [64], a method is presented to translate a concurrent program into a sequential program that simulates the execution of a large subset of the behaviours of the concurrent program. This technique has been implemented in SLAM [6] and has been used to verify concurrent windows device drivers. We believe that this technique can be directly adapted to work with SATABS, because its technical background is quite similar to SLAM's.

In the current version, DDVERIFY runs SATABS to verify each claim on its own. We have implemented it this way to make debugging SATABS and DDVERIFY easier. But this has the disadvantage that SATABS has to parse all the files for each claim. It takes more time than the run of the model checker, especially in smaller device drivers. For a final version, changing SATABS to verify all claims in one stage would improve the overall verification time by a factor quite noteworthy. In order to deal with a huge number of claims, SATABS could be changed to deal not only with either one claim or all claims, but with a given set of claims. DDVERIFY would then be used to split the set of all claims into smaller partitions and to forward them to SATABS.

7.2. Acknowledgments

There are a lot of people who have made my studies and this thesis not only possible, but an interesting and instructive time, too. First of all, I would like to thank my parents, Therese and Adrian, who have supported me in every possible way. Without my wife Juliane, the time of my study would not have been such a remarkable time. She always had the patience to listen to me about this thesis, what helped me very much to structure my thoughts.

I would like to thank Stephan Hölldobler for his engagement in the study program "Computational Logic" at the Technical University of Dresden. It was a great chance for me to go deep into the theory of computer science and to work together with highly motivated people.

Daniel Kroening gave me the unique chance to take a semester abroad in Switzerland and to write this thesis at his research group at the ETH Zürich. It was a great

possibility to apply all the theory in a practical project. Nicolas Blanc had part in that is was a wonderful time in Zürich for me. During my work, he reviewed my manuscript, gave me a lot of helpful comments, and helped me to find the right way again when I was lost. Without Hanni Sommer, it would have been much harder to organize my studies in Zürich.

Finally, I would like to thank Christoph Schmidt for his help explaining the English language to me and finding a lot of mistakes I made in this thesis.

A. Supported Linux Kernel API Functions

Function/Macro	Description
Semaphores	
`DECLARE_MUTEX(name)`	Declaration of a semaphore
`void init_MUTEX(struct semaphore *sem)`	Initialization of a semaphore
`void down(struct semaphore *sem)`	Requests the semaphore, waits as long as needed
`int down_interruptible(struct semaphore *sem)`	Requests the semaphore, waiting is interruptible
`int down_trylock(struct semaphore *sem)`	Requests the semaphore without waiting
`void up(struct semaphore *sem)`	Releases the semaphore
Spinlocks	
`spinlock_t lock = SPIN_LOCK_UNLOCKED`	Initialization of a spinlock at compile time
`void spin_lock_init(spinlock_t *lock)`	Initialization of a spinlock at runtime
`void spin_lock(spinlock_t *lock)`	Requests the spinlock
`void spin_unlock(spinlock_t *lock)`	Releases the spinlock
Memory usage	
`void *kmalloc(size_t size, int flags)`	Allocates memory
`void kfree(void *)`	Frees memory
Interrupt requests	
`int request_irq(unsigned int irq, irqreturn_t (*handler)(int, void*, struct pt_regs*), unsigned long flags, const char *dev_name, void *dev_id)`	Interrupt registration interface
`void free_irq(unsigned int irq, void *dev_id)`	Releases an interrupt

