# Study on Model Checking Techniques for OSEK/VDX Applications

ZHANG, Haitao

Japan Advanced Institute of Science and Technology

Doctoral Dissertation

# Study on Model Checking Techniques for OSEK/VDX Applications

ZHANG, Haitao

Supervisor: Toshiaki Aoki

School of Information Science
Japan Advanced Institute of Science and Technology

September, 2015

# Abstract

OSEK/VDX, a standard of automobile OS promulgated by the alliance of German and French automotive manufacturers in 1994, has been widely adopted by many automobile manufacturers to design and implement a vehicle-mounted OS, such as BMW, Opel and TOYOTA. Based on the OSEK/VDX OS, more and more applications are developed and deployed in vehicles to assist drivers to control vehicles, such as the cruise control system and temperature control system. However, with the growing functionalities in vehicles and increasing complexity in the development, how to straitly ensure the reliability of the developed OSEK/VDX applications is becoming a challenge for developers.

An application developed on OSEK/VDX OS consists of multiple tasks. To concurrently executing multiple tasks on one CPU, a scheduler within OSEK/VDX OS is used to dispatch tasks. Moreover, OSEK/VDX OS provides a standardized application interfaces (APIs) for its applications, and tasks within the applications can invoke the provided APIs to dynamically change the scheduling of tasks. To ensure the reliability of OSEK/VDX applications, model checking, which is an efficient and exhaustive verification technique for concurrent software, can be applied to verify OSEK/VDX applications for detecting subtle and logic errors.

There exist many model checking methods that have been applied to verify concurrent software such as the ANSI-C multi-threaded software and SystemC multi-threaded software. In the existing methods, due to the concurrency of target software, all of the interleavings of threads are checked in the verification process in order to exhaustively verify the given software. Although the existing methods can exhaustively check concurrent software, these methods usually perform an approximate verification since the behaviours of scheduler are not taken into account in verification process. If we apply these existing model checking methods to verify OSEK/VDX applications, it is too imprecise since a lot of unnecessary interleavings of tasks will be superfluously checked in the verification stage, especially these unnecessary interleavings will usually result in a spurious bug. Furthermore, as a result of the spurious bug, developers have to spend extra costs judging whether the detected bug is real one or not after completing verification. As to reduce the checking costs, a more accurate model checking approach should be used in the verification of OSEK/VDX applications.

Recently, to accurately check concurrent software such as ANSI-C multi-threaded software and SystemC multi-threaded software, several prominent model checking methods have been proposed by some senior researchers. In these methods, as to seek a more accurate verification result, the behaviours of scheduler are taken into account in the verification process. Even so, these prominent model checking methods are still unsuitable to accurately verify OSEK/VDX applications because of different scheduling policy. In the existing methods, the running thread within the concurrent software is arbitrarily determined by scheduler (non-deterministic scheduler is used to dispatch threads). However, compared with the non-deterministic scheduler, in OSEK/VDX applications which task is to be run is explicitly determined by OSEK/VDX scheduler — a deterministic scheduler is used to dispatch tasks.

The purpose of this thesis is to make model checking technique more accurate in the verification of OSEK/VDX applications. In order to achieve this purpose, in this thesis we describe and develop three approaches that can accurately and automatically verify the safety property of OSEK/VDX applications using model checking technique. To the best of our knowledge, our work is first to apply model checking technique to accurately verify the multi-tasks/threads software which is dispatched by a deterministic scheduler.

To accurately verify OSEK/VDX applications using model checking technique, the behaviours of OSEK/VDX OS such as scheduler behaviours should be taken into account in the checking process since the running task is explicitly determined

by OSEK/VDX scheduler and the APIs invoked from tasks will haphazardly change the scheduling of tasks. In our first approach, we investigate a checking method based on the existing model checker Spin. In the approach, as to employ Spin model checker to accurately verify OSEK/VDX applications, we propose a checking model which is a combination model of application model and OSEK/VDX OS model to precisely simulate the executions of the OSEK/VDX applications in real OSEK/VDX OS. Compared with our first approach (Spin-based checking approach), in our second approach we develop a new technique named execution path generator (EPG) to verify the OSEK/VDX applications based on the SMT-based bounded model checking. In the approach, as to avoid the behaviours of OSEK/VDX OS to be explored in the verification stage, the OSEK/VDX OS model is embedded in EPG (constructing model algorithm flat) to dispatch tasks and respond to the APIs invoked from tasks in the process of constructing checking model of application. Furthermore, based on the sequentialization idea, in our last approach we present a novel method to translate OSEK/VDX applications into sequential models in order to efficiently apply existing model checkers such as Spin and cbmc to directly check OSEK/VDX applications. In particular, as to avoid the behaviours of OSEK/VDX OS to be poured into the sequential models, like our second approach (EPG technique), in the approach the OSEK/VDX OS model is embedded in translation algorithm flat to dispatch tasks and respond to the APIs invoked from tasks in the sequentialization process.

We have implemented our approaches and conducted two types of experiments to evaluate the proposed approaches. In the first experiments, as to show the accuracy of our approaches, the sequentialization-based checking approach is selected as candidate from our checking approaches, and we compared the checking approach with the existing model checking methods for concurrent software. The experiment results denote that our approach is an accurate checking method for OSEK/VDX applications in contrast with the existing model checking methods for concurrent software. In the second experiments, we evaluated the efficiency and scalability of our approaches based on a series of experiments. According to the conducted experiment results, we find the following results,

- The Spin-based checking approach can accurately verify the OSEK/VDX applications, but the scalability of this approach is limited because too many details about OSEK/VDX OS model are explored in the verification stage, especially the state space explosion will happen if the checked applications hold a lot of tasks and APIs.
- EPG technique is more scalable than Spin-based checking approach in checking the applications which hold a lot of tasks and APIs. However, it is not efficient to check the applications which hold a large number of branches, since the checking model of application is constructed based on the execution paths in EPG technique. If an application hold a lot of branches, it will spend a lot of time exploring execution paths in the verification process, which will slow down the performance of EPG technique.
- Based on the sequentialization process of our approach, the selected model checker Spin can efficiently verify the applications which hold a lot of tasks, branches and APIs with the less cost in terms of states, time and memory compared with the Spin-based checking approach and EPG technique.

Furthermore, we have used our sequentialization-based approach to sequentialize many experimental OSEK/VDX applications which hold about 1000 lines of C code, and verified the sequentialized applications with the well-known bounded model checker cbmc. The performances indicate that the sequentialization-based approach and cbmc can be considered as a practical method to verify the OSEK/VDX applications with industrial complexity.

**keywords:**

OSEK/VDX applications, deterministic scheduler, model checking, Spin, bounded model checking, SMT, sequentialization

# Acknowledgments

I would like to first express my greatest gratitude to my principal advisor, Associate Professor Dr. Toshiaki Aoki, who encouraged and advised the idea from the beginning to the end of my dissertation. I am delighted and grateful to be able to work under his excellent supervision. He provided me a lot of instructive advices, useful suggestions, insightful criticism and professional guidance, without his consistent and illuminating instruction, this thesis could not be presented in its current form.

In addition, I should give my hearty thanks to Assistant professor Yuki Chiba. He has put considerable time and efforts in my research. He gave me a lot of invaluable comments to overcome my research problems, from whom I benefited a lot of knowledges. Also, I would like to thank my friends, Lin Wang, Zhuo Cheng and Shengbei Wang. They kindly gave me a hand when I was in frustration or depression. Their encouragement and unwavering support has sustained me through the hard times. Last but not the least, my thanks would go to my parents for their infinite love and great confidence for me all through these years. They have supported me continuously, which is the biggest motivity for my study.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

OSEK/VDX [34][43] is a standard of automobile OS, developed by a consortium of European automobile manufacturers and suppliers in conjunction with the University of Karlsruhe in 1994. The primary motivation of OSEK/VDX standard is to resolve the problem of increasing software content in vehicles and desire high-quality products. Currently, it has been widely adopted by many automobile manufacturers to design and develop a vehicle-mounted OS, such as BMW, Opel and TOYOTA. Based on the OSEK/VDX OS, more and more applications are developed and deployed in vehicles to assist drivers to control vehicles, such as the cruise control system and temperature control system. To an application which runs on the OSEK/VDX OS, the reliability is very important, since a nonreliance application will threaten our life. However, with the growing functionalities in vehicles and increasing complexity in the development, how to straitly ensure the reliability of the developed OSEK/VDX applications is becoming a challenge for developers.

