Master Thesis

Modeling and verifying embedded operating systems

Author(s):
Mack, Raphael

Publication Date:
2008

Permanent Link:

https://doi.org/10.3929/ethz-a-005575233 →

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis

Modeling and Verifying Embedded Operating Systems

Raphael Mack
03-906-542

Supervisors: Prof. Dr. Daniel Kröning
Georg Weissenbacher
Winfried Janz

Zuerich, April 3, 2008

Contents

1 Introduction

1.1 Goal

The goal of this work is to describe a method to model embedded operating systems and to formally verify the consistency of that model using model checking techniques. We applied the suggested approach to parts of an existing, industrial implementation of the OSEK/VDX standard for operating systems, and actually found differences between the documentation and the implementation. An application program, combined with the operating system, is not given during the verification phase and may even contain application specific interrupt service routines. These may concurrently change the systems state. The presented method is able to detect violations of pre- and postconditions, assertions and data invariants. Our approach can be used to specify the functionality of interrupt service routines and will detect bugs based on data races provoked by them. Although very important in industrial settings, timing aspects and termination are left aside.

The system model is given in the language described in Chapter 5.2, which allows to specify the observable behavior for the API and give a model implementation for all procedures. The proposed tool parses the specification and the model implementations and generates annotated C code, which is verified with the model checking tool CBMC in a fully automated manner.

Even though the proposed specification language and tools can be used to develop new systems using a *verify-while-develop* paradigm [15], we focus on *a posteriori* program verification of systems implemented in C and assembly languages. The considered models must be finite-state.

1.2 Structure of this thesis

In Section 2, we give an introduction to the OSEK standard for embedded operating systems. After the introduction of related concepts (Section 3) we propose our approach in Section 4, which combines several well-known concepts. Section 5 gives the syntax for our modeling language, which can be formally verified by the tool described in Section 6. We applied the proposed method to parts of an industrial implementation of the OSEK OS standard and give a few examples of properties we modeled and verified in Section 7. Finally, we discuss the results of this thesis.

2 OSEK/VDX Operating Systems

2.1 Introduction

OSEK[1]/VDX is a joint project of several players of the automotive industry. The OSEK work groups defined a set of standards for software components for embedded automotive applications. The OSEK OS Specification [20] defines a class of operating systems, for which the following chapter gives a brief introduction. Further details can be found in [42, 29].

OSEK systems aim at embedded real time applications, mainly found in automotive electronic control units (ECU). As these often have little resources (by means of memory footprint, power consumption and clock rates) available the implementations are highly optimized and configurable. The application program and the operating system are compiled and linked together. No dynamic creation of system objects, i.e., tasks, events, alarms, or resources, is supported. In contrast to time-triggered operating systems, scheduling of tasks is based on (static) priorities and is triggered by events.

Two modes of operation are defined by OSEK. "Standard Status" mode which is highly optimized for size and performance and "Extended Status" mode which carries out extensive error checking. Another setting, which is chosen at compile time, is the conformance class, which defines, whether the system supports tasks with a WAITING state (in ECC1 and ECC2) and whether multiple activations of the same task are allowed (in BCC2 and ECC2). The system may contain preemptive tasks, non-preemptive tasks or both, according to a configuration option.

The application program may not directly manipulate operating system variables, but has to call API functions to trigger the intended behavior.

2.2 Tasks

Each task has a priority and a statically assigned set of system objects it may use. These include events, alarms, and resources.

OSEK defines two different task concepts:

- basic tasks
- extended tasks

[1] OSEK is a registered trademark of Siemens AG

2.2.1 Basic Tasks

A basic task is in one of the states RUNNING, READY or SUSPENDED. Activation of a task transfers it into the READY state and triggers the scheduler, which then will transfer the ready task with highest priority into the RUNNING state and preempt the currently running task. Each task may activate other tasks with the system call ActivateTask() or terminate itself by a call of TerminateTask() (no asynchronous termination of tasks is supported). Termination transfers the currently running task into the SUSPENDED state and triggers the scheduler to execute the next ready task or an internal idle loop if there is no such task. On system startup every task is initialized either in the READY state, if it has the autostart property set, or else in the SUSPENDED state.

2.2.2 Extended Tasks

Extended tasks additionally may be in a WAITING state, to implement the event mechanism described in 2.4. The system call WaitEvent() will cause the running task to enter the WAITING state, while SetEvent() may transfer the receiver task from WAITING into READY state. An extended task may clear its own events by calling ClearEvent().

2.3 Interrupts

Two categories of interrupt service routines (ISR) are defined.

ISR category 1 (cat1-ISR) are not allowed to use the operating system API[2]. Therefore they do not have any effect on the operating system state and do not influence task management. Execution will continue after a cat1-ISR exactly at the instruction where the interrupt occurred.

ISR category 2 (cat2-ISR) may call OS API functions[3]. Therefore a rescheduling may be initiated by the generated ISR epilogue.

Depending on the implementation and the platform several interrupt priorities may be chosen for different ISRs. The OS provides several system calls to enable and disable interrupts. Either the status of only category 2 or all interrupts may be changed.

2.4 Events

Events provide a way for synchronization between different tasks or ISRs (only cat-2). Like all system objects, events are statically allocated and assigned to an extended task. On activation of an extended task, all its events are cleared and the task can wait for them by calling WaitEvent(). This system call will transfer the task into the WAITING state and return as soon as some other task or ISR has set one of the expected events and the scheduler has decided to execute the task, based on its priority.

[2]Exceptions are the system function to enable and disable interrupts.
[3]for a exact list of the allowed subset see [20], figure 12-1

2.5 Alarms

OSEK defines the concept of alarms for processing of recurring actions as timer ticks or increment of an camshaft axle, which increment a counter. An alarm will expire when a particular counter value is reached, which may be given statically or set via provided system calls at runtime. On expiration, a statically defined action, like activation of a specific task, setting an event, or executing a callback function will be triggered. Alarms may be setup as single or cyclic alarms, while the former will only expire once and the latter will be reinitialized on every expiration.

2.6 Resources

The resource concept is used to coordinate concurrent access of different tasks to the same resource. A resource is a binary semaphore. It can be used to encapsulate critical sections to ensure that only one task at a time enters a section protected by the same resource. The OSEK Specification [20] requires the system to use the Priority Ceiling Protocol (PCP, introduced in [40]) for resources. Therefore it is guaranteed, that no uncontrolled priority inversion [40] and no deadlocks occur by the use of these resources. For a formal analysis of PCP consult [16].

Some implementations allow the sharing of resources not only between tasks, but also with ISRs (cat-2).

2.7 Hook Routines

The system may be configured to call several hook routines. These are routines implemented by the application program and called by the operating system in defined situations. These contain the occurrence of runtime errors, before and after a task switch, and on system startup and shutdown. Hook routines may be used, e.g., for debugging, timing measures, or to write runtime error logs. Hook routines are called with disabled interrupts and may only use a subset of the operting system API.

3 Theoretical Background

3.1 Model Checking

Model checking [17] is a method to formally check a system against its specification, usually given in a temporal logic like LTL [34]. The system to be verified is modeled as a finite state machine, where its specification is checked for all reachable states. There are two approaches for this state exploration, explicit state based (e.g., used by the model checker SPIN [21]), constructing and storing a binary representation of each state, and symbolic based using a symbolic encoding for the set of reachable states and the property to be checked [25]. The program specifications define the set of illegal states and the program is faulty, if one of them is reachable. In contrast to theorem proving approaches, model checking is highly automated, and no interaction is needed during the verification. The main issue with model checking is the problem of state explosion, which limits this technique to small systems with a few million states.

Bounded Model Checking (BMC), as proposed by Biere et al. in [5], is a symbolic approach which suggests to use a decision procedure for propositional satisfiability (SAT) to search for illegal but reachable states in program executions only up to a certain length k. For all finite systems an upper bound for the length can be given and therefore bounded model checking does not miss any execution trace that violates the specification.

3.2 Modular Reasoning

Many software systems are composed of several components, which may be developed independently. Therefore, a formal verification system should be modular in a way such that each component can be specified and verified without the need for additional information about the interior construction of other components [15]. In a modular specification setup:

- a program can be developed by the use of modules of which one only knows their specification, without any knowledge of their implementation;
- a module can be implemented on the pure basis of its specification, without any knowledge about its use.

All facts an implementation or a program using a module relies on, must be explicitly addressed in its specification.