Function/Macro	Description
Wait queues	
`DECLARE_WAIT_QUEUE_HEAD(queue)`	Initialization of a queue at compile time
`void init_waitqueue_head(wait_queue_head_t *queue)`	Initialization of a queue at runtime
`void wait_event(queue, condition)`	Uninterruptible sleep
`int wait_event_interruptible(queue, condition)`	Sleeps as long as condition does not hold (the sleep is interruptible)
`void wake_up(wait_queue_head_t *queue)`	Wakes up all processes waiting on the given queue
`void wake_up_interruptible(wait_queue_head_t *queue)`	Wakes up all processes, with an interruptible sleep, waiting on the given queue
Tasklets	
`DECLARE_TASKLET(name, function, date)`	Initialization of a tasklet at compile time
`void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data)`	Initialization of a tasklet at runtime
`void tasklet_schedule(struct tasklet_struct *t)`	Schedule the tasklet for execution
`void tasklet_disable(struct tasklet_struct *t)`	Disable the tasklet
`void tasklet_enable(struct tasklet_struct *t)`	Enable the tasklet
Work queues	
`DECLARE_WORK(name, void (*function)(void *), void *data`	Declaration of a work queue entry
`INIT_WORK(struct work_struct *work, void (*function)(void *), void *data)`	Initialization of a work queue entry
`PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data)`	Initialization of a work queue entry
`schedule_work(struct work_struct *work)`	Schedules a work on the shared work queue

Function/Macro	Description
I/O ports	
`struct resource *request_region(unsigned long start, unsigned long len, char *name)`	Resource allocator for I/O ports
`void release_region(unsigned long start, unsigned long len)`	Deallocates the I/O port
`unsigned in[b|w|l](unsigned port)`	Reads a byte/word/longword from a port
`void out[b|w|l](unsigned short [byte|word|longword], unsigned port)`	Writes a byte/word/longword to a port
`void ins[b|w|l](unsigned port, void *addr, unsigned long count)`	Reads data from a port to memory
`void outs[b|w|l](unsigned port, void *addr, unsigned long count)`	Writes data from memory to a port

Function/Macro	Description
I/O memory	
`struct resource *request_mem_region(unsigned long start, unsigned long len, char *name)`	Allocates an I/O memory region
`void release_mem_region(unsigned long start, unsigned long len)`	Releases an I/O memory region
`void *ioremap(unsigned long phys_addr, unsigned long size)`	Makes an allocated I/O memory region accessible
`unsigned int ioread[8|16|32](void *addr)`	Reads one value from I/O memory
`void iowrite[8|16|32](u[8|16|32] value, void *addr)`	Writes one value to I/O memory
`void ioread[8|16|32]_rep(void *addr, void *buf, unsigned long count)`	Reads a series of values from I/O memory
`void iowrite[8|16|32]_rep(void *addr, const void *buf, unsigned long count)`	Writes a series of values to I/O memory
`void memset_io(void *addr, u8 value, unsigned int count)`	Fills an area of I/O memory with the given value
`void memcpy_fromio(void *dest, void *source, unsigned int count)`	Copies data from I/O memory
`void memcpy_toio(void *dest, void *source, unsigned int count)`	Copies data to I/O memory
`void *ioport_map(unsigned long port, unsigned int count)`	Maps I/O ports and makes them appear to be I/O memory
`void ioport_unmap(void *addr)`	Cancels the mapping of the I/O ports.