## 1.1 OSEK/VDX Applications and Model Checking

An application developed on OSEK/VDX OS consists of multiple tasks (OSEK/VDX application is multi-tasks software). To concurrently execute multiple tasks on one CPU, the *static priority scheduling policy* with non-preemptive strategy and full-preemptive strategy is adopted by OSEK/VDX scheduler to dispatch tasks. Particularly, as shown in Fig. 1.1,

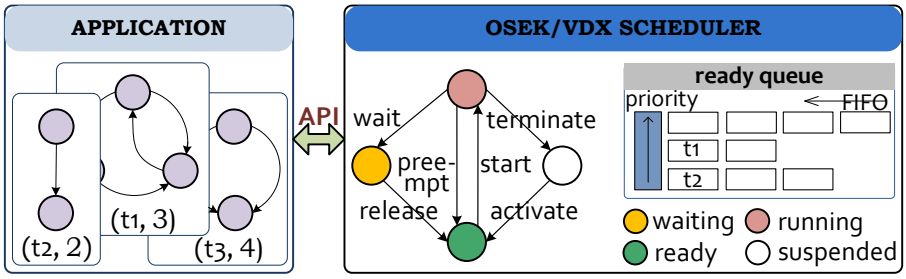



Figure 1.1: OSEK/VDX scheduler

a ready queue is used to lay out the execution order of tasks. Based on the ready queue, the running task is explicitly determined by scheduler (deterministic scheduler is used to dispatch tasks). OSEK/VDX OS can process two types of tasks, basic task and extended task. The states of a basic task consist of *running* state, *ready* state and *suspended* state. Compared with the basic task, the extended task can hold synchronization events and has a unique state called *waiting* state. Furthermore, OSEK/VDX OS provides a standardized application interfaces (APIs) for its applications (such as $ActivateTask(tk)$, $TerminateTask()$, $WaitEvent(eid)$ and $GetResource(rid)$), and tasks within the application can invokes these APIs to dynamically change the states of tasks, synchronization events and shared resources, e.g., invoke the API $ActivateTask(tk)$ to activate a suspended task.

To ensure the reliability of OSEK/VDX applications, testing as an important checking technique is being used in industry. However, there are several disadvantages when we apply testing technique to check the developed OSEK/VDX applications, e.g., as we know, it is difficult to exhaustively check an OSEK/VDX application using testing technique, especially when the tested OSEK/VDX application holds non-deterministic behaviours, because of the incomplete test-cases. In addition, it is hard to reproduce an error trace when we detect the error using testing technique, due to the concurrency of OSEK/VDX applications.

Model checking [2][3], an efficient and exhaustive verification technique for concurrent software, can be applied to verify OSEK/VDX applications instead of testing technique. In contrast with testing technique, there are several advantages in model checking technique,

e.g., it is usually an exhaustive and automatic checking technique. Particularly, it will return a counterexample when detecting an error. Based on the returned counterexample, developers can easily find the error trace and reproduce the erroneous behaviours.

## 1.2 Shortcomings of Existing Model Checking for OSEK/VDX Applications

There exist many model checking methods [26][50][51] that have been applied to verify concurrent software such as the ANSI-C multi-threaded software and SystemC multi-threaded software. In the existing methods, due to the concurrency of target software, all of the interleavings of threads are checked in the verification process in order to exhaustively verify the given software. Although the existing methods can exhaustively check concurrent software, these methods usually perform an approximate verification since the behaviours of scheduler are not taken into account in verification process. If we apply these existing model checking methods to verify OSEK/VDX applications, it is too imprecise since a lot of unnecessary interleavings of tasks will be superfluously checked in the verification stage, especially these unnecessary interleavings will usually result in a spurious bug. Furthermore, as a result of the spurious bug, developers have to spend extra costs judging whether the detected bug is real one or not after completing verification. As to reduce the checking costs, a more accurate model checking approach should be used in the verification of OSEK/VDX applications.

Recently, to accurately check concurrent software such as ANSI-C multi-threaded software and SystemC multi-threaded software, several prominent model checking methods [13][37] have been proposed by some senior researchers. In these methods, as to seek a more accurate verification result, the behaviours of scheduler are taken into account in the verification process. Even so, these prominent model checking methods are still unsuitable to accurately verify OSEK/VDX applications because of different scheduling policy. In the existing methods, the running thread within the concurrent software is arbitrarily

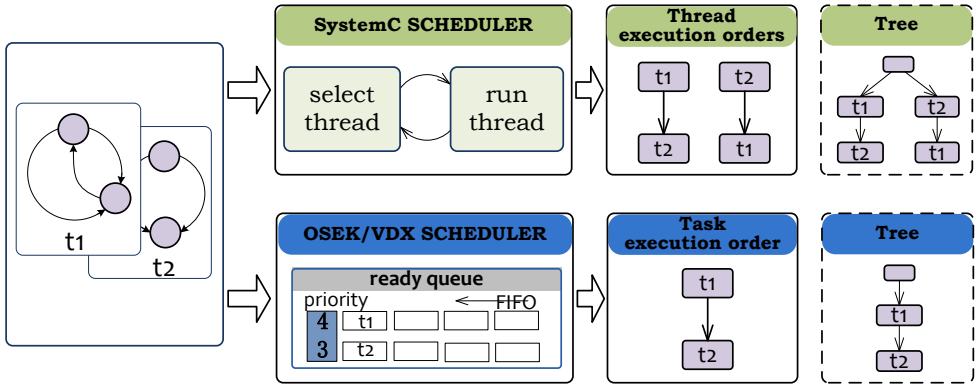



Figure 1.2: The difference of schedulers

determined by scheduler (non-deterministic scheduler is used to dispatch threads). E.g., there are two threads $t_1$ and $t_2$ in Fig. 1.2. If SystemC scheduler is used to dispatch these two threads, there are two possible thread execution orders ($t_1$, $t_2$) and ($t_2$, $t_1$). However, compared with the non-deterministic scheduler, in OSEK/VDX applications which task is to be run is explicitly determined by OSEK/VDX scheduler—a deterministic scheduler is used to dispatch tasks. E.g., for the same example shown in Fig. 1.2, if the OSEK/VDX scheduler is used to dispatch tasks $t_1$ and $t_2$, there is just one task execution order ($t_1$, $t_2$), where we assume that the priority of task $t_1$ is higher than task $t_2$ and these two tasks are currently in the *ready* state.

## 1.3 Thesis Purpose

The purpose of this thesis is to make model checking technique more accurate in the verification of OSEK/VDX applications. In order to achieve this purpose, in this thesis we describe and develop three approaches that can accurately and automatically verify the safety property of OSEK/VDX applications using model checking technique. To the best of our knowledge, our work is first to apply model checking technique to accurately verify the multi-tasks/threads software which is dispatched by a deterministic scheduler. There are two objects in this thesis, one is to show the details of our approaches, and the other is to investigate the accuracy, efficiency and scalability of proposed approaches based on

the experiments.

## 1.4 Checking Approaches for OSEK/VDX Applications

In OSEK/VDX application, the running task is explicitly determined by OSEK/VDX scheduler and the APIs invoked from tasks will haphazardly change the scheduling of tasks. There are several challenges that should be addressed when we apply model checking technique to accurately verify OSEK/VDX applications, e.g.,

- How to construct an accurate checking model for OSEK/VDX applications.
- How to handle the behaviours of OSEK/VDX OS, i.e., how to deal with the APIs invoked from tasks, and how to explicitly perform the scheduling behaviours.

In our first approach, as to apply existing model checkers to accurately verify OSEK/-VDX applications, we investigate a method based on the well-known Spin model checker [26]. To accurately verify OSEK/VDX applications using Spin, the key work is how to construct a checking model to describe the executions of an application. In our Spin-based checking approach, a combination model of application model and OSEK/VDX OS model is used to precisely simulate the executions of the OSEK/VDX applications in real OSEK/VDX OS, where all of the tasks within application are designed as `process`es, and the OS model that conforms to OSEK/VDX specification as a cooperative `process` is used to dispatch tasks and respond to the APIs invoked from tasks. We have conducted many experiments using this approach. The experiment results indicate that the Spin-based checking approach can accurately verify OSEK/VDX applications. However, the scalability of the approach is limited, because too many details about OS model are explored in the verification stage, especially the state space explosion [4] may happen if the checked application invokes a lot of service APIs.