To prove a modular system correct the assertional language used to encode a modules specification should be able to express the complete observable behavior of a module. Furthermore, the reasoning framework should provide rules to infer that each module

satisfies its specification and to deduce the correctness of the complete system composed from correct parts. Such formalisms are given in [2] and [15].

A system is called to be *open* if it works in, and interacts with an unknown environment. For some components no implementation may be given or even their specification may be reduced to the bare minimum of environment assumptions. In an completely arbitrary environment it would be impossible for a module to give any guarantee. Therefore the system specification needs to include some explicitly stated, global environment assumptions which hold for each component.

3.3 Rely-Guarantee Reasoning

In concurrent systems, the execution of several threads may be interleaved arbitrarily. Therefore, the specification of a program needs to include statements about the interference of the program with its environment. Rely-Guarantee Reasoning was introduced for message passing systems in [31] and by Jones for shared variable concurrency [22, 23]. A specification of a thread is given in the form (*pre*, *rely*, *guar*, *post*). The predicates *pre* and *post* are the thread's pre- and postcondition. The precondition formalizes the assumptions of the program about its initial state. Executing the program starting in a state not satisfying its precondition is considered as illegal use of the program and it does not give any guarantees then. The postcondition relates the initial and the terminating state and describes the behavior of the program. *rely* and *guar* impose conditions for state changes done by the environment and the thread itself.

Informally, a program P satisfies such a specification if

1. P is invoked in a state which satisfies *pre*, and
2. any state change done by the environment satisfies *rely*

   then
3. any state change done by P satisfies *guar*
4. if the computation terminates, the final state satisfies *post*.

Parallel composition of such programs, i.e., arbitrary interleaving of their atomic statements, is valid if each thread's *rely* condition is satisfied and each thread's *guar* condition implies the other's *rely* conditions. For a summary on the Rely-Guarantee method see [44].

4 Method

4.1 Modeling the System

Practical embedded system code often heavily uses inline assembler, compiler pragmas or dialects which do not adhere to the ANSI C standard. Therefore we require the developer to give a complete model for the system, to which we apply formal methods. The model is formally verified using model checking techniques. To ensure the correctness of an implementation a thorough manual review has to be made.

The model is given in terms of annotations within the code of the implementation, limiting the amount of code needed to be inspected for the review. The annotation language is defined in Section 5.2 and is written within C comments, to be able to compile the annotated system without the need of additional tools.

To verify integrity of the model, the annotations are extracted from the implementation and proof obligations are generated in form of instrumented C code. This transformation is done by a tool described in Section 6. The instrumented C code is passed to the Bounded Model Checker CBMC, which checks the extracted proof obligations.

4.2 CBMC

CBMC is a tool for Bounded Model Checking of ANSI-C programs, introduced in [11] and [12]. It checks for pointer safety, array bounds and user defined assertions by generating a Boolean formula expressing the program behavior and the negation of the specification. Using a SAT procedure, it checks the formula for satisfiability and generates a counterexample trace in case of a violation of the specification. While function calls are handled by inlining, recursion and loops are unwound until all longer traces become infeasible or the user supplied unwinding depth is reached.

Arithmetic is handled by expressing the operations on the bit level, allowing to properly represent arithmetic overflow and all bit level operations, which are heavily used in low level system software. The Boolean expressions for multiplication and division are rather complex and the resulting formulas hard to solve for the SAT procedure. As these are used rarely in the targeted systems, we did not experience any problems of that sort.

4.3 Theoretical Foundations

The system is viewed as a finite number of global variables, encoding the state space, and a number of procedures, modifying the system state. A procedure is a (finite) sequence of statements of a C like programming language, a number of local variables, procedure

parameters and an optional return value. Local variables, parameters and the return value define the local state space of the procedure and cannot be changed by another component, even if executed concurrently as an interrupt service routine.

4.3.1 Model Implementation

Each procedure may have a model implementation, which is a white box model of its behavior. It is a direct translation of the actual implementation of a function to the formal model. It is possible to extract this from the source code automatically, but as we apply the suggested approach to a system with large parts written in assembly language this would go beyond the scope of the current thesis. The annotation language defined in Section 5.2 uses the same syntax and semantics for statements as ANSI C. Therefore, it is straight forward to annotate an actual implementation.

If no model implementation is given for a particular procedure, no proof obligations are generated for it, but the specification is only used to verify other procedures. Therefore it is possible to develop a specification for an existing system and specify procedures which are not in the scope of the inspected system (i.e. library or callback function).

4.3.2 Statements

The supported statements to formalize the model implementation include assignments, conditionals, loops, and procedure calls from the language C with usual semantics. In addition assertions and assumptions may be used with the semantics that assertions introduce additional proof obligations into the model. An assumption terminates the program successfully, if its conditions fails.

The execution of a C statement is in general not guaranteed to be atomic. The compiler may generate code which stores intermediate results of an evaluation and fetches the involved operands in arbitrary order. Yet, we consider every statement as atomic. This may require to introduce temporal local variables in the model implementation to capture the actual semantics of the used language, hardware and compiler. To ensure that no data race stays undetected, it is sufficient to follow the principle that every assignment and test should contain at least one shared (global) variable.[1] Other formal approaches, as TCBMC [37], achieve this by transforming the code into an atomic equivalent in a preprocessing step. Note, that this also applies to initialization, incrementation and loop conditions in all kinds of loops as well as expressions in conditionals.

There are two general approaches to deal with procedure calls: inlining and behavioral abstraction. Inlining does not scale to large systems [19] and can only be applied to closed systems, where the implementation of each procedure is given. Therefore, we replace each call to a procedure by its specification to verify the calling procedure.

[1] This is sometimes called Reynolds' Criterion [15, 32, 24]

4.3.3 Modularity

To improve the scalability and to allow reasoning about open systems our verification approach needs to be *procedure-modular* and *thread-modular* [19]. We gain procedural modularity by replacing a procedure call by the specification of the called procedure. This consists of a precondition, a postcondition and a modifies clause. The precondition is given as a predicate over the global state and the parameters of the procedure. It imposes obligations for the caller of the procedure, as the procedure must not be called in a state not satisfying its precondition. The postcondition is a predicate over two versions (the values in the pre-state, just before the execution of the first statement and the values in the post-state, right after the last statement) of the global variables, the parameters and the return value of a procedure. It formally describes the behavior of the procedure by relating the values of parameters and global variables before the call and the return value and global variables after the call. The modifies clause gives a list of all global variables the procedure may modify during its execution. This behavioral specification of an procedure can be seen as a contract between implementor and the programmer who uses a procedure, as suggested by L. Lamport in [26]. Such an approach is also used in the Design-by-Contract methodology of Eiffel [30], JML [27], and many other systems. It is common practice in software engineering that such specifications are given informally and often incomplete in natural language. In Section 5.2, we define a language to give them in a machine checkable formalism. The modular approach also allows to exchange the implementation of a procedure by a new one, as long as the new one satisfies the same (or a stronger) specification. Such replacements do neither break the system, nor require the re-verification of the whole system.

Being procedure-modular yields a simple answer of how to handle library function: giving a specification for their behavior in our annotation language is sufficient.

4.3.4 Interrupts

All major processor architectures used in embedded systems support interrupts, which allow to implement a limited form of concurrency in such systems. During the execution of a program, the processor may be requested to execute an interrupt service routine (ISR). Whenever interrupts are enabled and such a request is pending the current execution will be interrupted and the ISR is processed. When the ISR terminates, the processor will resume the execution of the interrupted program. In the proposed approach the *interrupt condition* is a predicate over the system state, formalizing the processors conditions to enable interrupts. The procedures executed as interrupt service routines are modeled as *interrupt procedures*, whose execution may change the global state of the system by assigning values to global variables. The local state of the interrupted procedure is not affected.

Such interrupt systems can be modeled by general approaches to concurrency [38] but differ in the following way:

- No arbitrary interleaving of statements between interrupt and non-interrupt programs, i.e., the execution of the interrupting procedure is finished, when the inter-

rupted one continues. In other words, interrupts are executed in an atomic manner, w.r.t. the interrupted program.

- There is a mechanism to prevent the execution of an interrupt procedure by disabling interrupts.
- There is an arbitration/prioritization mechanism ensuring a bounded depth for interruptions.

