View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Digitale Bibliothek Thüringen

PROCEEDINGS

55. IWK

Internationales Wissenschaftliches Kolloquium

International Scientific Colloquium

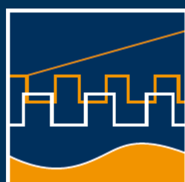
13 - 17 September 2010

Crossing Borders within the ABC

Automation,

Biomedical Engineering and

Computer Science



Faculty of
Computer Science and Automation

www.tu-ilmenau.de

TECHNISCHE UNIVERSITÄT
ILMENAU

Home / Index:
http://www.db-thueringen.de/servlets/DocumentServlet?id=16739

Impressum
Published by

Publisher:	Rector of the Ilmenau University of Technology Univ.-Prof. Dr. rer. nat. habil. Dr. h. c. Prof. h. c. Peter Scharff
Editor:	Marketing Department (Phone: +49 3677 69-2520) Andrea Schneider (conferences@tu-ilmenau.de) Faculty of Computer Science and Automation (Phone: +49 3677 69-2860) Univ.-Prof. Dr.-Ing. habil. Jens Haueisen
Editorial Deadline:	20. August 2010
Implementation:	Ilmenau University of Technology Felix Böckelmann Philipp Schmidt

USB-Flash-Version.

Publishing House:	Verlag ISLE, Betriebsstätte des ISLE e.V. Werner-von-Siemens-Str. 16 98693 Ilmenau
Production:	CDA Datenträger Albrechts GmbH, 98529 Suhl/Albrechts
Order trough:	Marketing Department (+49 3677 69-2520) Andrea Schneider (conferences@tu-ilmenau.de)

ISBN: 978-3-938843-53-6 (USB-Flash Version)

Online-Version:

Publisher:	Universitätsbibliothek Ilmenau ilmedia Postfach 10 05 65 98684 Ilmenau

© Ilmenau University of Technology (Thür.) 2010

The content of the USB-Flash and online-documents are copyright protected by law.
Der Inhalt des USB-Flash und die Online-Dokumente sind urheberrechtlich geschützt.

Home / Index:
http://www.db-thueringen.de/servlets/DocumentServlet?id=16739

CASE STUDY: VERIFICATION OF A REAL TIME OPERATING SYSTEM

Alexander Pacholik, Wolfgang Fengler

TU Ilmenau,
Computer Architecture and
Embedded Systems Group
Helmholtzplatz 1,
98693 Ilmenau, Germany

Tobias Simon

TU Ilmenau,
Integrated Communication
Systems Group
Gustav-Kirchhoff-Str. 1,
98693 Ilmenau, Germany

ABSTRACT

Verification of complex distributed systems is a challenging task. There is a number of approaches, based on the correctness by design concept, involving code generation. However such approaches are not always feasible. On the other side embedded systems incorporate distributed multitasking systems are hard to be verified by code analysis alone. In our approach we combine automated code level analysis with high level modeling of selected hardware mechanisms. The fitness of the approach is demonstrated by checking a number of critical properties in a custom embedded real time operating system (eRTOS) for a high performance 32-Bit digital signal processor.

1. INTRODUCTION

Embedded systems are continuously growing in performance, which often adds complexity and the demand to integrate more features and functionality. Even small systems use multitasking systems or networking capabilities. This results in a growing complexity and a growing uncertainty during development. A well defined development cycle and model based development can help to cope with this complexity, however in some situation such an approach is not feasible, e.g. if an existing system needs to be extended. Here approaches are necessary, which aid the developer to detect faults, or at least suggest on problematic parts.

In a recent project we developed a custom real time operating system with basic features according to multitasking, driver management and so on. Due to the growing demand on computing power the system was extended to a multiprocessor platform. The growing complexity led to spontaneous faults, which were hard to detect.

This led to the idea to make a case study about the fitness of existing verification methods. Model checking methods[1] can be applied automatically, if the required properties are well known. However, there are two main problems: How to transform existing code into a model checking problem? How to recognize the concurrency in terms of tasks and interrupt functions used in the embedded system? Thus we decided to close the gap with an appropriate method.

Some general restrictions apply to statical analysis of software (source code) in terms of automatic, formal verification, due to combinatorial growing of the state space. The state space of software grows exponential to program size and number of tasks, see section 2.7. Thus we need to restrict on states required for describing interaction between tasks, interrupts and operating system. This leads to a property preserving abstraction of the analyzed system.