In order to avoid the behaviors of OSEK/VDX OS model to be explored in the verification stage and intend to handle a complex OSEK/VDX application which holds a large

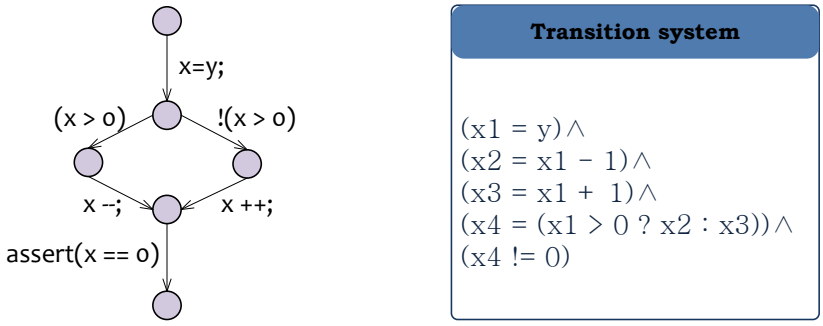



Figure 1.3: Example for bounded model checking

number of states, in our second approach we propose a new technique named execution path generator (EPG) based on the advanced SMT-based bounded model checking (BMC) [11][14]. The core work of verifying a system using BMC is how to construct a transition system (checking model) to reflect the executions of given system. As shown in Fig. 1.3, we can use the transition system to reflect the executions of the example. However, compared with the simple example shown in Fig. 1.3, an OSEK/VDX application holds multiple tasks and a scheduler is used to dispatch tasks. Moreover, the APIs invoked from tasks will haphazardly change the scheduling of tasks, e.g., the different APIs locating at different branches will lead to the different task execution orders. In contrast with the shown example, it is difficult to construct a transition system for OSEK/VDX application.

In our EPG technique, like other BMC methods for concurrent software (e.g., the method [37] for ANSI-C multi-threaded software), we explore all of the execution paths for constructing the transition system in order to successfully apply BMC technique to verify OSEK/VDX applications. Particularly, as to avoid the behaviors of OSEK/VDX OS model to be explored in the verification stage and construct an accurate transition system, the OSEK/VDX OS model is embedded in EPG (the algorithm of constructing model flat) to dispatch tasks and respond to the APIs invoked from tasks. There are two advantages in our EPG technique, e.g., (*i*) EPG: we can construct an accurate transition system without the behaviors of OSEK/VDX OS model for OSEK/VDX application based on the generated execution paths, since OSEK/VDX OS model is embedded in EPG to dispatch tasks and respond to the APIs invoked from tasks. (*ii*) SMT-based BMC: it can

make our approach more efficient and scalable in checking the complex applications which hold a large number of states, because the state-of-the-art SMT solver [10][17] is used to carry out the verification.

We have developed a corresponding tool named `osek-bmc` according to the proposed approach, and conducted many experiments using the developed tool. The experiment results show that, our EPG technique is more efficient and scalable than Spin-based checking approach in checking the OSEK/VDX applications which hold a lot of tasks and APIs, but it is not efficient to handle the applications which hold a lot of branches. This is because, in our EPG technique the transition system of an OSEK/VDX application is constructed based on the execution paths. If the application holds a lot of branches, it will spend a lot of time exploring execution paths for constructing transition system, which will slow down the performance of our EPG technique.

As to efficiently check the large-scale applications which hold a lot of tasks, APIs and branches, and moreover, enable existing model checkers such as `spin` [26] and `cbmc` [16] to directly verify OSEK/VDX applications, in our third approach we propose a novel method to translate OSEK/VDX applications into sequential models based on the sequentialization idea. In order to efficiently and accurately sequentialize OSEK/VDX applications, There are several challenges that should be addressed, e.g.,

- How to explicitly perform the scheduling behaviours of OSEK/VDX OS and deal with the invoked APIs.
- How to efficiently construct a sequential model to describe the behaviours of an OSEK/VDX application, e.g., how to efficiently handle branches and the same execution behaviours of application.

In our sequentialization-based approach, like our EPG technique, we embed OSEK/VDX OS model in the translation algorithm flat to explicitly perform the scheduling behaviours and respond to the APIs invoked from tasks. In addition, as to efficiently construct a sequential model, an extended directed graph is used to implement the sequentialization process of OSEK/VDX application. The key idea of sequentialization-based approach is

that, we symbolically execute application using the extended directed graph and explicitly perform the scheduling behaviours. In other words, in the sequentialization process task statements such as condition statements and assignment statements are not computed, instead, the extended directed graph is used to describe the executions of application under the dispatch of embedded OSEK/VDX OS model. There are several advantages in our sequentialization-based approach, e.g.,

- It can efficiently construct a sequential model for OSEK/VDX applications based on the directed graph, e.g., we can employ the combination states in directed graph to avoid the process of exploring execution paths, and use cycles to represent the same execution behaviours such as loops in application.

- The constructed sequential model does not hold the behaviours of OSEK/VDX OS model, because OSEK/VDX OS model is embedded in translation algorithm flat to dispatch tasks and respond to the invoked APIs.

- Based on the sequentialization process, the existing model checkers can be directly applied to verify OSEK/VDX applications.

We have implemented a tool named `autoC` according to the sequentialization approach, and conducted many experiments using the implemented tool. In the experiments, to comprehensively investigate the effectiveness of our sequentialization-based approach, the OSEK/VDX applications which hold different task number, API number, loop number and different scheduling behaviours are selected as our benchmarks. In addition, in the experiments, the well-known Spin is selected as the back-end model checker, and we compared our sequentialization-based approach with Spin-based checking approach and EPG technique. According to the conducted experiment results, we find the following results,

($i$) The Spin-based checking approach can accurately verify the OSEK/VDX applications, but the scalability of this approach is limited because too many details about

OSEK/VDX OS model are explored in the verification stage, especially the state space explosion will happen if the checked applications hold a lot of tasks and APIs.

(*ii*) EPG technique is more efficient and scalable than Spin-based checking approach in checking the applications which hold a lot of tasks and APIs. However, it is not efficient to check the applications which hold a large number of branches, since the transition system of application is constructed based on the execution paths in EPG technique. If the application holds a lot of branches, it will spend a lot of time exploring execution paths for constructing transition system, which will slow down the performance of our EPG technique.

(*iii*) Based on the sequentialization process of our approach, the selected back-end model checker Spin can efficiently verify the applications which hold a lot of tasks, loops and APIs with the less cost in terms of states, time and memory compared with the Spin-based checking approach and EPG technique.

Furthermore, we have used our sequentialization-based approach to sequentialize many experimental OSEK/VDX applications which hold about 1000 lines of `C` code, and verified the sequentialized applications with the well-known bounded model checker `cbmc`. The performances indicate that the sequentialization-based approach and `cbmc` can be considered as a practical method to verify the OSEK/VDX applications with industrial complexity.

## 1.5 Thesis Contribution

With the growing functionalities in vehicles and increasing complexity in the development, how to straitly ensure the reliability of the developed OSEK/VDX applications is becoming a challenge for developers. The approaches presented in this thesis can help developers to accurately check the developed OSEK/VDX applications. There are two main contributions in this thesis. The first contribution is, our approaches can reduce the

costs on the verification of OSEK/VDX applications, which benefits from the following superiorities,

1. Developers can easily use our approaches to verify an OSEK/VDX application, since our approaches support all of the OSEK/VDX OS modules and related APIs. Particularly, the corresponding tools support a subset of C programming language as input, developers can use the corresponding tools to automatically carry out the verification.

2. In the verification process, developers do not need to judge whether the detected bug is a real one or not after completing verification. This is because, the behaviours of OSEK/VDX scheduler are taken into account in our approaches. Based on the scheduling of OSEK/VDX scheduler, our approaches can perform an accurate verification.

3. Our approaches make the checking process of OSEK/VDX applications more trustworthy, since our approaches are based on the model checking that is an exhaustive checking technique.