Verification of these systems is performed by transforming the concurrent program with interrupts into a sequential program which simulates the interrupt behavior. As the system may be composed with interrupt procedures provided by the application, we need modularity w.r.t. interrupts, and we cannot generate calls directly to the interrupt service routines (as done in the Avinux project [35]), because they are unknown. From the rely-guarantee method we borrow the idea to formalize the state changes which can be done by the environment and generate calls which simulate all possible behaviors of interrupt procedures. Which state changes are allowed to be made by interrupts (the environment) is given by a number of two-state-predicates which we call **dynamic invariant**. Jones used a similar concept to specify possible environment transitions for general concurrency [24]. While in the rely-guarantee approach every thread states the assumptions about the environment and the commitments it gives to its environment, we formalize one environment specification which applies to all components. Therefore we do not need to reason that all relies are satisfied by the other components' guarantees.

Every interrupt procedure integrated into the system has to ensure that its pre- and poststate satisfy this dynamic invariant. On the other hand, each procedure has to tolerate every concurrent state change according the dynamic invariant when the interrupt condition holds, i.e., interrupts are enabled.

Many processors do not give any guarantees about the execution of multiple interrupt procedures or may execute the same procedure multiple times before resuming the interrupted program. Therefore, we require the dynamic invariant to be transitive (i.e. if it holds for a interrupt procedure, it must also hold for the initial state of the first and the final state of the last execution of a sequence of several interrupt procedures).

The execution of interrupt procedures is not triggered by a call statement but directly by the underlying hardware. Therefore, the meaning of its precondition is slightly different from non-interrupt procedures. It formalizes the condition for the interrupt request guaranteed by the hardware, and not the prerequisites for a caller to be "allowed" to call the procedure. If an interrupt procedure makes some assumptions about the system state, these have to be guaranteed by the invariant conjuncted with the interrupt condition. Interrupt procedures may not take any parameters and have no return value.

4.3.5 Data Invariants

For most practical systems, the global model variables will span a state space of which not every state is admissible. For example, a variable may only be allowed to have a value in some restricted range, or several global variables may be related to each other. The

formalization of these constraints is given by a number of predicates, that, conjuncted, form the (static) invariant. Each procedure may rely on the invariant to hold on its entry and has to guarantee the invariant during it's execution whenever the interrupt condition is satisfied and on its termination.

4.3.6 Constrained Model Constants

Many operating systems are configurable by a set of numbers that do not change during the runtime of the system but are given by the application program, e. g., the number of tasks in the system or their static priorities. In our model they are represented by model constants, which are declared similar to model variables by a type and a name. For the verification they are once chosen nondeterministically and assumed to be fixed during the runtime of the system. Most such model constants may not be chosen completely arbitrary by the application, but only within some restricted range or according some dependency to other constants. Such restrictions may be given by constraints, which are predicates over the model constants. As with many formal specifications, the user has to ensure, that the constraints are satisfiable, or else the verification will vacuously succeed.

4.4 Preprocessing

Practical embedded system code often uses preprocessor directives to gain small and fast executables without the loss of flexibility. Our verification framework does not directly support the use of preprocessor directives but requires to preprocess the code before passing them to the verification tool. Annotations may be written in #ifdef sections, to be able to specify compile time conditional behavior. Only one such configuration is verified each run, but it should be straightforward to create a tool enumerating all combinations of preprocessor configurations. If the exponential blow-up cannot be handled as result of the number of configuration options it may be feasible to verify only an interesting subset or even only those configurations actually used for an application. Because the annotations are written within C or assembly comments the preprocessor has to be configured to preserve comments.

System software often contains macros appearing to be function calls. We suggest to annotate these as it would be real calls, and to introduce a procedure for the macro definition, specifying its behavior and giving a model implementation. As it may be useful to embed annotations for procedures in a header file, which is included in several source files, our tool is prepared to detect such multiple definitions and handle them appropriately.

4.5 Correctness

To gain evidence for the correctness of the inspected system, we do not only want to formally specify its intended and actual behavior, but also formally verify that the model is consistent. The system is considered correct if

- the given model implementations satisfy the specification of any called procedure
- all model implementations satisfy their own specification and assertions
- each procedure preserves the system invariant whenever the interrupt condition holds or another procedure is called
- all interrupt procedures satisfy the dynamic invariant
- these correctness conditions hold even in the presence of interrupts, modifying the system state according the dynamic and static invariants

Conformance of the model implementation with the procedures specification is verified separately for each procedure. To verify an implementation of the system against its model, it has to be guaranteed that the model implementations exactly represent the real implementation code. Because this manual review can be done locally it is much simpler to grasp than to check the whole system manually.

The proposed approach only considers partial correctness, i.e., each procedure will not terminate in a state not satisfying its postcondition, but it may fail to terminate. Even though for sequential programs a termination bound is guaranteed by the unwinding assertions of CBMC, it cannot be guaranteed in presence of interrupts.

4.6 Modelchecking Bounds

Bounded Modelchecking is complete, if counterexamples up to the length of the completeness threshold are considered [14]. CBMC unwinds loops and recursions up to a depth k, which it tries to find by heuristics. If it fails to infer k the user has to give the unwinding bound on the command line. To guarantee completes of the bounded model checking CBMC checks that enough unwinding is done by introducing unwinding assertions. The loops found in the inspected operating system all had a simple structure, and CBMC could find an unwinding bound.

4.7 Limitations

The main limitation of the suggested approach is, that not all lowlevel constructs, i.e. assembly instructions, can be modeled in the C based annotations language. Therefore no arbitrary change of the control flow can be expressed, which is needed to implement a scheduler in a multitasking system, and the scheduler cannot be accurately modeled.

It is not possible to specify and verify liveness properties such as termination or scheduler fairness. We do not include a model of the stack and therefore cannot detect stack overflows.

Not all properties about pointers and arrays which are expressible within the annotation language can be verified. This is due to the over-approximation, used for the pointer analysis performed by CBMC. Therefore properties which lead to assumptions about the address stored in a pointer will not show any effect.

5 Annotations

5.1 General Setup

Using the language described in the current Section the source code of the operating systems is annotated. After processing with the C preprocessor the tool described in Section 6 parses the specifications and model implementations. It generates ANSI C code instrumented with `__CPROVER_assume` and `__CPROVER_assert` statements. This intermediate code encodes the proof obligations to ensure correctness, and is verified by CBMC. A counterexample trace is given, if a specified property is violated. The user has to inspect this and fix the program and model implementation, if a bug was revealed, or improve the specification, if a the counterexample is spurious. Figure 5.1 shows an overview of the complete process.

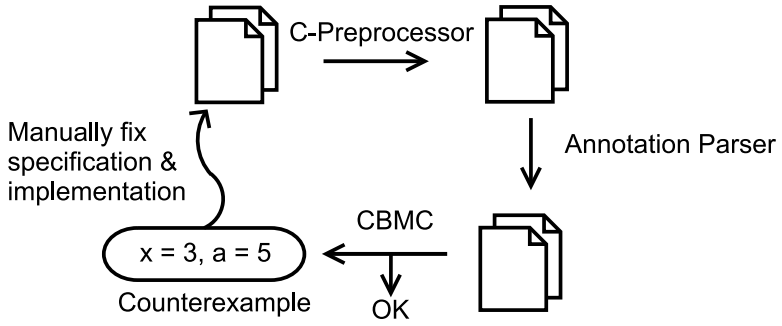

Figure 5.1: Verification Workflow: Overview

5.2 Annotation Language

The annotation language we propose is based on expressions and statements of ANSI C. Annotations are given within special forms of comments to allow a direct embedding in the source code of the annotated system. These comments are introduced by /*@, //@, or ;@. The latter two are defined to end at the end of the line, while the first is terminated by @*/. The end of this annotation comment also terminates the annotation statement. All assertions, namely pre-, postconditions, static and dynamic invariant and constant constraints, are Boolean expressions. They may not contain function calls or have side effects. Given this syntax, it is possible to compile the annotated code without

any further preprocessing and to extract the annotations to check them for consistency. The annotations can be removed from the source code before the delivery.

At the top level, the model is a set of global model variables, system invariants, procedures, dynamic invariants and the interrupt condition. Values, which are fixed during the runtime of the system but may differ from one application to another, are represented by model constants and constraints. This is just to improve the performance of the verification, as these values could also be modeled as variables and invariants to constrain their range.

Model ::= *InterruptCondition Annotation**
Annotation ::= *VariableDecl* | *ConstDecl* | *StaticInvariant* | *DynamicInvariant*
| *Constraint* | *Procedure*

5.2.1 Language Constructs

Model variables span the global state space of the system.