In section 2 the prerequisites for our approach and the used models are described. Section 3 describes implementation issues and experimental results. In the final section 4 we give an outlook to future work.

2. MODELING

This section describes the process for deriving implementation models from implementation source code. First some prerequisites are stated, on which the work relies. Thereafter the used models, and algorithms are described.

2.1. Prerequisites

The modeling concepts are based on some prerequisites, in terms of coding guidelines, relating the source code.

1. Functions must not be recursively implemented.
2. Pointers to Functions (Callback Functions) are not allowed, because data dependencies cannot be (easily) resolved by static code analysis.
3. Data structures for task synchronization (Mutex) must be global, to be available or static code analysis.
4. Assembler functions should contain exactly one *return* statement, for easier parsing of source code.
5. All tasks must be a priory known and should reside during runtime.

Prerequisite 1 is a general coding guideline (at least for embedded systems) to avoid stack overflows. Pointers to functions are often used to define entry points for tasks and should be avoided elsewhere. Thus prerequisite 2 is also covered by coding guidelines. Task entry functions are accepted because they are known (prerequisite 5). The eRTOS used for the case study provides special functions for mutexes. Thus mutex accesses can be modeled separately (prerequisite 3). Prerequisite 4 has been used during case study to simplify the parser grammar. Violations can be easily detected and an extension of the grammar is also possible. Prerequisite 5 states that tasks should be known and constantly reside in memory. For most embedded systems this is a general concept, even if some tasks are rarely or never used.

2.2. Analysis Process

Figure 1 depicts the steps of the analysis process. First the assembler source code is parsed and results in an abstract syntax tree (AST). Compared to C-code the assembler code also exhibits compiler impact, for example optimizations. From the AST a call graph is generated, which exploits dependencies between functions. Thereby possible recursions and unreachable functions can be efficiently identified. Detailed information about dependencies is provided by a control flow graph, which is extracted from the AST. The control flow graph subdivides the program into blocks with different reachability. A colored Petri net is used to exploit interdependencies between several tasks, which are not directly available from the control flow graph. The Petri net is build by connecting the control flow graphs of several functions. The resulting colored Petri net is isomorphic to the control flow of original source code. Now safety and fairness properties can be checked by applying model checking methods on the state space of the colored Petri net.

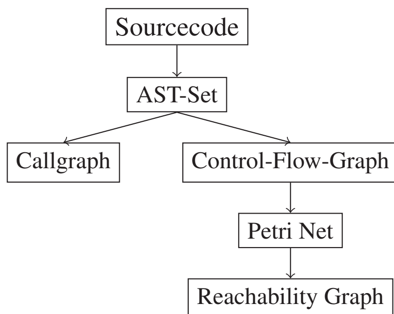


Fig. 1. Overview: Analysis chain

2.3. Parsing and AST

A parser generated from an EBNF grammar is used to process assembler source code and generate an abstract syntax tree. We used reversed engineering to define a reduced grammar, which contains reasonable a set of 16 rules, sufficient to extract the needed information and distinguish comments, control flow specific instructions and instructions without affecting the control flow.

2.4. Call Graph

A call graph is a directed graph in which each node represents a function f, and each directed arc from f_1 to f_2 describes f_1 is possibly calling f_2. The recursion prerequisite, see section 2.1, is met, if the graph is acyclic, which can be decided by depth first search.

2.5. Control Flow Graph

The call graph contains only coarse information about the original program, information about the order or mutually exclusiveness of function calls is omitted. The control flow graph is created, by extending the call graph from section 2.4 with conditional and unconditional jump instructions. Predicates of conditional jumps ore omitted. Optionally suitable techniques, such as data flow analysis may be used [2].

2.5.1. Definition Control Flow Graph

A control flow graph is defined as 10-Tuple:

$$G_C(B, B_S, B_R, C, J, J_C, P, A, f_{CA}, B_{IS}, B_{TS})$$

- B is the set of blocks, whereby a block represents a set of instructions with equal reachability.
- $B_S \subseteq B$ is the set of entry blocks of functions.
- $B_R \subseteq B$ is the set of return blocks of functions.
- $C \subseteq B \times B_S$ is the set of function calls.
- $J \subseteq B \times B$ is the set of jumps, whereas $C \cap J = \emptyset$
- $J_C \subseteq J$ is the set of conditional jumps.
- A is the set of function arguments, necessary to model dependencies, like mutual exclusiveness.
- f_{CA} is a surjective mapping from function calls C to arguments A; $f_{CA} : C \to A$.
- $B_{IS} \subseteq B_S$ is the set of starting blocks of used interrupt functions.
- $B_{TS} \subseteq B_S$ is the set of starting blocks of used task functions.