4. Our approaches will return a counterexample when finding a bug. Based on the returned counterexample, developers can easily understand the bug and correct the bug.

The second contribution comes from our sequentialization-based checking approach. Based on the sequentialization process of our approach,

1. Developers can verify a large-scale application using existing model checkers, since the existing model checkers just verify a sequential model instead of a concurrent model.

2. Developers can conveniently select suitable model checkers to verify OSEK/VDX applications, e.g., use `cbmc` to efficiently find subtle bugs, and use `esbmc` [31] to verify the applications which hold unbounded loops.

## 1.6 Thesis Outline

The rest of the thesis is structured as follows. The background for model checking technique is presented in chapter 2. As to easily understand our checking approaches, the preliminaries for OSEK/VDX OS and applications are discussed in chapter 3. Based on the discussion about the execution characteristics of OSEK/VDX applications, the overviews of our three checking approaches are stated in chapter 4. Then, the details of our approaches are presented in chapter 5 to chapter 7. As to show the accuracy, efficiency and scalability of our approaches, the experiments are carried out in chapter 8. Related work is discussed in chapter 9. Conclusion of thesis is placed in the last chapter.

# Chapter 2

# Background

To improve the reliability of developed software, testing technique is widely used in the industry of software. However, there is an important problem that need to be solved in the verification of software, namely concurrent program verification. As we know, concurrency errors are particularly hard to find by testing technique, because the errors are difficult to reproduce. Model checking technique, an efficient and exhaustive verification technique, can solve the problem. As shown in Fig. 2.1, the key principle of model checking is: let $M$ be a state transition graph of a system (i.e., Kripke structure or finite state machine), let $f$ be a given property (i.e., assertion or temporal logic formula), explore all states $s$ of $M$ to determine whether the given property $f$ is true in the state transition graph $M$ (i.e., $M, s \models f$). If the given property $f$ is false in the state transition graph $M$, the trace in the state transition graph $M$ can be considered as a witness for revealing the given

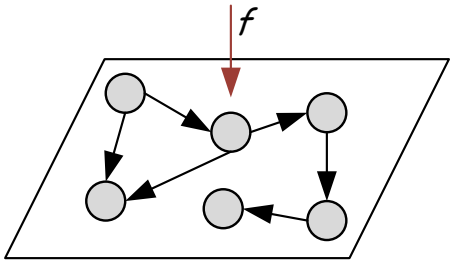



Figure 2.1: The principle of model checking

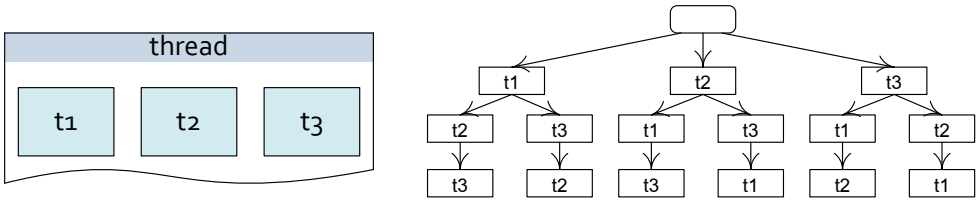



Figure 2.2: The model for a SystemC multi-threaded software

property $f$. In model checking, the trace is usually named counterexample. Based on the counterexample, the errors can be easily reproduced.

In this chapter, the model checking for concurrent software and linear temporal logic (LTL) will be firstly introduced, and then the advanced bounded model checking (BMC) technique will be discussed based on several examples.

## 2.1 Model Checking

### 2.1.1 Model Checking for Concurrent Software

Concurrent software usually consists of multiple threads. To concurrently execute multiple threads on one CPU, a scheduler is used to dispatch threads. There have been many concurrent software standards such SystemC standard, POSIX standard and ANSI-C standard that have been widely used in the industry of software. In these concurrent software standards, which thread within software is to be run is arbitrarily determined by scheduler. In other words, the running thread is stochastic when the multi-threaded software runs on the machine. If testing technique is used to check such software, it is difficult to reproduce an error track because of the non-deterministic scheduling. In contrast to testing technique, model checking can solve the problem based on the reported counterexample.

The key idea of model checking is: firstly construct a model to describe the executions of a target system, and then search the constructed model with given property. To verify concurrent software, a tree as a model is usually constructed for demonstrating all of the

possible interleavings of threads. E.g., as shown in Fig. 2.2, there is a SystemC multi-threaded software, and the multi-threaded software consists of three threads. In model checking, a tree shown in the right side of Fig. 2.2 as a model will be constructed to exhaustively describe all of the possible executions of the multi-threaded software. When the model of concurrent software is constructed, there existing many checking techniques that can be applied to search the constructed model for checking whether the constructed model satisfies the given property or not, such as explicit state checking technique [23], binary decision diagrams (BDD) technique [6][45], symbolic checking technique [21] and bounded model checking technique [24].

Currently, based on the model checking technique, there are many model checkers that have been established based on the different model checking techniques, and these established model checkers have been successfully applied to verify multi-threaded software that conforms to different concurrent software standards. E.g., Bell Labs developed a model checker named `spin` [26] based on the explicit state model checking technique. For the ANSI-C standard, a bounded model checking based model checker named `esbmc` [31] has been implemented in the university of Southampton. In addition, in the group of Cimatti, a model checker named `kratos` [12] has been established for SystemC multi-threaded software. There are also several famous model checkers, such as symbolic model checker `SMV` [7], bounded model checker `cbmc` [16], etc.

### 2.1.2 Liner Temporal Logic (LTL) Specification

In the verification of software, assertion as a common artifice is usually inserted into the model of target software to judge whether the target software models the specification. However, it is hard to use assertion to exhaustively check whole model of concurrent software, because the insertion position of assertion is confoundedly difficult to be determined, caused by concurrency. Linear Temporary Logic (LTL) is an instance of modal logic, instead of assertion, often used to specify properties in the verification of concurrent software, e.g., it can be used to specify a property for judging whether the value of a vari-

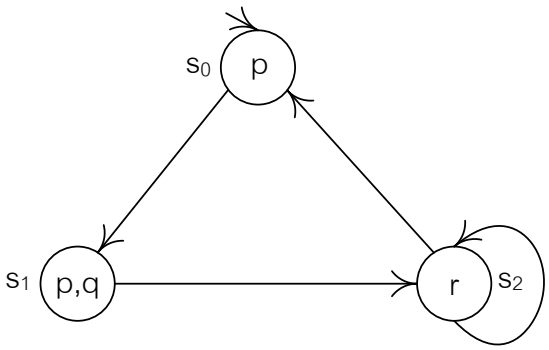



Figure 2.3: An example for Kripke structure

able will become a specified value in the future. In this part, based on Kripke structure [14] and paths of Kripke structure, we will firstly introduce the semantics of LTL, and then show several examples to explain some general properties specified in LTL.

**Definition 1** *A Kripke structure $M$ is a tuple $M = (S, I, R, L)$. $S = \{s_1, s_2, ..., s_n\}$ is the finite set of states. $I \subseteq S$ is the set of initial states. $R \subseteq S \times S$ is the transition relation, where $\forall s \in S, \exists s' \in S$ such that $(s, s') \in R$. $L : S \to 2^{AP}$ is labelling function from states to the power set of atomic propositions $AP$, where for a state $s \in S$, the set $L(s)$ is made of the atomic propositions that hold in $s$.*

For example, a kripke structure is shown in Fig. 2.1. In the shown example, $S = \{s_0, s_1, s_2\}$, and $p, q, r$ are the atomic propositions.

**Definition 2** *A path $\pi$ of a Kripke structure $M$ is an infinite sequence $s_0, s_1, \cdots$ of states, where $\forall i, \langle s_i, s_{i+1} \rangle \in R$. $\pi^i = s_i, s_{i+1}, \cdots$ is the subpath of $\pi$ starting from state $s_i$, and $\pi(i)$ denotes the $i$-th state $s_i$ of path $\pi$. If the starting state $\pi(0)$ of a path $\pi$ is an initial state (i.e., $\pi(0) \in I$), then we say that the path $\pi$ is initialized.*

For example, as shown in Fig. 2.3, the path $\pi = (s_0, s_1, s_2, s_2, \cdots)$ is an initialized path in the shown Kripke structure, where $\pi(0)$ is the initial state $s_0$.