VariableDecl ::= **model** *Type Identifier*

Model constants declare arbitrary but fixed values. They are used to model constants which are application specific and therefore not known in the system code (for example task priorities).

ConstDecl ::= **const model** *Type Identifier*

Constraints are Boolean expressions over model constants. They restrict the values of the model constants.

Constraint ::= **constraint** ‘:” *Expression* “:” *Description*

Description gives a human readable explanation of the system property formally described by *Expression*, and may not contain the character “:”.

Static invariants are Boolean expressions over the global model variables which hold for any legal global state. Each procedure may rely on the invariants to hold in its pre-state, while it has to preserve them whenever the interrupt condition holds, or the procedure terminates.

Invariant ::= **invariant** “:” *Expression* “:” *Description*

Description gives a human readable explanation of the system property formally described by *Expression*, and may not contain the character “:”.

The Interrupt Condition is a Boolean expression over the global model variables. Whenever the system is in a state satisfying the interrupt condition, the currently executed procedure may be interrupted by the execution of an interrupt service routine. This condition is fixed by the hardware architecture and the model for its interrupt enabling mechanism.

InterruptCondition ::= **interrupt condition** “:” *Expression* “:” *Description*

Description gives a human readable explanation of the system property formally described by *Expression*, and may not contain the character ":".

Dynamic invariants are Boolean expressions over global model variables relating two states. The dynamic invariant specifies all legal state transitions of interrupt procedures. For predicates about the pre-state the **\old** statement may be used as a subexpression.

DynamicInvariant ::= **dynamic invariant** ":" *Expression* ":" *Description*

Description gives a human readable explanation of the system property formally described by *Expression*, and may not contain the character ":".

Procedures define the API of the system, internal helper and library functions. They may be called whenever the procedures precondition and the invariants hold. The conjunction of the **requires** clauses give the procedure's precondition, the conjunction of the **ensures** clauses its postcondition. An procedure declared as **interrupt** may be executed spontaneously as an interrupt service routine.

Procedure ::= (**procedure** | **interrupt**) ":" *Type Identifier*
"(" [*ArgList*] ")" *Spec* VariableDecl* Behavior**
ArgList ::= *Type Identifier* ["," *ArgList*]
Spec ::= **requires** ":" *Expression* ":" *Description*
| **modifies** ":" *IdList*
| **ensures** ":" *Expression* ":" *Description*

Identifier must be distinct from all other procedures, global model variables and model constants. *Identifier*s in the parameter list must be different from each other. The *modifies* clause lists all global model variables and out parameters, possibly affected by executing the procedure. If the global variable is an array the syntax *Identifier*[∗] specifies that the complete contents of the array *Identifier* may be modified. If only designated elements may be modified they can be given explicitly as *Identifier*[*Expr*] while it has to be ensured that *Expr* does not have side effects and is invariant during the execution of the procedure. Within the *ensures* clause, **\result** refers to the return value of the procedure. **\old**(*Expression*) will evaluate *Expression* in the pre-state of the procedure, which allows to refer to both, the initial and the final state of the procedure. *Identifier*s from the *ArgList* may be used in the postcondition. The *Behavior* of a procedure may be given as a model implementation. Procedure arguments may not be written to within the model implementation. *Description* gives a human readable explanation of the system property formally described by *Expression*, and may not contain the character ":".

Local variables span the local state of a procedure. They have similar semantics as local variables in ANSI C and may be used in the model implementation.

VariableDecl ::= **local** *Type Identifier*

Model implementation of the procedure in given by ANSI C statements. This is a direct translation of the system code, modeling the direct effect of the procedure on local and global mode variables. The model implementation may contain *Assertion*s, which introduce additional proof obligations.

Behavior ::= **do** ":" *Implementation* | *Assertion*

Implementation is ANSI C code, which may call other procedures, contain loops and assign to global and local model variables. It is forbidden to assign to arguments of the procedure. The model implementation is the code, which is actually verified.

Assertions provide a way to introduce proof obligations within the behavioral specification. *Expression* may contain identifiers of procedure parameters, constants, global, and local model variables. Its evaluation may not show any sideeffects.

Assertion ::= **assert** ":" *Expression* ":" *Description*

Description gives a human readable explanation of the system property formally described by *Expression*, and may not contain the character ":".

All *Identifier*s have to be valid ANSI C identifiers.

5.2.2 Expressions

Any sideeffect-free ANSI C expression without function calls may be used in expressions for the specifications. Within the statements of the model implementation this limitation does not apply. Depending on the context, the expressions may use *Identifier*s of model constants, global and local variables, procedure parameters, and the keyword **\result**. Expressions in postconditions and dynamic invariants may use **\old(***Expr***)** to evaluate a subexpression *Expr* in the pre-state.

To allow universal quantification over a finite range of integers, expressions in specifications may contain the \forall construct with the following syntax:

ForallExpr ::= **\forall** *var* **in** ("["|"(") *lBound* "," *uBound* ("]"|")") ":"
Expression$_{var}$ ":" *Description*

This is a syntactic shortcut for

Expression[var/low] && *Expression*[var/(low + 1)] && ... && *Expression*[var/upper]

\forall allows to formulate assertions for each element of an array. *lBound* and *uBound* must be given as number or a symbol which is resolved to a number by the preprocessor. It is not possible to use model constants as bounds. *var* needs to be a valid ANSI C identifier and is a new local variable with the scope of the \forall construct. Each free occurrence of *var* in *Expression* will be bound to the loop counter going from *low* to *high*. Any variable with the same name is hidden. *low* is equal to *lBound*, if the opening brace is a "[" and *lBound* + 1 if preceded by a "(". *upper* is equal to *uBound*, if the closing brace is a "]" and *uBound* - 1 if preceded by a ")". This is the widely used syntax for integer intervals in mathematics.

5.2.3 Types

As data types for model variables, constants, procedure parameters, and return values the types defined by ANSI C may be used. In Addition the type `_Bool` for truth-values is supported. Pointer types must explicitly use the `*` symbol and may not be hidden in a typedef.

5.3 Example

In Listing 5.1 we give a simple example of a model for a sequential program, i.e., without any interrupt service routines. The program stores width, height and area of a rectangle and provides an procedure to increase its width. The invariant $area == width * height$ which we labeled "correct area" specifies a dependency between the values of width, height and area. The procedure is specified to increase the value of width and to only assign values to the global variables width and area, which states, that height will remain unchanged. Verification of this model checks whether the model implementation, given in the "do-annotations" satisfies all these specifications.

Listing 5.1: Example

```
1  /*@ model: int width @*/
   /*@ model: int height @*/
3  /*@ model: int area @*/

5  /*@ invariant: area == width * height: correct area @*/

7  /*@ interrupt condition: FALSE @*/

9  /*@ procedure: void increaseWidth(int w) @*/
   /*@ modifies: width, area @*/
11 /*@ ensures: width == \old(width + w): width increased by w @*/
   /*@ do: width = width + w; @*/
13 /*@ do: area = width * height; @*/
```

6 Annotation Parser

During the work on this thesis we implemented a conversion tool which parses the annotations and generates the intermediate verification code, passed to CBMC. It is realized as a command line tool, which integrates into an automated process to allow the verification of different configurations of the inspected system. According the rules in this Chapter, the annotations are transformed into an ANSI C program, and instrumented with assumptions and assertions to express the proof obligations necessary for the correctness of the system. As the correctness of this tool is a critical issue in the verification process an automated test suite has been written and applied. For each language construct tests with at least one succeeding and one failing procedure are contained.

6.1 Model Variables

Model variables are directly adopted into the intermediate code as global variables. All global model variables are initialized nondeterministically by a C function `__choose_state()`. Calling this function will choose an arbitrary system state.

6.2 Invariant

From the invariant two C functions are generated: `__assert_invariant()` and `__assume_invariant()`. The former one will introduce proof obligations for the invariant, while the latter introduces the assumption, that the invariant holds. Calls to these functions are generated at all locations, where the program may rely on the invariant to hold or has to ensure the invariant respectively.

6.3 Model Constants and Constraints

For every model constant a global variable will be generated in the CBMC code. As it does not change during the systems runtime it will once be chosen nondeterministically in the `__init_consts()` function. After choosing a value for the constants, assumptions are generated for the constraints to rule out all combinations of constants not satisfying the constraints.