B. Result of the Translation

$$
\begin{aligned}
W_{\texttt{devopen}} \stackrel{\text{def}}{=}\ & call^{\texttt{devopen}}.(\\
& (read^{\texttt{result2}}(x).(\text{if } x = 0 \text{ then } \overline{res}(1) \text{ else } \overline{res}(0)))\ Into(x) \\
& (\text{if } x = 1 \text{ then} \\
& \quad \overline{down}^{\texttt{lock}}.Done\ Before \\
& \quad Done\ Before \\
& \quad (\overline{res}(1)\ Into(x)\ (\overline{write}^{\texttt{open}}(x).Done))\ Before \\
& \quad (\overline{up}^{\texttt{lock}}.Done) \\
& \text{else } Done) \\
&)\ Before\ \overline{return}^{\texttt{devopen}}.W_{\texttt{devopen}} \\
W_{\texttt{devclose}} \stackrel{\text{def}}{=}\ & call^{\texttt{devclose}}.(\\
& (read^{\texttt{open}}(x).(\text{if } x = 1 \text{ then } \overline{res}(1) \text{ else } \overline{res}(0)))\ Into(x) \\
& (\text{if } x = 1 \text{ then} \\
& \quad \overline{down}^{\texttt{lock}}.Done\ Before \\
& \quad Done\ Before \\
& \quad (\overline{res}(0)\ Into(x)\ (\overline{write}^{\texttt{open}}(x).Done))\ Before \\
& \quad \overline{up}^{\texttt{lock}}.Done \\
& \text{else } Done) \\
&)\ Before\ \overline{return}^{\texttt{devclose}}.W_{\texttt{devclose}} \\
W_{\texttt{devread}} \stackrel{\text{def}}{=}\ & call^{\texttt{devread}}.(\\
& (read^{\texttt{open}}(x).(\text{if } x = 1 \text{ then } \overline{res}(1) \text{ else } \overline{res}(0)))\ Into(x) \\
& (\text{if } x = 1 \text{ then} \\
& \quad W'\ Before \\
& \quad Done\ Before \\
& \quad \overline{res}(0)\ Into(x)\ (\overline{write}^{\texttt{readdata}}(x).Done)
\end{aligned}
$$

$$
\begin{aligned}
&\text{else } \mathit{Done})\\
&) \mathit{Before}\, \overline{return}^{\texttt{devread}}.W_{\texttt{devread}}\\
W' &\stackrel{\text{def}}{=} read^{\texttt{readdata}}(x).(\text{if } x = 1 \text{ then } \mathit{Done} \text{ else } wakeupQueue_0^{\texttt{queue}}.W')\\
W_{\texttt{devwrite}} &\stackrel{\text{def}}{=} call^{\texttt{devwrite}}.(\\
&(read^{\texttt{open}}(x).(\text{if } x = 1 \text{ then } \overline{res}(1) \text{ else } \overline{res}(0)))\ \mathit{Into}(x)\\
&(\text{if } x = 1 \text{ then}\\
&\quad \mathit{Done}\\
&\text{else } \mathit{Done})\\
&) \mathit{Before}\, \overline{return}^{\texttt{devwrite}}.W_{\texttt{devwrite}}\\
W_{\texttt{irqhandler}} &\stackrel{\text{def}}{=} call^{\texttt{irqhandler}}.(\\
&\mathit{Done}\ \mathit{Before}\ \overline{startWakeup}^{\texttt{queue}}.\mathit{Done}\\
&) \mathit{Before}\, \overline{return}^{\texttt{irqhandler}}.W_{\texttt{irqhandler}}\\
W_{\texttt{init}} &\stackrel{\text{def}}{=} call^{\texttt{init}}.(\\
&(\overline{res}(0) + \overline{res}(-1))\ \mathit{Into}(x)\ (\overline{write}^{\texttt{result1}}(x).\mathit{Done})\ \mathit{Before}\\
&(read^{\texttt{result1}}(x).(\text{if } x = 0 \text{ then } \overline{res}(1) \text{ else } \overline{res}(0)))\ \mathit{Into}(x)\\
&(\text{if } x = 1 \text{ then}\\
&\quad ((\overline{requestIrqHandler}^{7}(\texttt{irqhandler}).\overline{res}(0) + \overline{res}(-1))\\
&\qquad \mathit{Into}(x)\ (\overline{write}^{\texttt{result2}}(x).\mathit{Done}))\ \mathit{Before}\\
&\quad (read^{\texttt{result2}}(x).(\text{if } x = -1 \text{ then } \overline{res}(1) \text{ else } \overline{res}(0)))\ \mathit{Into}(x)\\
&(\text{if } x = 1 \text{ then}\\
&\qquad \overline{call}^{\texttt{cleanup}}.return^{\texttt{cleanup}}.\mathit{Done}\\
&\quad \text{else } \mathit{Done})\\
&\text{else } \mathit{Done})\\
&) \mathit{Before}\, \overline{return}^{\texttt{init}}.W_{init}\\
W_{\texttt{cleanup}} &\stackrel{\text{def}}{=} call^{\texttt{cleanup}}.(\\
&(read^{\texttt{result1}}(x).(\text{if } x = 0 \text{ then } \overline{res}(1) \text{ else } \overline{res}(0)))\ \mathit{Into}(x)\\
&(\text{if } x = 1 \text{ then}\\
&\quad \mathit{Done}\ \mathit{Before}
\end{aligned}
$$