Some information can be derived from this definition. $B_S \cap B_R = \emptyset$ and $B_I = B - (B_S \cup B_R)$ is the set of inner blocks. The set E of edges results from $E = C \cup J$. And the set J_U of unconditional jumps can be results from $J_U = J - J_C$.

2.5.2. Model Validity

The underlying model requires that for each function exists exactly one starting block and one return block,

such that each function eventually returns.[1] This requirement can be stated as integrity of the set E, which is defined as follows, by valid combinations of block types and edge types. The first three combinations apply to return blocks, which must not contain unconditional jumps, while function calls and one conditional jump are allowed, see equation 1.

$$\forall b_1 \in B_R: \quad \begin{array}{l} |\{b_2|(b_1, b_2) \in E - J_U\}| \leq 1 \wedge \\ |\{b_2|(b_1, b_2) \in J_U\}| = 0 \end{array} \quad (1)$$

The other three valid combinations apply to non return blocks $B_{NR} = B - B_R$. A non return block may either be left by exactly one unconditional jump and optionally one unconditional jump or function call, see equation 2. A valid model satisfies both conditions.

$$\forall b_1 \in B_{NR}: \quad \begin{array}{l} |\{b_2|(b_1, b_2) \in J_U\}| = 1 \wedge \\ |\{b_2|(b_1, b_2) \in J_C \cup C\}| \leq 1 \end{array} \quad (2)$$

2.5.3. *Initial Activities*

Initial activities start from distinct blocks, and can be divided into interrupts B_{IS} and tasks B_{TS}, whereas $B_{IS} \cap B_{TS} = \emptyset$. This is necessary, because interrupts and tasks are conceptually different in this approach.

2.5.4. *Reduction*

Using information from the call graph and its induction, unreachable functions can be removed. This is possible, because function pointers are not allowed, see prerequisite 2 in section 2.1.

Another reduction can be applied according to critical functions. Critical functions reside in the operating system core and are a priori defined. All functions directly or indirectly calling critical functions are considered as *possibly critical*, all other functions are *uncritical*. Thus, if only behavior regarding critical functions is concerned, all *uncritical* functions can be removed.

2.6. Petri Net

To make statements about the dynamics of the embedded system, we need a standardized representation, which includes the concurrent behavior. Petri nets allow a graphical representation as well as formal analysis of concurrent tasks. A colored Petri net allows a conjoint modeling of the system structure and concurrent activities. Otherwise (parts of) the structure need to be replicated for each task, which can be achieved by deconvolving the colored net.

The universal definition from [3] is very comprehensive and thus not suitable for our purposes. We analyzed typical scenarios and developed a reduced definition, using the following simplifications:

[1]For functions, which are not called, e.g. the main function, the return block must not be reachable.

- The number of colors equals the number of tasks plus a neutral color for task interaction.
- The capacity of places is either *zero* or *one* for each color. Each place can contain either neutral color tokens (capacity is *zero* for other colors) or other color tokens (capacity is zero for the neutral color), but not both. No total capacity is defined.
- Arcs don't contain extra information. The cardinality is always *one*. Test arcs are permitted.

2.6.1. *Definition Colored Petri Net*

The resulting definition differs from generalized colored Petri nets. A colored Petri net is a bipartite graph which can be defined as 8-tuple:

$$PN(P, T, A, P_N, A_T, C, c_N, m_0)$$

- P is the set of places.
- T is the set of transitions.
- A is the set of arcs, $A \subseteq (P \times T) \cup (T \times P)$.
- $P_N \subseteq P$ is the set of places holding only the neutral color. The set of colored places is $P_C = P - P_N$.
- C is the set of colors, which can be contained in places.
- $c_N \in C$ is the neutral color. The set $C - \{c_N\}$ of colors is available for tasks and interrupts.
- m_0 is the initial marking of the net. It defines the initially available tokens.