6.4 Interrupt Simulation

To simulate the occurrence of interrupts a C function `__int_model()` is generated. It does the following:

1. choose a Boolean value and return if the outcome is `TRUE`
2. assume the interrupt condition
3. assert the (static) invariant
4. choose an arbitrary system state (by calling `__choose_state()`)
5. assume the (static) invariant
6. assume the dynamic invariant

A call to this function is generated after each atomic statement in the model implementations.

6.5 Procedures

For each procedure the following C functions are generated in a file verify_<procedure>.c with "<procedure>" replaced by its name .

verify<procedure> contains the code from the model implementation (including assertions), instrumented with calls to `__int_model()` after each atomic statement.

main is the entry-point for the verification of the model implementation of the considered procedure. After initialization of the model constants, an arbitrary system state and parameters for the procedure are chosen. Invariant and precondition are assumed and the expressions needed in \old are evaluated and stored in local variables. The function _verify_<procedure> is called and afterwards the invariant and postcondition is asserted. If the considered procedure is an interrupt procedure additionally the dynamic invariant is asserted.

main_modify is the entry-point to verify that a model implementation only modifies global variables, which are mentioned in its modifies-clause. It initializes the model constants and chooses system state and parameters according the invariant and precondition. Then the model implementation is inlined, which is instrumented with assertions stating that all global variables, not in the modifies-clause still have the same value as before. [1]

<procedure> is a function with the same signature (i.e., return type, name and parameter declaration) as the procedure. It is a stub, which simulates the behavior of the procedure according its specification. It does so by asserting invariant and precondition, choosing any modified global variable nondeterministically and assuming invariant and postcondition. This function may be called by the model implementation of other procedures.

[1] This seems to be non-modular, as all global variables need to be enumerated and therefore known. Actually this is not a problem, as it is guaranteed by the lexical analysis of CBMC that no unknown global variable is changed, even if the application program contains additional variables.

6.6 Expressions

6.6.1 forall

A forall expression will be translated into a C loop, increasing the loop variable from the lower to the upper bound and asserting or assuming the subexpression given in the forall.

6.6.2 old

Subexpressions included in \old() are evaluated in the prestate of the current procedure. Before calling the code generated from the model implementation a unique local variable[2] prefixed by _old gets the value of the subexpression assigned. The \old construct then gets replaced by the local variable with the type of the subexpression within the \old statement.

6.7 Technical restrictions

The current implementation has the following limitations.

- many syntax errors in the annotations are neglected and will be reported by the verification backend CBMC
- it only handles one "for"-loop within a do-annotation, which is no real limitation, since model implementations may be constructed from several do-annotations
- initialization and the first evaluation of the loop condition of for loops are simulated atomically
- conditionals must use a block for their statements
- usage of `elseif` is not supported in model implementations
- \forall-expression cannot be used as operands for Boolean operations
- For failed proof obligations CBMC will report a counterexample for the generated code together with the description of the failed assertion. To identify the bug in the annotated source code the user has to inspect the generated intermediate code.
- parameters of a procedure may not be modified in model implementations
- only one-dimensional arrays with statically fixed size may be used
- structs are not supported
- no typedefs for pointer types may be used, i.e., pointer types must be given syntactically using the symbol *

[2] If the\old-expression occurs within a \forall an array is generated to store all needed values.

7 Application

We applied the suggested method to parts of an industrial implementation of an OSEK OS for the microcontroller C167. We have chosen the following configuration options:

- conformance class is BCC1, i.e., only basic tasks, no event subsystem, no multiple activation of tasks
- resources may not be acquired within interrupt service routines
- no extended resource features (i.e. internal and linked resources) are supported
- status mode is "extended status", i.e. api functions do extensive checks on their parameter values and return error codes on failure
- number of tasks is limited to 16

We did not annotate all hook routines and their calls, and skipped the alarm subsystem and some macros, which are only used for debugging.

7.1 Model

To specify the API functions we want to verify, many of the global variables of the systems are required. We included all these in our system model. An array for task states (`TaskState`) and a field of bits (`QReadyPrios`) are used to mark the availability of a ready task for each priority. Two variables store the index and priority of the currently active task (`ActiveTaskIndex`, and `ActiveTaskPrio`). We modeled the interrupt enabled flag (`IEN`), the interrupt level register of the C167 cpu and the request flag for the interrupt used for the dispatcher. The resource subsystem uses arrays to store the priority and index of the task occupying a resource.

An array of Boolean values records which tasks have been executed as an auxiliary variable. The bit for a task is set when the scheduler dispatches this task. This array is used in specifications to denote that a procedure returns only after executing another task, e.g. on activation of a task with priority higher than the currently executed one.

The configuration of the system gives a priority for each task and a bit mask derived from this priority, which are represented as model constants. The tasks may be configured to be activated on startup by an array of Boolean model constants.

In Listing 7.1 we give an example for the use of auxiliary variables. It shows two API procedures to enable and disable interrupts. These two procedures must be called pairwise, i.e., it is not allowed to call either of them two times successively. `IEN` is a Boolean variable, modeling the processors interrupt enabled flag. As other procedures

may modify `IEN` directly the state of this bit cannot be used in the preconditions to flag whether it is allowed to call the procedure. Therefore we introduce the auxiliary variable `auxInCriticalSection`, which is only used for specification purposes and may only be modified by the model implementation of the procedures Disable- and EnableAllInterrupts.

Listing 7.1: Example for the use auxiliary variables

```
1  /*@ interrupt condition: IEN @*/
   /*@ dynamic invariant: IEN == \old(IEN): interrupts may not change the \
       interrupt enable flag @*/
3  /*@ dynamic invariant: auxInCriticalSection == \
       \old(auxInCriticalSection): interrupts may not terminate within a \
       critical section @*/

5  /*@ model: _Bool IEN @*/
   /*@ model: _Bool auxInCriticalSection @*/
7

9  /*@ procedure: void EnableAllInterrupts() @*/
   /*@ modifies: auxInCriticalSection, IEN @*/
11 /*@ requires: auxInCriticalSection == TRUE: only called when already in \
       critical section @*/
   /*@ ensures: IEN == TRUE: interrupts enbled @*/
13 /*@ ensures: auxInCriticalSection == FALSE: critical section is left @*/
   /*@ do: IEN = TRUE; @*/
15 /*@ do: auxInCriticalSection = FALSE; @*/

17
   /*@ procedure: void DisableAllInterrupts() @*/
19 /*@ modifies: auxInCriticalSection, IEN @*/
   /*@ requires: auxInCriticalSection == FALSE: not called when already in \
       critical section @*/
21 /*@ ensures: IEN == FALSE: interrupts disbled @*/
   /*@ ensures: auxInCriticalSection == TRUE: critical section is entered \
       @*/
23 /*@ do: auxInCriticalSection = TRUE; @*/
   /*@ do: IEN = FALSE; @*/
```

7.2 Data Invariants

We annotated and verified the following data invariants:

- The array of task states contains at most one RUNNING task, which is consistent with the variable storing the index of the active task (`ActiveTaskIndex`) and the bit field storing the ready priorities (`QReadyPrios`).
- The array of task states only contains valid states (i.e., one of the constants RUNNING, SUSPENDED, READY and PRE_READY).
- Variables storing the index and priority of the running task and the ready task with highest priority are consistent with task states, static priorities and occupied

resources. Their values are within the range of valid task indexes or priorities, respectively, and the corresponding tasks are in a state which is schedulable (i.e., not SUSPENDED).

- All bits from the ready masks of the active task and the ready task with highest priority are set in the bit field storing the ready priorities.
- Only bits corresponding to an existing priority are set in the bit field of ready priorities.
- The array with ready tasks contains a valid task index, for all priorities, which are marked as ready.
- A task occurs in the ready task array only for priorities at least as high as its static priority.

Besides these we added many technical invariants, which are not explicitly stated in the documentation of the operating system. Range restrictions of variables which are used as index into an array are a typical example for these.

7.3 Procedure Specifications

We annotated and verified many internal procedures with pre- and postconditions, a modifies clause and a model implementation. These are not part of the OSEK standard and are only used internally within the operating system. One simplified example is the procedure `SchedulePrio` (Listing 7.2). It calculates which task to run next, by evaluating the bit field for the ready priorities (`QReadyPrios`). It requires interrupts to be disabled and modifies the variables `HighReadyTaskIndex` and `HighReadyTaskPrio`. It ensures, that these variables indicate the task with highest priority that is ready, and the procedure will return TRUE iff scheduling is needed, i.e., the highest ready task is different from the currently active task or there is no task ready. Listing 7.3 shows an example where SchedulePrio is called from the API function TerminateTask.