$$
\begin{array}{rl}
 & (read^{\texttt{result2}}(x).(\text{if } x = 0 \text{ then } \overline{res}(1) \text{ else } \overline{res}(0)))\ Into(x) \\
 & (\text{if } x = 1 \text{ then} \\
 & \quad \overline{releaseIrqHandler}^{7}.Done \\
 & \quad \text{else } Done) \\
 & \text{else } Done) \\
 &)\ Before\ \overline{return}^{\texttt{cleanup}}.W_{\texttt{cleanup}} \\
Init & \stackrel{\text{def}}{=} \overline{call}^{\texttt{init}}.return^{\texttt{init}}.\overline{startExec}.nil \\
Exec & \stackrel{\text{def}}{=} startExec.(\overline{callFunc}.Exec + \overline{exit}.nil) \\
Func & \stackrel{\text{def}}{=} callFunc.(\\
 & \quad \overline{call}^{\texttt{devopen}}.return^{\texttt{devopen}}.\overline{startExec}.Func+ \\
 & \quad \overline{call}^{\texttt{devclose}}.return^{\texttt{devclose}}.\overline{startExec}.Func+ \\
 & \quad \overline{call}^{\texttt{devread}}.return^{\texttt{devread}}.\overline{startExec}.Func+ \\
 & \quad \overline{call}^{\texttt{devwrite}}.return^{\texttt{devwrite}}.\overline{startExec}.Func) \\
Exit & \stackrel{\text{def}}{=} exit.\overline{call}^{\texttt{cleanup}}.return^{\texttt{cleanup}}.nil \\
Model & \stackrel{\text{def}}{=} (Init \mid Exec \mid Func \mid Exit) \setminus \{startExec, callFunc, exit\}
\end{array}
$$

Bibliography

[1] T. Ball and S. K. Rajamani. *Boolean Programs: A Model and Process For Software Analysis*. Technical Report MSR-TR-2000-14, Microsoft Research, 2000.

[2] T. Ball and S. K. Rajamani. *Bebop: A Symbolic Model Checker for Boolean Programs*. SPIN Workshop on Model Checking of Software, vol. 1885 of LNCS, Springer Berlin Heidelberg, pp. 113-130, 2000.

[3] T. Ball and S. K. Rajamani. *SLIC: A specification language for interface checking*. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[4] T. Ball and S. K. Rajamani. *The SLAM Toolkit*. Proceedings of CAV, vol. 2102 of LNCS, Springer Berlin Heidelberg, pp. 260-264, 2001.

[5] T. Ball and S. K. Rajamani. *The SLAM Project: Debugging System Software via Static Analysis*. Proceedings of POPL, pp. 1-3, 2002.

[6] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani and A. Ustuner. *Thorough Static Analysis of Device Drivers*. Proceedings of EuroSys'06: European System Conference, 2006.

[7] J.A. Bergstra, A. Ponse and S.A. Smolka. *Handbook of Process Algebra*. Elsevier Science B.V, 2001.

[8] A. Biere, C. Artho and V. Schuppan. *Liveness Checking as Safety Checking*. vol. 66 of Electronic Notes in Theoretical Computer Science, No. 2, 2002.

[9] D. Bovet and M. Cesati. *Understanding the Linux Kernel - Third Edition*. O'Reilly, 2005.