2.6.2. *Coloring Function*

In colored Petri nets a coloring function describes the consumption and creation of tokens with distinct colors for each transition. For our Petri net class this coloring function is implicitly defined, because tasks behave equally and interact only by designated mechanisms. The capacities for places in P_C are one for each color but capacity for the neutral color is zero, and vice versa for places in P_N. A transition requires either to be connected to colored places by at least one pre- and post-arc, or to neutral places only. During firing of a transition exactly tokens of one color (plus the neutral color) are consumed and produced. Thus the coloring function is properly defined.

2.6.3. *Construction of the Petri Net*

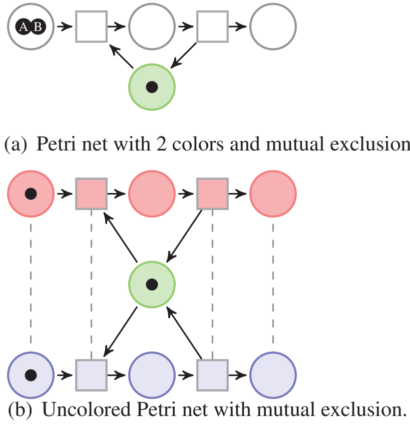
The Petri net is constructed from the control flow graph by applying the following rules:

1. Blocks are transformed into places: $B(G_C) \Rightarrow P(PN)$
2. Jumps are transformed to transitions: $J(G_C) \Rightarrow T(PN)$.

3. Function calls are transformed into one waiting place for the return and two transitions, one to the starting block and one from the returning block of the called function.
4. Designated functions, which are used for interaction, are described in the following.

2.6.4. *Mutual Exclusion*

A mutual exclusion (*mutex*) is a synchronization, which ensures that a set sequence of instructions does not infer with other tasks. Therefore symmetric functions „`lock`“ and „`unlock`“ are used. The „lock“ function disables all interrupts and consumes a special token. The `unlock` function returns this token and enables interrupts. To prevent a deadlock, the functions „`lock`“ and „`unlock`“ must always be used in pairs. Figure 2(a) shows a mutex place in a colored Petri net and its deconvolved (uncolored) Petri net in figure 2(b), without interrupt enable/disable.



(a) Petri net with 2 colors and mutual exclusion

(b) Uncolored Petri net with mutual exclusion.

Fig. 2. Synchronization using a mutex place

2.6.5. *Global Interrupt Enable/Disable*

An interrupt service function (ISR) intercepts execution of the currently running task triggered by an external source. This is not allowed during critical sections. Interrupts can be enabled or disable by designated functions „`enable_GIE`“ and „`disable_GIE`“. First all currently running interrupt functions need to be finished. Figure 3 shows the appropriate Petri nets: While a token is in p_{on}, all interrupts can be restarted by firing a restart transition t_{X_rst}. If a token arrives in $p_{disable_begin}$, the token is removed from p_{on} via t_{off} and a token is put into p_{off}, and no interrupt can be restarted. If all running interrupts are finished, transition $t_{disable_wait}$ fires and interrupts are disabled. Activation of interrupts starts by putting a token into p_{enable_begin}. Then, after firing t_{on}, p_{on} holds a token and interrupts are enabled. Finally t_{enable_wait} fires and the function returns.

2.7. Reachability Graph

The Petri net contains information about the dynamics of concurrently active functions, but is not sufficient for analysis. Therefore a reachability graph can be used, which contains all reachable markings of the Petri net, where a marking contains the coloring of all places in the Petri net in a certain state. The reachability graph can be constructed from the Petri net beginning with its initial marking m_0.

2.7.1. *Definition*

A reachability graph is a 2-Tuple $R(M, S)$ whereby:

- M is the set of markings, which represents the vertices of the graph.
- S is the set of directed arcs connecting the vertices, with $S \subseteq M \times M$.

The most important property is deadlock-freedom. A deadlock means, the Petri net contains a marking with no progress[2].

$$\exists_{m_D \in M} \left(\bigvee_{m \in M - \{m_D\}} (m_D, m) \notin S \right)$$

A deadlock marking either contains no outgoing arc, or only a marking back to the deadlock marking (self loop).

The reachability graph is constructed by adding all reachable markings beginning with m_0, and their directly reachable markings by firing a transition.