Listing 7.2: SchedulePrio.c

```
/*@ procedure: _Bool SchedulePrio() @*/
/*@ modifies: HighReadyTaskIndex, HighReadyTaskPrio @*/
/*@ requires: IEN == FALSE: disabled interrupts @*/
/*@ ensures: \result == ((HighReadyTaskIndex != ActiveTaskIndex) || \
    HighReadyTaskIndex == NO_ACTIVE_TASK): returns true whenever \
    scheduling is needed @*/
/*@ ensures: HighReadyTaskPrio == NO_ACTIVE_PRIO || (QReadyPrios & \
    0x8000) != 0 || (0x0001 << (16 - HighReadyTaskPrio)) > QReadyPrios: \
    highest ready prio was scheduled @*/
boolean SchedulePrio(void)
{
   unsigned short hiPriorTask;
   /*@ local: unsigned short hiPriorTask @*/

```

```
    if (QReadyPrios == 0)
12  /*@ do: if(QReadyPrios == 0) @*/
    {
14     /*@ do: { @*/
       HighReadyTaskPrio = NO_ACTIVE_PRIO;
16     /*@ do: HighReadyTaskPrio = NO_ACTIVE_PRIO; @*/
       HighReadyTaskIndex = NO_ACTIVE_TASK;
18     /*@ do: HighReadyTaskIndex = NO_ACTIVE_TASK; @*/
       return TRUE;
20     /*@ do: return TRUE; @*/
    }
22  /*@ do: } @*/

24  hiPriorTask = _prior(QReadyPrios);
    /*@ do: hiPriorTask = _prior(QReadyPrios); @*/
26
    HighReadyTaskIndex = QReadyTask[hiPriorTask];
28  /*@ do: HighReadyTaskIndex = QReadyTask[hiPriorTask]; @*/

30  HighReadyTaskPrio = hiPriorTask;
    /*@ do: HighReadyTaskPrio = hiPriorTask; @*/
32
    if (HighReadyTaskIndex != ActiveTaskIndex)
34     /*@ do: if(HighReadyTaskIndex != ActiveTaskIndex) @*/
       {
36        /*@ do: { @*/
          return TRUE;
38        /*@ do: return TRUE; @*/
       }
40  /*@ do: } @*/
    else
42     /*@ do: else @*/
       {
44        /*@ do: { @*/
          return FALSE;
46        /*@ do: return FALSE; @*/
          /*@ do: } @*/
48     }
}
```

Listing 7.3: TerminateTask.c

```
1  /*@ procedure: StatusType TerminateTask(void) @*/
   /*@ modifies: IEN, QReadyPrios, TaskState[ActiveTaskIndex], \
       ActiveTaskIndex, ActiveTaskPrio @*/
3  /*@ requires: ActiveTaskIndex != NO_ACTIVE_TASK : a task must be running\
       @*/
   /*@ requires: IEN == TRUE: interrupts are enabled @*/
5  /*@ ensures: FALSE: procedure does not return @*/

7  StatusType TerminateTask ( void )
   {
9     DisableInterrupts();
      /*@ do: DisableInterrupts(); @*/
```

```
11
    TaskState[ActiveTaskIndex] = SUSPENDED;
13  /*@ do: TaskState[ActiveTaskIndex] = SUSPENDED; @*/

15  QReadyPrios &= ~ReadyPrioMask[ActiveTaskIndex];
    /*@ do: QReadyPrios &= ~ReadyPrioMask[ActiveTaskIndex]; @*/
17
    ActiveTaskIndex = NO_ACTIVE_TASK;
19  /*@ do: ActiveTaskIndex = NO_ACTIVE_TASK; @*/
    ActiveTaskPrio = NO_ACTIVE_PRIO;
21  /*@ do: ActiveTaskPrio = NO_ACTOVE_PRIO; @*/

23  (void) SchedulePrio();
    /*@ do: SchedulePrio(); @*/
25
    EnableInterupts();
27  /*@ do: EnableInterrupts(); @*/

29  Dispatch();
    /*@ do: Dispatch(); @*/
31 }
```

The used compiler provides an intrinsic function `_prior` returning the position of the most significant bit which is set in a processor word. This function is called by one of the annotated procedures. Therefore we specified the procedure according the documentation without giving a model implementation, as it is done with every library function.

In addition to the OS internal procedures, we specified and verified the following properties of API functions:

ActivateTask

The procedure `ActivateTask` does proper error checking, i.e., it returns an error code if called with illegal `TaskID` parameter, disabled interrupts or the task to activate is already running. If a task with priority higher than the currently executed task was activated, the new task is executed before the return of `ActivateTask`. If a task with lower priority is activated, the new task is transferred into the READY state. On return the procedure `ActivateTask` has restored the interrupt enabled state.

TerminateTask

The procedure `TerminateTask` will return an error code when not called by a task (i.e. the interrupt level is greater than 0). It requires to be called with enabled interrupts and the currently active task may not occupy any resources and therefore `ActiveTaskPrio` is equal to the tasks static priority. `TerminateTask` does only return to the caller task if an error occurred.

GetTaskID

The procedure `GetTaskID` does proper error checking, i.e., returns an error code if called with disabled interrupts. The parameter `*TaskId` is used as out-parameter, to which the id of the currently active task is copied.

GetTaskState

The procedure `GetTaskState` does proper error checking, i.e., returns an error code if called with disabled interrupts or with illegal parameter `TaskId`. The parameter `*State` is used as out-parameter, to which the state of task with the requested TaskID is copied. The internal data structure which stores task states may contain an additional state PRE_READY, which is converted to READY by `GetTaskState`.

Interrupt Subsystem

There are the following procedures to enable and disable interrupt handling:

DisableAllInterrupts/EnableAllInterrupts disables and enables all interrupts globally, including cat-1 and cat-2, nesting is not allowed, i.e. calling `DisableAllInterrupts` more than once consecutively without a call to `EnableAllInterrupts` between, is forbidden.

SuspendAllInterrupts/ResumeAllInterrupts disables and enables all interrupts globally, including cat-1 and cat-2, nesting is supported, i.e. calling `SuspendAllInterrupts` more than once consecutively without a call to `ResumeAllInterrupts` between is allowed. The depth of this nesting is counted in a variable, which bounds the depth of the nested calls.

SuspendOSInterrupts/ResumeOSInterrupts disables and enables only operating system interrupts (cat-2) by adjusting the interrupt level. Nesting is handled in the same way as for `Suspend-` and `ResumeAllInterrupts`.

All these procedures update an auxiliary variable, which stores the status of the interrupt API and is used in the specification of other API procedures. The Disable- and Suspend-Interrupt procedures store the interrupt enabled state in the outer most call, which is restored by the corresponding Enable- and Resume-Interrupt call.

GetResource

The procedure `GetResource` requires to be called with enabled interrupts, not within a interrupt procedure and the resource to get is not already occupied. It may not be called with the parameter `ResId` to identify a resource with ceiling priority less than the currently running tasks static priority. It returns an error code when called with a non-existing `ResId` as parameter. If no error is returned, the dynamic priority of the calling task is stored and the priority is adjusted to the ceiling priority of all occupied resources (including the one `GetResource` is currently called for).

ReleaseResource

The procedure `ReleaseResource` requires to be called with enabled interrupts, not within a interrupt procedure. The stored priority for the resource to be released must be valid (i.e., less than the resource ceiling priority and greater than the static priority of the active task) and the resource must be occupied by the currently active task. It ensures, that the stored priority will be restored and the resource will not be occupied anymore after return.

StartOS

The procedure `StartOS` is called by the application program to initialize the operating system. `StartOS` initializes the global variables and hereby establishes the invariants. All other API procedures may only be called after this initialization. The startup procedure does only have one precondition, namely that interrupts are disabled, but it does not rely on the invariant. To model this accurately with the proposed method[1] we introduce an auxiliary variable `invariantEstablished` of Boolean type. All invariants are transformed into an implication `invariantEstablished` $\Rightarrow$ *Inv* and every procedure which relies on the invariant to hold, additionally requires `invariantEstablished`. The startup procedure has to establish the invariant and set the auxiliary variable to TRUE.

Besides the initialization, `StartOS` activates all tasks which are configured as *autostart* and calls the dispatcher. Therefore `StartOS` will never return. Obligations for `StartOS`, i.e., *autostart* actually are activated, the invariant is established, and no resources are occupied by any task are specified as assertions in the model implementation just before the dispatcher is called.