[10] N. Blanc, A. Groce and D. Kroening. *Verifying C++ with STL Containers via Predicate Abstraction*. To appear.

[11] P. T. Breuer and S. Pickin. *Symbolic approximation: an approach to verification in the large*. Innovations in Systems and Software Engineering, vol 2, pp. 147-163, Springer London, 2006.

[12] W. R. Bush, J. D. Pincus and D. J. Sielaff. *A Static Analyzer for Finding Dynamic Programming Errors*. Software - Practice and Experience, vol. 7, pp. 775-802, 2002.

[13] T. Cattel. *Modeling and Verification of sC++ Applications*. TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems. vol 1384 of LNCS, pp. 232-248, 1998.

[14] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman and K. Yorav. *Efficient Verification of Sequential and Concurrent C Programs*. Formal Methods in System Design, Springer Netherlands, vol. 25 (2-3), pp. 129-166, 2004.

[15] S. Chandra, P. Godefroid and C. Palm. *Software Model Checking in Practice: An Industrial Case Study*. Proceedings of International Conference on Software Engineering, pp. 431-441, 2002.

[16] J. Chen. *On Verifying Distributed Multithreaded Java Programs*. Proceedings of the 33rd Hawaii International Conference on System Sciences, 2000.

[17] A. Chou, J. Yang, B. Chelf, S. Hallem and D. Engler. *An Empirical Study of Operating Systems Errors*. In SOSP 2001, ACM Press, pp. 73-88, 2001.

[18] E. Clarke, O. Grumberg and D. Preled. *Model Checking*. The MIT Press, 1999.

[19] E. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. *Counterexample-guided abstraction refinement*. In CAV'00: Computer Aided Verification, vol. 1855 of LNCS, pp. 154-169, 2000.

[20] E. Clarke, O. Grumberg, S. Jha, Y. Lu and Helmut Veith. *Counterexample-guided Abstraction Refinement*. Computer Aided Verification, pp. 154-169, 2000.

[21] E. Clarke and D. Kroening. *ANSI-C Bounded Model Checking - User Manual*. Available from the CBMC homepage: `http://www.cs.cmu.edu/~modelcheck/cbmc/`, 2003.

[22] E. Clarke, D. Kroening. N. Sharygina and K. Yorav. *Predicate Abstraction of ANSI-C Programs Using SAT*. Technical Report CMU-CS-03-186, Carnegie Mellon University, 2003.

[23] R. Cleaveland and M. Hennessy. *Priorities in Process Algebras*. Proceedings of the Third Annual Symposium on Logic in Computer Science, pp. 193-202, 1988.

[24] R. Cleaveland, G. Lüttgen, V. Natarajan and S. Sims. *Modeling and Verifying Distributed Systems Using Priorities: A Case Study*. In Proceeding of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96). LNCS 1055, pp. 287-297, 1996.

[25] R. Cleaveland, J. Parrow and B. Steffen. *The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems*. ACM Transactions on Programming Languages and Systems, pp. 36-72, 1993.

[26] R. Cleaveland, T. Li and S. Sims. *The Concurrency Workbench of the New Century - Version 1.2 - User's Manual*, 2000.

[27] B. Cook, D. Kroening and N. Sharygina. *Symbolic Model Checking for Asynchronous Boolean Programs*. SPIN, Springer, pp. 75-90, 2005.

[28] J. Corbet, A. Rubini and G. Kroah-Hartman. *Linux Device Drivers - Third Edition*. O'Reilly, 2005.

[29] J. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby and H. Zheng. *Bandera: Extracting Finite-state Models from Java Source Code*. In Proceedings of the 22nd International Conference on Software Engineering (ICSE), pp. 439-448, 2000.

[30] M. Das, S. Lerner and M. Seigle. *ESP: Path-Sensitive Program Verification in Polynomial Time*. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, vol. 37, pp. 57–68, 2002.