2.7.2. *Complexity of the Reachability Graph*

The state complexity of the reachability graph (number of markings) of a colored Petri net is EXPSPACE[4]. Given the number of places P and colors C the maximum number of reachable markings is

$$|M| \leq 2^{|P|*|C|}.$$

We can give further approximations based on the information gathered during the analysis. So we can restrict a color to the places P_i, which are reachable from the associated task (or ISR). The maximum number of markings then results in

$$|M| \leq 2^{\sum_{i=1}^{|C|}(P_i)}.$$

In our case each function is modeled by the places P_f, in which only one place can be marked per color. Thus each function adds at most $|P_f| + 1^{|C|}$ markings. This results in

$$|M| \leq \prod_{i=1}^{|C|} (\prod_{j=1}^{m} (|P_{f_{ij}}| + 1))$$

[2] In contrast to the co-trap definition for petri nets.

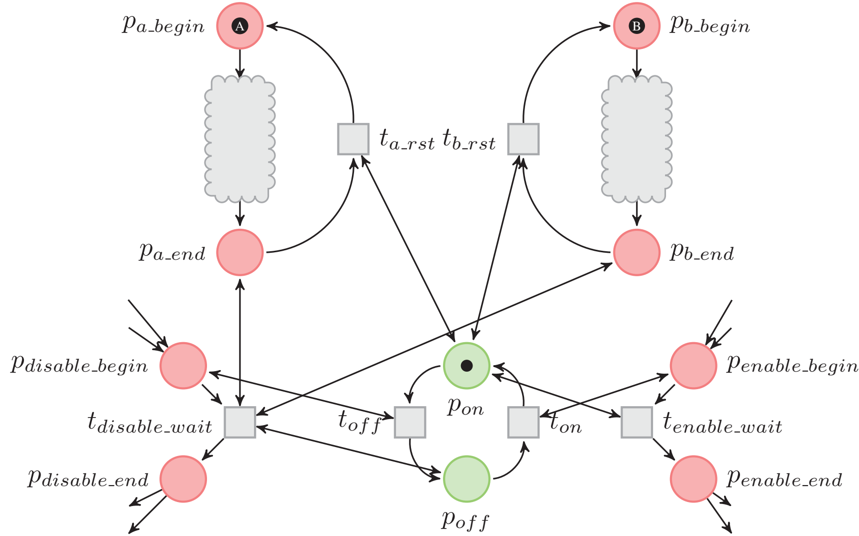


Fig. 3. Global activation/ deactivation of two interrupts.

markings, where $P_{f_{ij}}$ is the number of places of the j-th function reachable from task i.

The whole method is sensitive to the number of model elements (places) necessary to model a function, to the number of functions and the number of running tasks. Therefore reduction techniques need to be applied, which reduce the structure, without affecting analyzed properties, such as deadlock detection, see section 2.5.4. Furthermore the structure of the Petri net can be simplified by merging places with equivalent reachability properties, according to standard rules, see [5].

3. CASE STUDY

This section explains implementation issues and obtained experimental results.

3.1. Implementation

We implemented the models given in section 2 using Java^{TM}. The Parser was created using the parser generator JavaCC. The other models where created using the definitions given in section 2. Erroneous markings can be interpreted in terms of tasks and functions. Each state in the reachability graph represents the involved tasks to execute distinct code sequences. As expected, the most critical part was the creation of the reachability graph, more precisely the realization of markings, because of its size. So we analyzed several alternative, preferably small data structures for implementation. We used integrated data structures with *Hash* and *Tree* based implementation, further called *HashMarking* and *TreeMarking*. We also used custom data structures based on text-encoding *PackedMarking* and bit-encoding *EnumMarking*.

3.2. Experimental Results

Subject of the case study analysis is a custom real time operating system developed by our department for a TMS320C6000 digital signal processor [6]. A sample application used 4 tasks and 4 interrupt functions. With our implementation, we can easily analyze the assembler source code obtained from the compiler regarding to deadlock scenarios, after configuring running tasks and used ISRs.