7.4 Dispatcher

Any operating system which can handle several tasks needs a dispatcher (or scheduler) to decide which task to run next, switch the context, and transfer the control to the right location within this task.

In the industrial system we annotated, the dispatcher is implemented as a procedure, which is called from the API procedures, whenever a re-scheduling is needed. This may happen on activation of a task with higher priority than the currently active one, on task termination or when releasing a resource. It is not possible to model such a preemptive scheduler within the semantics of the proposed annotation language, because it uses low level assembly language constructs to jump to the scheduled task. Therefore we specified this dispatch procedure with the properties that hold in the case the dispatcher re-schedules the currently active task, to be able to verify the model implementations of procedures calling the dispatcher. These are:

- After `Dispatch` returned all ready tasks with priority higher than the currently running task have been executed (which is recorded in a auxiliary Boolean array

[1] note, that each procedure may rely on the invariant and has to ensure it on termination

variable).

- After `Dispatch` returned the global variables `ActiveTaskPrio` and `ActiveTaskIndex` have the same value as before.
- During the `Dispatch` call the system state may be modified, as the dispatcher actually may have executed other tasks, which could call arbitrary system functions.
- `Dispatch` will not return, if no task is READY (which happens on task termination).

7.5 Deadlock-Freeness

The priority ceiling protocol (PCP, [40, 8]) ensures that the use of resources does not yield a deadlock. Therefore we specified the involved functions to ensure that PCP is implemented correctly.

GetResource

- GetResources may only be called for resources with ceiling priority larger than the currently running tasks static priority
- dynamic adjustment of the priority of the current running task
- the priority before the dynamic adjustment is stored

ReleaseResource

- ReleaseResource may only be called for resources the running task has actually occupied
- calls of ReleaseResource are in reversed order as the corresponding GetResource calls (i.e. they are properly nested), which is expressed by requiring the dynamic priority of the active task to be equal to the ceiling priority of the released resource and higher than its stored priority
- dynamic priority of currently running task is restored

Dispatcher

- no task with priority less than currently active priority is scheduled

TerminateTask

- On termination of a task it has released all its resources (i.e., dynamic == static prio)

ActivateTask

- the newly activated task does not occupy any resources, i.e., the task has a dynamic priority which is lower than the ceiling priority of all resources it could occupy

StartOS

- no resources are occupied after initialization

Except for the dispatcher we specified and verified all these properties. From the proofs in [40, 33] we therefore conclude, that the modeled procedures for resource handling ensures deadlock freeness, absence of unbounded priority inversion and multiple blocking.

7.6 Hook Routines

Hook routines may only call a small subset of API procedures and may not modify the system state directly by assigning values to system variables. From the OSEK OS Specification and the specification of the allowed procedure calls the specification of the hook routines can be derived. The procedure `ErrorHook` is implemented by the application program, and called by the operating system if an API function is called with illegal parameters or in an illegal state. We specified the procedure `ErrorHook` and annotated the calls in the model implementations. To specify that a procedure calls the `ErrorHook` with the correct error code, an auxiliary variable `auxLastErrorHookStatus` is used. Hook routines are defined to be called with disabled interrupts, which is specified as precondition. Listing 7.4 gives a simplified example of the specification of the ErrorHook procedure and its use in procedure `TerminateTask`.

Listing 7.4: Hook Routine Example: ErrorHook

```
1  /*@ model: StatusType modelLastErrorHookStatus @*/

3  /*@ procedure: void ErrorHook(StatusType ErrorCode) @*/
   /*@ modifies: IEN, modelLastErrorHookStatus @*/
5  /*@ requires: IEN == FALSE: interrupts are disabled @*/
   /*@ ensures: modelLastErrorHookStatus == ErrorCode: ErrorHook called \
       with correct status code @*/
7  /*@ ensures: IEN == \old(IEN): Suspend/ResumeAllInterrupts called \
       pairwise @*/

9  /*@ procedure: StatusType TerminateTask(void) @*/
   /*@ ensures: \old(IntNestingDepth == 0) || \result == E_OS_CALLLEVEL: \
       returns error code if called with IntNestingDepth != 0 @*/
11 /*@ ensures: \result == E_OK || modelLastErrorHookStatus == \result: \
       ErrorHook called with correct status code @*/
   StatusType TerminateTask (void)
13 {
      if (IntNestingDepth != 0)
15    {
         /*@ do: if (IntNestingDepth != 0) { @*/
17       DisableInterrupts();
```

```
        /*@ do: DisableInterrupts(); @*/
19      ErrorHook(E_OS_CALLEVEL);
        /*@ do: ErrorHook(E_OS_CALLEVEL); @*/
21      EnableInterrupts();
        /*@ do: EnableInterrupts(); @*/
23      return E_OS_CALLEVEL;
        /*@ do: return E_OS_CALLEVEL; @*/
25    }
      /*@ do: } @*/
27  ...
```

7.7 Runtime errors

As CBMC has built-in support to check for runtime errors, i.e., pointer safety, array bounds and division by zero, a successful verification of the system model will also ensure these properties.

7.8 Results of the Application

During the application of the proposed method on the industrial implementation we found the following two issues:

- The procedure `StartOS` requires interrupts to be disabled, which is a requirement formulated in the documentation. The direct formalization of this requirement leads to a precondition, which is actually not strong enough for the code to satisfy the invariant. This is because there are two ways to disable interrupts: setting the global interrupt enable flag to FALSE or increasing the interrupt level. The verification showed, that it is actually required to disable interrupts by setting the interrupt level.
- The dispatcher is implemented by the usage of an interrupt service routine to do the actual context switch. We found an example trace that violated the property that activation of a high priority task returns only after executing this task, due to unfortunate timing with the execution of interrupt service routines. Further investigation showed, that this trace was spurious and does not appear in a real execution due to an upper bound of the time between setting a interrupt request and execution of the corresponding service routine, which was not documented in the processors manual.

7.9 Figures

We specified 51 procedures with modifies clauses, pre-, and postconditions and annotated the implementation of 40 of them. The model contains 49 variables, including 19 arrays and 8 model constants, which all have an array type and are restricted by 14 constraint annotations. To express the required data invariants we formulated 81 invariant clauses,

while the state transitions of interrupt procedures are restricted by 27 dynamic invariants. Altogether we wrote 1131 lines annotations.

The verification of these 40 annotated procedure implementations took 46 minutes and 48 seconds consuming a maximum of 308 MB RAM. The average verification time was about 70 second per procedure, while the maximum time for one procedure was about 11 minutes. All measures were taken on a Intel Pentium 4, 2.6 GHz, 512 MB RAM running Microsoft Windows XP.

8 Discussion

8.1 Conclusion

In this thesis we propose a method for specification and formal verification of embedded operating systems. We developed a tool to automatically check safety properties of systems with interrupts, communicating via shared variables. The method is modular with respect to procedures and interrupts. We applied our approach to an industrial implementation and actually found bugs. Although the approach does not cover all low level constructs and therefore cannot be used to model the complete operating system, i.e. no complete specification of the dispatcher is possible, it is a viable method for finding bugs and improving the quality of the system.

As the specification language is based on ANSI C and the verification is done in a fully automated manner, the approach is easy to learn and does not require deep theoretical knowledge in topic of formal verification. To find useful static and dynamic invariants is not a trivial task, yet a programmer familiar with the system under verification should rapidly be able to interpret the counterexample traces given for every violated property.

Our approach naturally scales to larges systems with many procedures and interrupt handlers, as the approach is modular and every procedure is verified separately. The limiting factors are the number of global variables and the complexity of the static and dynamic invariants.

8.2 Related Work

Procedure-modular specification using pre- and postconditions is used in many approaches for other programming languages, including JML/ESC/Java [27, 28], Eiffel [30], and Spec#/Boogie [4] with different levels of dynamic and static checking. The approaches using static checking solve the frame problem by giving a modifies clause, as we do. In Spec# all locations not mentioned in the modifies clause must contain the same value in the final state as in the initial state of a method invocation, while for JML the modifies clause is checked syntactically [9], and therefore even assignments like `x := x` are forbidden. In a concurrent environment the frame must be respected in every intermediate state, therefore we require that each global variable not mentioned in the modifies clause must contain the same value as in the initial state, whenever the procedure may be interrupted, i.e. the *interrupt condition* holds, and on termination. Borgida suggests [6] to give the modifies set only implicitly by the convention, that only variables mentioned in the pre- or postcondition are modified.