[31] C. Demartini, R. Iosif and R. Sisto. *Modeling and Validation of Java Multithreading Applications using SPIN*. In Proceedings of the 4th SPIN workshop, 1998.

[32] D. Dougherty and A. Robbins. *sed & awk - second edition*. O'Reilly & Associates, Inc., 1997.

[33] D. Engler and M. Musuvathi. *Static analysis versus software model checking for bug finding*. vol. 3653 of LNCS, 2005.

[34] D. Engler, D. Yu Chen, S. Hallem, A. Chou and B. Chelf. *Bugs as Deviant Behaviour: A General Approach to Inferring Errors in System Code*. In Proceedings of 18th ACM Symposium on Operating Systems Principles, pp. 57-72, 2001.

[35] K. Erk and L. Priese. *Theoretische Informatik - Eine umfassende Einführung*, Springer Verlag Berlin Heidelberg, 2. Auflage, 2002.

[36] C. Flanagan. *Verifying Commit-Atomicity using Model-Checking*. vol. 2989 of LNCS, pp. 252-266, 2004.

[37] P. Glück and G. Holzmann. *Using SPIN Model Checking for Flight Software Verification*. IEEEAC paper #435, 2002.

[38] P. Godefroid. *Software Model Checking: The VeriSoft Approach*. Technical report, Bell Labs Technical Memorandum ITD-03-44189G, 2003.

[39] A. Gurfinkel and M. Chechik. *Why Waste a Perfectly Good Abstraction?*, In Proceedings of TACAS 2006: Tools and Algorithms for the Construction and Analysis of Systems, vol 3920 of LNCS, pp 212-226, 2006.

[40] J. Hatcliff and M. Dwyer. *Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software*. vol. 2154 of LNCS, 2001.

[41] K. Havelund and J. U. Skakkebaek. *Applying Model Checking in Java Verification*. In Proceedings of 6th SPIN Workshop, Toulouse, 1999.

[42] T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre and W. Weimer. *Temporal-Safety Proofs for Systems Code*. In CAV 2002, vol. 2404 of LNCS, pp. 526-538, 2002.

[43] T. Henzinger, R. Jhala, R. Majumdar and M. A.A. Sanvido. *Extreme Model Checking*. In International Symposium on Verification: Theory and Practice. vol. 2772 of LNCS, pp. 332-358, 2003.

[44] T. Henzinger, R. Jhala and R. Majumdar. *Lazy Abstraction*. Symposium Principles of Programming Languages, ACM Press, pp. 58-70, 2002.

[45] M. Hohmuth and H. Tews. *The VFiasco approach for a verified operating system*. 2nd ECOOP Workshop on Programming Languages and Operating Systems (ECOOP - PLOS'05), 2005.

[46] G. J. Holzmann. *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley Longman, 2004.

[47] G. J. Holzmann. *Static Source Code Checking For User-Defined Properties*. Bell Labs Technical Report, 2002.

[48] G. J. Holzmann. *Logic Verification of ANSI-C code with SPIN*. In Proceedings of the 7th International SPIN Workshop. vol. 1885 of LNCS, pp. 131-147, 2000.

[49] G. J. Holzmann. *An Automated Verification Method for Distributed Systems Software Based on Model Extraction*. vol. 28 of IEEE Transactions on Software Engineering, pp. 364-377, 2002.

[50] M. Huth and M. Ryan. *Logic in Computer Science - Modelling and Reasoning about Systems, Second Edition*. Cambridge University Press, 2004.

[51] R. Love. *Linux Kernel Development - Second Edition*. Pearson Education, 2005.

[52] G. Lüttgen. *Pre-emptive Modelling of Concurrent and Distributed Systems*. Shaker Verlag, 1998.

[53] K. L. McMillan. *A Methodology for Hardware Verification using Compositional Model Checking*. Science of Computer Programming, vol 37, number 1-3, pp. 279-309, 2000.