In general, LTL inherits propositional variables (in mathematical logic, a propositional variable is the variable which can either be true or false) and logic operators such

as negation $\neg$, conjunction $\wedge$, disjunction $\vee$ and implication $\rightarrow$, etc. In addition to the propositional variables and logic connectives, LTL formula holds several temporal operators, such as the next time operator $\mathtt{X}$, the global operator $\mathtt{G}$, the liveness operator $\mathtt{F}$, the binary temporal operators Until ($\mathtt{U}$) and Release ($\mathtt{R}$). The syntax of LTL over atomic propositions $AP$ with $p \in AP$ is stated below,

$$\varphi ::= true|false|p|\varphi_1 \wedge \varphi_2|\varphi_1 \vee \varphi_2|\varphi \rightarrow \varphi|\neg\varphi|\mathtt{X}\varphi|\mathtt{F}\varphi|\mathtt{G}\varphi|\varphi\mathtt{U}\varphi|\varphi\mathtt{R}\varphi$$

For example, for the given example shown in Fig. 2.3, we can specify a LTL formula $\varphi = \mathtt{F}(p \wedge q)$ to verify the system.

**Definition 3** *Let $\pi$ be a path in Kripke structure $M$, $\varphi$ be a LTL formula. The LTL formula $\varphi$ is held by the path $\pi$ iff $M, \pi \models \varphi$, where*

$M, \pi \models true \qquad M, \pi \not\models false$

$M, \pi \models p \qquad iff \quad p \in L(\pi(0))$

$M, \pi \models \neg\varphi \qquad iff \quad M, \pi \not\models \varphi$

$M, \pi \models \varphi_1 \wedge \varphi_2 \quad iff \quad M, \pi \models \varphi_1 \text{ and } M, \pi \models \varphi_2$

$M, \pi \models \varphi_1 \vee \varphi_2 \quad iff \quad M, \pi \models \varphi_1 \text{ or } M, \pi \models \varphi_2$

$M, \pi \models \varphi_1 \rightarrow \varphi_2 \quad iff \quad M, \pi \not\models \varphi_1 \text{ or } M, \pi \models \varphi_2$

$M, \pi \models \mathtt{G}\varphi \qquad iff \quad \text{for all paths } \pi^i \text{ for all } i \geq 0,\ M, \pi^i \models \varphi$

$M, \pi \models \mathtt{F}\varphi \qquad iff \quad \text{for some path } \pi^i \text{ for some } i \geq 0,\ M, \pi^i \models \varphi$

$M, \pi \models \mathtt{X}\varphi \qquad iff \quad M, \pi^1 \models \varphi$

$M, \pi \models \varphi_1\mathtt{U}\varphi_2 \qquad iff \quad \text{exists } i \geq 0 \text{ such that } M, \pi^i \models \varphi_2 \text{ and for all } j < i,\ M, \pi^j \models \varphi_1$

$M, \pi \models \varphi_1\mathtt{R}\varphi_2 \qquad iff \quad \text{exists } i \geq 0 \text{ such that } M, \pi^i \models \varphi_1 \text{ and for all } j \leq i,\ M, \pi^j \models \varphi_2 \text{ or for all } i \geq 0\ M, \pi^i \models \varphi_2$

For the example shown in Fig. 2.3, according to the semantics of LTL stated in definition 3, we can easily find that the system holds the LTL formula $\varphi = \mathtt{F}(p \wedge q)$.

## 2.2 Bounded Model Checking

The original motivation of bounded model checking was to leverage the success of satisfiability (SAT) [39][41] in solving Boolean formulas to model checking. Compared to explicit state model checking technique and BDD-based model checking technique [6][45], it offers the advantage of handling the verification of large state spaces. The main idea of BMC is to avoid the full state space generation and look for witnesses of an existential specification on the suitable subsets of the full model under the given bound. Once a sub-model is selected, the considered sub-model will be translated into propositional formulae. Then, the propositional formulae and the negative form of given specification (given property) will be solved by a specialized SAT solver. If the verification result is true, it means that the sub-model does not satisfy the given specification, a counterexample will be report by the SAT solver. Otherwise, a larger sub-model will be selected and the whole procedure will be run again until the whole model is verified.

### 2.2.1 Transition system with Bound

Bounded model checking consists of a transition system $M$, a given property $f$ and a user-supplied bound $k$ ($k \geq 0$). The transition system $M$ is a conjunctive normal formula (CNF), used to model the target system. The given property can be stated as a LTL formula. The user-supplied bound $k$ is used to limit the verification space. If given a system $M$, a bound $k$ and a property $f$, we can use the propositional formula shown in definition 4 to carry out verification based on the SAT solver.

**Definition 4** $[\![M, \neg f]\!]_k := I(s_0) \wedge (\bigwedge T(s_k, s_{k+1})) \wedge [\![\neg f]\!]_k$

Where, $I(s_0)$ is the initial function used to initialize the values of variables declared in the system, $T(s_k, s_{k+1})$ is the transition relation.

For example, as shown in Fig. 2.4, for a 2-bit counter which is taken from paper [24], if the bound $k$ is set to 2, we can use the following transition system $[\![M]\!]_2$ to model the

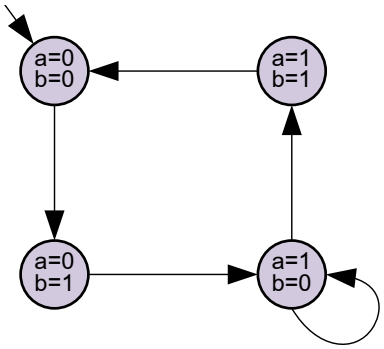



Figure 2.4: A 2-bit counter

system, where the initial function $I(s_0)$ is $(a_0 = 0 \wedge b_0 = 0)$.

$$[\![M]\!]_2 := I(s_0) \wedge (a_0 = 0 \wedge b_0 = 0) \wedge (a_1 = 0 \wedge b_1 = 1) \wedge (a_2 = 1 \wedge b_2 = 0)$$

### 2.2.2 LTL Property in Bounded Model Checking

Bounded model checking supports the property specified in LTL formula in the verification. The semantics of LTL property $f$ in bounded model checking are shown in definition 5.

**Definition 5** *Let $k \geq 0$, and let $\pi$ be a path in transition system $M$. Then a LTL formula $f$ is valid along $\pi$ with bound $k$ ($\pi \models_k f$) iff $\pi \models_k^0 f$, where*

$\pi \models_k^i p$ *iff* $p \in L(\pi(i))$

$\pi \models_k^i \neg p$ *iff* $p \notin L(\pi(i))$

$\pi \models_k^i f \wedge g$ *iff* $\pi \models_k^i f$ *and* $\pi \models_k^i g$

$\pi \models_k^i f \vee g$ *iff* $\pi \models_k^i f$ *or* $\pi \models_k^i g$

$\pi \models_k^i \mathtt{G}f$ *is always false*

$\pi \models_k^i \mathtt{F}f$ *iff* $\exists j, i \leq j \leq k . \pi \models_k^i f$

$\pi \models_k^i \mathtt{X}f$ *iff* $i \leq k$ *and* $\pi \models_k^{i+1} f$

$\pi \models_k^i f \mathtt{U} g$ *iff* $\exists j, i \leq j \leq k . \pi \models_k^j g$ *and* $\forall n, i \leq n \leq j . \pi \models_k^n f$

$\pi \models_k^i f \mathtt{R} g$ *iff* $\exists j, i \leq j \leq k . \pi \models_k^j f$ *and* $\forall n, i \leq n \leq j . \pi \models_k^n g$

Based on the semantics shown in definition 5, the following functions can be used to translate a LTL property $f$ into the verification expression with bound $k$, which have been stated in paper [14]. In the translation function, $i \leq k$, $succ(i) := i + 1$ and $[\![f]\!]_k^{k+1} := 0$.