Pilling, Burns, and Raymond prove [33], that the Priority Ceiling Protocol guarantees

the absence of deadlocks, unbounded priorty inversion (transitive blocking) and multiple blocking. They specified the Original Priority Ceiling Protocol as presented by Sha et al. [40] which adjusts the dynamic priority only if an actual block has occurred. Therefore this proof does not directly apply to the implementation in the system we verified, as there the Immediate Priority Ceiling Protocol (IPCP [8]) is used. IPCP adjusts the priority with every resource usage, which simplifies the implementation without affecting the worst-case behavior [8].

Waszniowski and Hanzálek [43] use timed automata to model an OSEK application and verify TCTL properties using the UPPAAL verifier. They work on an abstract model and check timing constraints, while we work in close relation to the source code and cannot give timing properties. Although we focused on verifying the implementation of the underlying operating system, our approach can also be applied to applications.

Interrupt driven systems are examined by Brylow, Damgaard and Palsberg [7]. They use model checking to do static analysis of stack usage, interrupt latency and typechecking of stack elements. They work directly on the assembly language and included a fixed model of the processor into their verification tool. Their approach is not modular, as they inspect the complete program at once. A general approach for reasoning about interrupt system using general approaches to concurrency is described in [38]. We performed the source-to-source transformation they give and applied the verification tool SATABS [13] to the resulting code with multiple threads. This approach did not scale and it was not possible to verify any interesting property, which probably is because predicate abstraction does not fit well the properties we want to verify. TCBMC [37] is an extension of CBMC which handles multiple threads. It is a promising approach which could verify multithreaded code obtained from the same transformations. As the tool is not publicly available we could not apply it. Our approach to handle interrupts is similar to the *Preemption Simulation* used in the AVINUX project [35]. TCBMC [37] and KISS [36], both use a non-modular approach to concurrency.

General approaches to concurrency are more complex than the interrupt simulation we use, as they need to consider arbitrary interleavings. In our model the interrupting procedure is executed completely before the interrupted one is resumed. This is also the reason, why we do not need to apply the full rely-guarantee method [23] and it suffices to guarantee $\mathit{InterruptCondition} \Rightarrow \mathit{StaticInvariant}$ instead of the dynamic invariant in every intermediate state of all interruptible procedures.

MAGIC [10] uses Labeled Transition Systems, a form of abstract state machines, as specification of procedures. They use a general, modular approach to concurrency using blocking message passing, rather than communication via shared variables.

SDV [3] and Avinux [35] give the specification in the languages SLIC and SLICx. Their approaches are not modular, do not handle interrupts and do not provide a way to verify data invariants.

In the Caduceus tool [18] correctness assertions over C programs are compiled into an intermediate programming language for which interactive and automated prover backends are available. The assertion language they use is very similar to ours, but they do not handle concurrency or interrupt semantics. They also cannot detect bugs related to arithmetic overflow.

As part of the Verisoft project [1] a Hoare calculus and formal semantics of the C subset C0 on top of Isabelle/HOL were developed [39], however, their verification of programs is not completely automated.

The Assertion Checking Environment (ACE, [41]) allows to specify C functions with assertions, invariants, pre- and postconditions. Their approach does not model interrupts and modularity is only supported manually, as each call to a external function needs to be annotated. Locations modified by a function are not specified, which may lead to a vast number of postconditions stating no-modification of variables.

8.3 Future Work

In many cases the model implementation can be generated from the actual system code, or even from compiled binaries. A further improvement of the suggested method would be to implement a tool, extracting model implementations from the actual system code.

Our interpretation of atomic statements is limited and requires the user to add auxiliary variables to represent the correct atomicity of annotated expressions of the C language. This transformation could be done in a preprocessing step (as performed by TCBMC [37]).

To ensure that the specification is not only vacuously satisfied, e.g. by giving a invalid invariant, a helpful improvement would be to automatically verify the satisfiability of the preconditions and the static invariant.

8.4 Acknowledgments

There are a lot of people who have made my studies and this thesis not only possible, but also an interesting and instructive time. First of all I want to thank Annick for her support during the whole time I spend on this thesis. Rainer Künnemeyer, Winfried Janz and Marcel Läzer always assisted me in understanding the internals of the operating system I applied the suggested approach to. They always listened to my problems and answered my questions in a very friendly and helpful way. During my work, Georg Weissenbacher reviewed my manuscript, gave me a lot of helpful comments and pointers to related research. Many discussions about theoretical issues of formal verification deepened my understanding of the topic. I would like to thank Prof. Daniel Kröning for the opportunity to write this thesis in industry.

Bibliography

[1] The Verisoft Project, http://www.verisoft.de, February 2005.

[2] Krzysztof R. Apt. Ten years of hoare's logic: A surveypart i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[3] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *EuroSys '06: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 73–85, New York, NY, USA, 2006. ACM.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview.

[5] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[6] Alexander Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *Software Engineering*, 21(10):785–798, 1995.

[7] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *International Conference on Software Engineering*, pages 47–56, 2001.

[8] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[9] Néstor Cataño and Marieke Huisman. Chase: a static checker for JML's *assignable* clause.

[10] 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, to appear.*, 2004.

[11] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.

[12] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[13] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.

[14] Edmund Clarke, Daniel Kroening, Ofer Strichman, and Joel Ouaknine. Completeness and complexity of bounded model checking. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96, 2004.

[15] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, November 2001.

[16] Bruno Dutertre. The Priority Ceiling Protocol: Formalization and analysis using PVS. Technical report, SRI International, Menlo Park, CA 94025, 1999.

[17] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[18] Jean-Christophe Filliâtre and Claude Marché. Multi-prover Verification of C Programs. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.

[19] C. Flanagan, S. Qadeer, and S. Seshia. A modular checker for multithreaded programs, 2002.

[20] OSEK Group. OSEK/VDX Operating System Specification 2.2.3, http://www.osek-vdx.org, February 2005.

[21] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[22] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[23] C. B. Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *Proceedings Information Processing 83 (IFIP 9th world congress)*, pages 321–332. Elsevier Science Publishers B. V. (North-Holland), 1983.

[24] C. B. Jones. Wanted: a compositional approach to concurrency. pages 5–15, 2003.

[25] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[26] Leslie Lamport. What good is temporal logic? In R.E.A. Mason, editor, *Proceedings Information Processing 83 (IFIP 9th world congress)*, pages 657–668. Elsevier Science Publishers B. V. (North-Holland), September 1983.

[27] G. Leavens and Y. Cheon. Design by contract with jml, 2003.

[28] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[29] Joseph Lemieux. *Programming in the OSEK/VDX Environment.* C M P Books, 2001.

[30] Bertrand Meyer. *Object Oriented Software Construction (2nd edition).* Prentice Hall International, 2 edition, May 1997.

[31] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Softw. Eng.*, 7(4):417–426, 1981.

[32] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.

[33] M. Pilling, A. Burns, and K. Raymond. Formal specifications and proofs of inheritance protocols for real-time scheduling. *IEE/BCS Software Engineering Journal*, 5(5):263–279, September 1990.

[34] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE, IEEE Computer Society Press.

[35] Hendrik Post and Wolfgang Küchlin. Integrated static analysis for linux device driver verification. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods (IFM), 6th International Conference, Oxford, UK, July 2-5, 2007, Proceedings*, volume 4591 of *Lecture Notes in Computer Science*, pages 518–537. Springer, 2007.

[36] Shaz Qadeer and Dinghao Wu. Kiss: Keep it simple and sequential.

[37] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2005.

[38] John Regehr and Nathan Cooprider. Interrupt verification via thread verification. *Electron. Notes Theor. Comput. Sci.*, 174(9):139–150, 2007.

[39] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *LNAI*, pages 398–414. Springer, 2005.

[40] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

[41] Babita Sharma, S. D. Dhodapkar, and S. Ramesh. Assertion checking environment (ace) for formal verification of c programs. In *SAFECOMP '02: Proceedings of the 21st International Conference on Computer Safety, Reliability and Security*, pages 284–295, London, UK, 2002. Springer-Verlag.

[42] Vector Informatik GmbH, Ingersheimer Str. 24, 70499 Stuttgart, Germany. *OSEK/VDX Workshop*, October 2006.

[43] Libor Waszniowski and Zdeněk Hanzálek. Analysis of OSEK/VDX based automotive applications. In *IFAC Symposium on Advances in Automotive Control, Salerno*. Elsevier, April 2004.

[44] Qiwen Xu, Willem P. de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.