[54] C. Meinel and T. Theobald. *Algorithmen und Datenstrukturen im VLSI-Design: OBDD - Grundlagen und Anwendungen*. Springer-Verlag Berlin Heidelberg, 1998.

[55] R. Milner. *A Calculus of Communicating Systems*. vol. 92 of LNCS, 1980.

[56] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[57] I. Molnar. *Generic Mutex Subsystem*. Linux kernel 2.6.19.1 documentation: `Documentation/mutex-design.txt`.

[58] J. Mühlberg and G. Lüttgen. *BLASTING Linux Code*. To appear in the proceedings of 11th International Workshop on Formal Methods for Industrial Critical Systems, 2006.

[59] M. Musuvathi and D. Engler. *Model Checking Large Network Protocol Implementations*. In Proceeding of The First Symposium on Networked Systems Design and Implementation, USENIX Association, pp 155-168, 2004.

[60] M. Musuvathi, David Y.W. Park, A. Chou, D. R. Engler and D. L. Dill. *CMC: A Pragmatic Approach to Model Checking Real Code*. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation, 2002.

[61] D. Y.W. Park, U. Stern and D. L. Dill. *Java Model Checking*. In Proceedings of 5th IEEE International Conference on Automated Software Engineering, pp. 253-256, 2000.

[62] M. Papathomas. *A Unifying Framework for Process Calculus Semantics of Concurrent Object-Based Languages and Features*. Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing, vol. 612 of LNCS, pp. 53-79, 1992.

[63] H. Post and W. Küchlin. *Automatic Data Environment Construction for Static Device Drivers Analysis*. In Proceedings of the 2006 conference on Specification and verification of component-based systems, pp. 89-92, 2006.

[64] S. Qadeer and D. Wu. *KISS: Keep It Simple and Sequential*. Accepted at the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2004.

[65] J. Regehr. *Thread Verification vs. Interrupt Verification*. In Proceedings of TV'06: Multithreading in Hardware and Software: Formal Approaches to Design and Verification, 2006.

[66] *Satabs & CBMC user manual*. ETH Zürich. Available from the SATABS homepage: `http://www.verify.ethz.ch/satabs/`

[67] I. Satoh and M. Tokoro. *Semantics for a Real-Time Object-Oriented Programming Language*. In Proceedings of IEEE Conference on Computer Languages, pp. 159-170, 1994.

[68] I. Satoh and M. Tokoro. *A Formalism for Real-Time Concurrent Object-Oriented Computing*. Proceedings of ACM OOPSLA'92, pp. 315-326, 1992.

[69] B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin and W. Tu. *Model Checking An Entire Linux Distribution for Security Violations*. ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference, IEEE Computer Society, pp 13-22, 2005.

[70] M. Stokely, S. Chaki and J. Ouaknine. *Parallel Assignments in Software Model Checking*. Proceedings of the 3rd International Workshop on Software Verification and Validation (SVV), vol. 157 of ENTCS, pp. 77-94, 2006.

[71] U. Ultes-Nitsche. *Do We Need Liveness? - Approximation of Liveness Properties by Safety Properties*. In Proceedings of SOFSEM'02, vol. 2540 of LNCS, pp. 279-287, 2002.

[72] Y. Xie and A. Aiken. *Saturn: A SAT-Based Tool for Bug Detection*. Computer Aided Verification, vol. 3576 of LNCS, pp 139-143, 2005.

[73] Y. Xie and A. Aiken. *Scalable Error Detection using Boolean Satisfiability*. POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 351-363, 2005.

[74] UNO homepage: `http://www.spinroot.com/uno/`.

[75] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula and E. Brewer. *SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques*. To appear in the Proceedings of OSDI, 2006.

Statement of Academical Honesty

Hereby, I declare that the work of this Master's thesis is completely the result of my own work, except where otherwise indicated. All the resources I used for this thesis are given in the list of references.

Dresden, 7. Mai 2007

Thomas Witkowski
Matr. No: 2949565