$$[\![p]\!]_k^i := p(s_i)$$

$$[\![\neg p]\!]_k^i := \neg p(s_i)$$

$$[\![f \wedge g]\!]_k^i := [\![f]\!]_k^i \wedge [\![g]\!]_k^i$$

$$[\![f \vee g]\!]_k^i := [\![f]\!]_k^i \vee [\![g]\!]_k^i$$

$$[\![\mathtt{G}h]\!]_k^i := [\![h]\!]_k^i \wedge [\![\mathtt{G}h]\!]_k^{succ(i)}$$

$$[\![\mathtt{F}h]\!]_k^i := [\![h]\!]_k^i \vee [\![\mathtt{F}h]\!]_k^{succ(i)}$$

$$[\![\mathtt{X}h]\!]_k^i := [\![\mathtt{X}h]\!]_k^{succ(i)}$$

$$[\![h\mathtt{U}g]\!]_k^i := [\![g]\!]_k^i \vee ([\![h]\!]_k^i \wedge [\![h\mathtt{U}g]\!]_k^{succ(i)})$$

$$[\![h\mathtt{R}g]\!]_k^i := [\![g]\!]_k^i \wedge ([\![h]\!]_k^i \vee [\![h\mathtt{R}g]\!]_k^{succ(i)})$$

For example, for the given example shown in Fig. 2.4, if we given a LTL property $f = \mathtt{F}(a = 1)$ and set bound $k$ to 2, the below expression can be used to verify the system with the given LTL property $f$.

$$[\![M, \neg f]\!]_2 := (a_0 = 0 \wedge b_0 = 0) \wedge (a_1 = 0 \wedge b_1 = 1) \wedge (a_2 = 1 \wedge b_2 = 0) \wedge (\neg(a_0 = 1 \vee a_1 = 1 \vee a_2 = 1))$$

### 2.2.3 Satisfiability Modulo Theories (SMT)

In bounded model checking, a SAT solver [8][28][40] is usually employed to check whether the constructed transition system $M$ satisfies the given property $f$ (i.e., $M \models_k f$). Although the SAT-based BMC [38] is more efficient and scalable than explicit state model checking technique and BDD model checking technique, the checking ability of SAT-based BMC is still limited by the solving performance of SAT solver when checking a complex

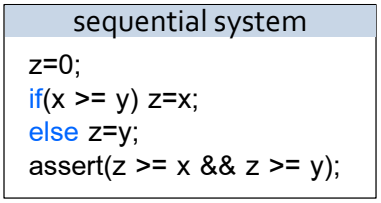



Figure 2.5: A sequential C system

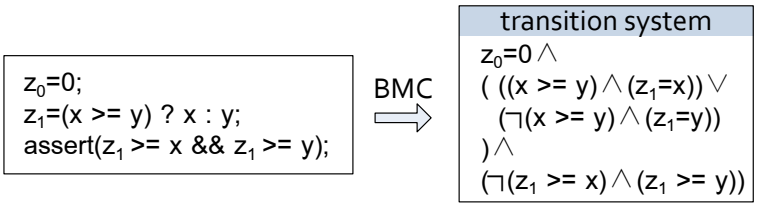



Figure 2.6: The transition system for the given sequential C system

system which holds a large number of states. Fortunately, an important research [11] of checking a transition system with satisfiability modulo theory (SMT) solver [17] instead of SAT solver shows that SMT not only can check more conditions of the transition system within shorter time than SAT solver, but also supports arithmetic, bitvectors, tuples, arrays, and other decidable first-order theories which allow us to more conveniently translate the transition system $M$ into SMT solver.

Currently, based on the satisfiability modulo theory, there exist many SMT solvers that have been implemented by researchers and software company. E.g., Microsoft Research group has developed a SMT solver named Z3 [35], which has been widely used in the research field and industry of software. Yices [20] developed by SRI is also a famous and efficient SMT solver. In addition, there are other SMT solvers that are using in the academic institution, such as MiniSmt [27], Mistral [29] and OpenSMT [15].

### 2.2.4 SMT-based BMC for Sequential Software and Concurrent Software

SMT-based BMC has been widely applied in the verification of sequential software and concurrent software. The key work of using BMC to verify a system is how to construct

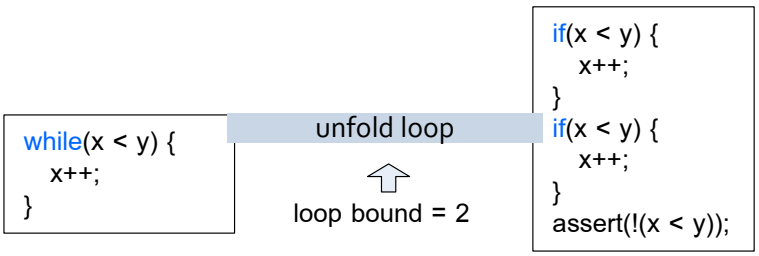



Figure 2.7: BMC for loop

a transition system $M$. Firstly, let us consider a sequential C system which is shown in Fig. 2.5. In the system, there is a branch that is used to return the maximum value of variables $x$ and $y$ to variable $z$. To construct a transition system $M$ for the system, an easy idea is to explore execution paths, and then construction the transition system $M$ based on the explored paths. However, there is a shortcoming, that is, if a system consists of a lot of branches, a large number of execution paths will be explored, which will significantly slow down the performance of BMC. To efficiently verify a sequential system which holds a lot of branches, in BMC the single static assignment (SSA) technique [18] is employed to combine branches before applying BMC to carry out verification. This idea has been stated in paper [11]. E.g, as shown in Fig. 2.6, for the system shown in Fig. 2.5, we can firstly applied SSA technique to eliminate the branch, and then use the principle of BMC to construct a transition system $M$ for the system.

In addition to the branches, a sequential system usually consists of loops. Unfortunately, the satisfiability modulo theory based SMT solver does not support the repetition structure (SAT solver also does not support repetition structure). As to successfully employ SMT solver to verify the system which holds loops, in BMC the loop within the system will be firstly unfolded as branch structure according to a loop bound. Particularly, in order to judge whether the loop is unfolded enough or not, an assertion which hold the negative condition of the loop is inserted into the end of the unfolding loop. E.g., for the given example shown in Fig. 2.7, the loop has been unfolded as branch structure when the loop bound is set to 2. Based on the unfolding process, BMC then can verify the system. Actually, in general BMC, there exist two types of bounds, one is the depth bound which is used to limit the search space, and the other is loop bound which is used to unfold loops. Based on the SMT-based BMC technique, a tool named

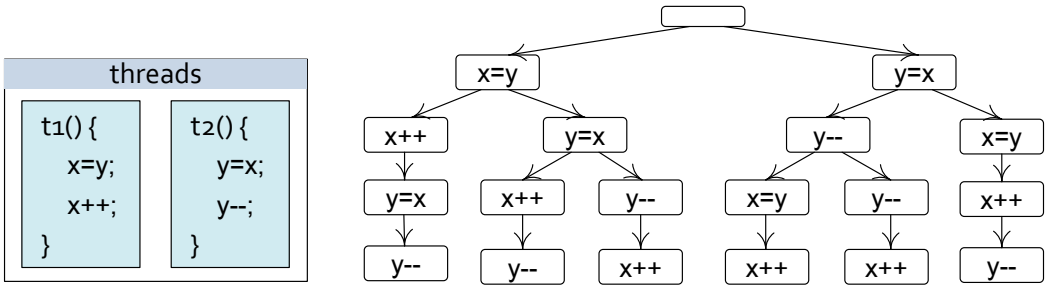



Figure 2.8: A concurrent ANSI-C system and its reachability tree

`cbmc` [16] has been established for the sequential C software.

In contrast with sequential software, the checking process of applying BMC to verify a concurrent system is more complex because of the scheduling behaviours. E.g., there is a ANSI-C concurrent system in Fig. 2.8. According to the ANSI-C standard, which thread is to be run is arbitrarily determined by ANSI-C scheduler. Usually, we named this type of scheduler as non-deterministic scheduler. Due to the non-deterministic scheduling behaviours, the running thread cannot be explicitly determined in the verification. In BMC, as to completely verify such system, all of the possible interleavings of threads will be taken into account in the checking process. In general, a tree named reachability tree is first constructed to demonstrate all of the interleavings, and then the constructed reachability tree is translated into the corresponding transition system $M$. E.g., for the given example shown in Fig. 2.8, the reachability tree illustrated in the right side of Fig. 2.8 can be used to demonstrate all of the interleavings of threads $t_1$ and $t_2$. According to the above presentation, BMC can exhaustively verify a concurrent system. Based on the SMT-based BMC technique, a tool named `esbmc` has been established for the multi-threaded software that conforms to ANSI-C standard.