Altogether we detected 201 functions, of which 73 were not reachable, due to unused device drivers and deactivated tasks. Another 100 functions could automatically be removed, because they did not interact with critical functions, e.g. functions for hardware initializations, which run only during booting. After reduction the control flow graph contains 28 functions, which are relevant for analysis. This is about $7,1\%$ of the original functions. The generated Petri net has 317 places, 353 transitions and 829 arcs. After reducing the Petri net 136 places, 141 transitions and 399 arcs where left. Without reductions using control flow graph and Petri net, a large Petri net with 1921 places, 2100 transitions and 5325 arcs results, which currently would not be able to be analyzed. So the optimized Petri net (regarding to places) is 7% of the unoptimized net.

Table 1 shows the dependencies of the Petri net size on the number of tasks and reductions. In the unoptimized Petri net only unreachable functions were removed. The size of the Petri net strongly depends on the complexity of tasks.

As mentioned in section 2.7.2, the number of tasks has a major influence on the size of the reachability graph, because for each task a color is added, which is equivalent to replicate a part of the original Petri net. Table 2 shows the dependency of the size of the reachability graph on the number of tasks. A number of dead-

	Unoptimized Petri Net			Optimized/Reduced Petri Net		
Tasks	Places	Transitions	Arcs	Places	Transitions	Arcs
1	77	72	174	36	28	86
2	252	269	641	104	101	293
3	333	371	871	135	140	395
4	334	372	875	136	141	399

Table 1. Petri net complexity depending on task number and reduction

Tasks	Markings	Edges	Duration
1	41	45	<1 sec
2	2733	6397	1 sec
3	292424	1.050.033	70 sec
4	292.424	1.167.105	95 sec

Table 2. Complexity of the reachability graph depending on number of tasks

lock scenarios where detected, which were „false positives“. The reason was an interrupt deactivation inside an interrupt. Interrupt deactivation inside an interrupt is forbidden, because it waits on its own completion, see section 2.6.5. These scenarios are further removed in the source code, because of bad coding style.

Some experiments have been made with erroneous mutex accesses and previous versions of the eRTOS. These scenarios were correctly recognized during analysis.

Some experiments were made to determine the influence of the implementation of markings on memory usage and runtime, see section 3.1. We used an example reachability graph with 140000 markings. The results are depicted in table 3. It is obvious, that *HashMarking* needs most runtime and memory, followed by *TreeMarking*. Both are outperformed by *PackedMarking*, and *EnumMarking*, which are optimized for fast comparison. While *PackedMarking* needs slightly less memory, *EnumMarking* is faster.

140000	Memory	Time
HashMarking	1477 MB	185s
TreeMarking	1060 MB	107s
EnumMarking	167 MB	44s
PackedMarking	154 MB	85s

Table 3. Memory consumption and runtime depending on marking

4. OUTLOOK AND FUTURE WORK

We presented an approach for automatically verifying deadlock properties in a multitasking operating system for embedded systems. The main advantage is its simplicity while connecting small model templates with automatically derived control flow models. So it is easy to reinterpret analysis results in terms of the original model. The general fitness of the approach has been demonstrated using sample scenarios. So far the verification back-end has been created straight forward, further optimizations, e.g. using compression and BDD-based verification methods are still possible. In future work a more powerful verification tool could be used, which supports model checking and thus allows checking more complex properties, such as fairness.

The general approach may be extended in two ways. First, distributed systems may be supported by using a template for communication, e.g. for connecting send- and receive functions of different processors. Another extension can be made toward real time behavior. When approximating runtime of code sequences, the analysis model may be easily extended toward timed Petri nets.

5. ACKNOWLEDGEMENTS

This work has been supported by German Research Foundation under grant SFB622.

6. REFERENCES

[1] Gerard J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley Professional, September 2003.

[2] F. Nielson, H. Riis Nielson, and C. L. Hankin, *Principles of Program Analysis*, Springer, 1999.

[3] Kurt Jensen, *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, Springer, 1997, Three Volumes.

[4] R. J. Lipton, “The reachability problem requires exponential space.,” Tech. Rep., Yale University, Department of Computer Science, Jan, 1976.

[5] R. Davis and H. Alla, *Discrete, Continous, and Hybrid Petri Nets*, Springer-Verlag Berlin Heidelberg, 2005.

[6] Bernd Dne and Wolfgang Fengler, “Modeling and simulation of operating system behavior,” *MSO 2003. The IASTED International Conference on Modelling, Simulation and Optimization. July2-4 2003, Banff, Canada*, pp. 78–81, 2003.