## 2.3 Advantage and Disadvantage of Model Checking

Currently, model checking as an efficient checking technique has been widely applied to verify concurrent software. Moreover, with the development of model checking such as the improvement of automatization and application of advanced searching techniques, it has become an important checking technique in the verification of software. Compared with other checking techniques such as testing technique and theorem proving technique, model checking technique holds the

following advantages and disadvantages.

### 2.3.1 Advantages of Model Checking

Compared with testing technique, there are several advantages in model checking technique.

- If we apply testing technique to check a concurrent software, it is difficult to reproduce the error track although it can successfully find the error. However, based on the counterexample that is reported by model checking technique, we can easily constructed an error track for the detected bug and the counterexample clearly shows why the specification does not hold.

- Compared with testing technique, model checking supports temporal logical such as LTL in the verification of software, it can easily express many of the properties that are needed for reasoning about concurrent systems.

- If we apply testing technique to check a software which holds non-deterministic behaviours, it is difficult to completely check the software because of incomplete test-case set. As Edsger Dijkstra said: program testing can at best show the presence of errors but never their absence. However, based on the advanced satisfiability modulo theories (SMT), the SMT-based model checking can efficiently and completely verify the software which holds non-deterministic behaviours.

- If we apply testing technique to check a software which holds unbounded loops, it is difficult to judge whether the software satisfies a given property or not because of infinite states. However, based on the advanced $k$-induction and invariants method, model checking technique can successfully verify the software which holds unbounded loops.

In addition, compared with other formal verification techniques such as theorem proving, there are several advantages in model checking technique.

- Model checking is usually an automatic verification technique. Based on the different model checking techniques, there are many corresponding tools have been implemented such as `spin`, `cbmc` and `SMV`. Compared with theorem proving, in the verification process user does not need to construct a correctness proof.

- Compared with theorem proving, model checking technique is more efficient and scalable. In practice, as Clarke said in book titled "25 years of model checking": the other formal verification techniques may require months of time to prove a system.

### 2.3.2 Disadvantages of Model Checking

However, there is an important disadvantage that need to be solved or overcome in model checking technique, namely state space explosion problem. This is because, for a complex software, as to describe the all of the possible executions of the software, in model checking a huge checking model will be constructed in the verification process, which can easily make verification failed because of the memory or time limitations, although the advanced hardware has been used in our machines.

However, model checking still holds its own advantages in the development of software. E.g., for a complex software, we can firstly abstract a prototype model that holds the important behaviours of the target software rather than all of the details, and then apply model checking to verify the abstracted model with specification. Based on the verified model, developers can easily implement a more correct system. In addition, in the checking process we can applied model checking technique to check some key parts rather than whole software, e.g., the parts which hold non-deterministic behaviours.

# Chapter 3

# OSEK/VDX

OSEK/VDX, a standard of automobile OS promulgated by the alliance of German and French automotive manufacturers in 1994. The primary motivation of OSEK/VDX standard is to resolve the problem of increasing software content in vehicles and desire high-quality products. Currently, it has been widely adopted by many automobile manufacturers to design and develop a vehicle-mounted OS, such as BMW, Opel and TOYOTA. Based on the OSEK/VDX OS, more and more applications are developed and deployed in vehicles to assist drivers to control vehicles, such as the cruise control system and temperature control system. In this chapter, based on the specification of OSEK/VDX, we will briefly introduce OSEK/VDX OS and its applications.

## 3.1 OSEK/VDX OS

A general OSEK/VDX OS consists of a scheduler module, synchronization event process module, shared resource process module, alarm process module and interruption process module. Based on these system modules, OSEK/VDX OS supports a standardized application interfaces (APIs) for user to develop customized applications. In our research, we focus on the applications that communicate with scheduler module, synchronization event process module and resource process module. The structure of OSEK/VDX OS with an application is shown in Fig. 3.1.

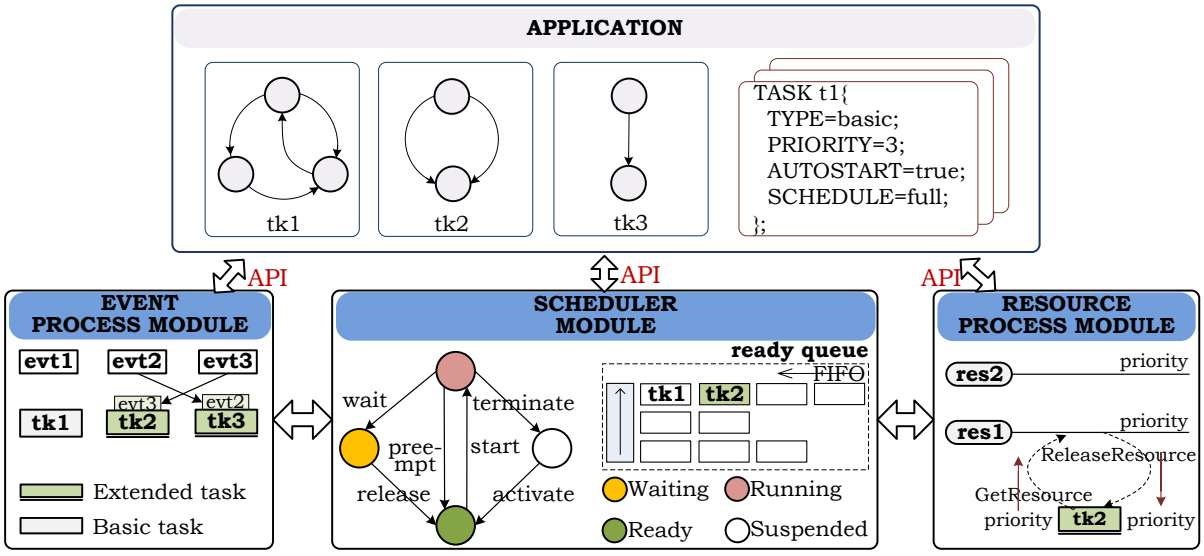



Figure 3.1: The structure of OSEK/VDX OS with an application

### 3.1.1 Scheduler Module

OSEK/VDX OS can process two types of tasks, basic task and extended task. The states of a basic task consist of *running* state, *suspended* state, and *ready* state. Compared with the basic task, the extended task can hold synchronization events and has a unique state called *waiting* state. In the scheduling process, the *static priority scheduling policy* with non-preemptive and full-preemptive strategies is adopted by the scheduler to conduct the executions of tasks. Particularly, in OSEK/VDX OS, scheduler manages a ready queue to lay out the execution order of tasks. Based on the ready queue, the running task is explicitly determined by OSEK/VDX scheduler.

In addition, scheduler module provides several APIs for applications, such as *TerminateTask*, *ActivateTask* and *ChainTask*. Tasks within the application can invoke these APIs to dynamically change the states of tasks, and the changed states will affect the scheduling of tasks, e.g., the context switch of tasks will happen when a suspended task is activated by running task using API *ActivateTask* or *ChainTask*. For instance, as shown in Fig. 3.2, there are two tasks $t1$ and $t2$, and the priority of task $t1$ is lesser than $t2$. Currently, task $t1$ is running task. When the API *ActivateTask*($tk2$) is invoked by $t1$, scheduler will move task $t2$ from *suspended* state to *ready* state. At this moment, the context switch of tasks will happen, because the priority

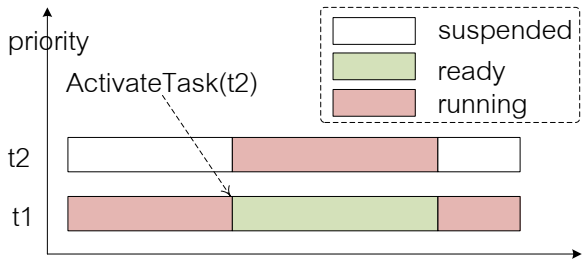



Figure 3.2: The context switch of tasks and API *ActivateTask*

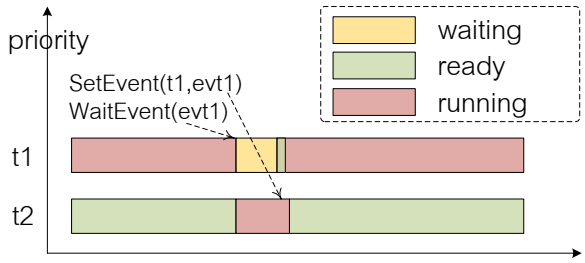



Figure 3.3: The synchronous behaviours between tasks

of task $t2$ is higher than $t1$.

### 3.1.2 Event Process Module

In the synchronization event process module, OSEK/VDX OS provides a synchronization mechanism for implementing synchronous executions between tasks. Particularly, only extended tasks can hold a definite number of events, and events are the criteria for the switching of task states from *running* state to *waiting* state or from *waiting* state to *ready* state. There are three APIs $SetEvent$, $WaitEvent$ and $ClearEvent$ that can be responded by event process module, and tasks within application can invoke these APIs to implement the synchronous behaviours. E.g., as shown in Fig. 3.3, when the running task $t1$ waits for the event $evt1$ using API $WaitEvent(evt1)$, task $t1$ cannot continue until the event $evt1$ is set by other tasks (basic tasks or extended tasks) using API $SetEvent(t1,evt1)$.

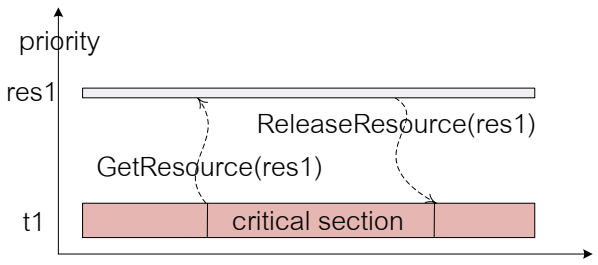



Figure 3.4: The priority ceiling protocol

### 3.1.3 Resource Process Module

OSEK/VDX OS adopts the *Priority Ceiling Protocol* [9] to coordinate the behaviors of task accessing shared resources in the resource process module. The resource process module supports two APIs (*GetResource* and *ReleaseResource*) that can be invoked by tasks to access shared resources. E.g., as shown in Fig. 3.4, if the API *GetResource*($res1$) is invoked by *running* task $t1$, and the priority of the task is lower than the ceiling priority of the resource $res1$, the priority of the task will be raised to the ceiling priority of the resource $res1$, and the priority of the task will be reset to the priority before requiring the resource $res1$ when *ReleaseResource*($res1$) is invoked by the task. Note that, the ceiling priority of a shared resource is lower than the lowest priority of all tasks that do not access the resource, and it is higher than the priorities of all tasks that access the resource.

## 3.2 OSEK/VDX Application and Execution Characteristics

### 3.2.1 OSEK/VDX Application

An application developed based on OSEK/VDX OS consists of two files, one is the source file, and the other is the configuration file. The source file which can be developed by C language is used to present the concrete behaviors of the application. The configuration file is used to define tasks, synchronization events and shared resources.

In the configuration file, the attribute `AUTOSTART` is used to set the initial state of tasks.

Source file (.cpp)

```
int a, b, c, h;

TASK t1(){
  if(b >= a){
    a++;
    ActivateTask(t2);
  }
  else{
    b++;
    ActivateTask(t3);
  }
  ActivateTask(t4);
  TerminateTask();
}
TASK t2(){
  if(h > 0)
    h--;
  else
    h++;
  TerminateTask();
}
TASK t3(){
  c=b;
  c=(c+b)/b;
  TerminateTask();
}
TASK t4(){
  assert(a > b);
  TerminateTask();
}
```

Configuration file (.oil)

```
TASK t1{
  TYPE=BASIC;
  SCHEDULE=FULL;
  PRIORITY=5;
  AUTOSTART=TRUE;
};

TASK t2
{
  TYPE=BASIC;
  SCHEDULE=NON;
  PRIORITY=6;
  AUTOSTART=FALSE;
};

TASK t3
{
  TYPE=BASIC;
  SCHEDULE=NON;
  PRIORITY=4;
  AUTOSTART=FALSE;
};

TASK t4
{
  TYPE=BASIC;
  SCHEDULE=NON;
  PRIORITY=4;
  AUTOSTART=FALSE;
};
```

Figure 3.5: An OSEK/VDX application

If the attribute `AUTOSTART` of a task is set to be `TRUE`, the task starts from *ready* state in the initial state (it will be inserted into ready queue according to the priority of the task). Otherwise, the task starts from *suspended* state. The attribute `SCHEDULE` is used to indicate the scheduling type. If the attribute `SCHEDUL` of a task is set to be `FULL`, the task can be preempted by higher priority tasks. Otherwise, the task cannot be preempted by higher priority tasks. A simple OSEK/VDX application without synchronization events and shared resources is shown in Fig. 3.5. As to clearly comprehend the execution characteristics of OSEK/VDX applications, an example is symbolically executed in this part.

In the application shown in Fig. 3.5, only the attribute `AUTOSTART` of $t1$ is set to be `TRUE`. Thus, $t1$ will be firstly moved to *running* state by scheduler and then executed in the initial

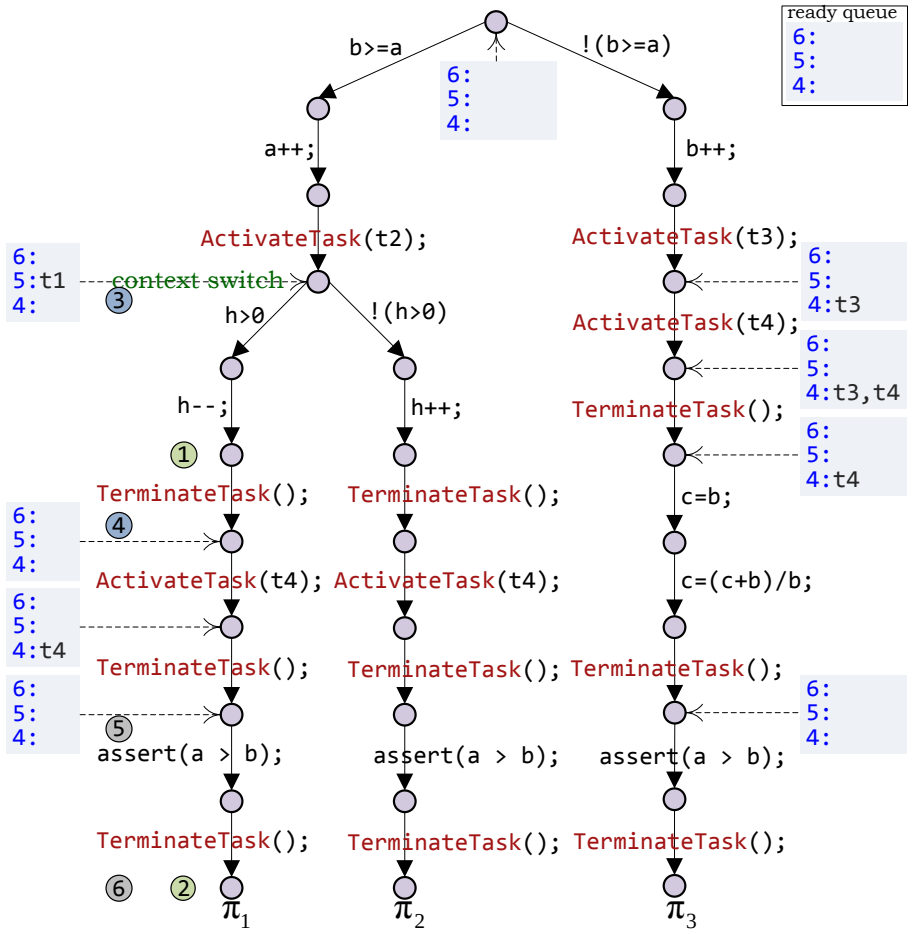



Figure 3.6: The execution tree for the application shown in Fig. 3.5

state. As shown in Fig. 3.6, when the API $ActivateTask(t2)$ is invoked by $t1$ (the branch "$(b >= a)$" is chosen to symbolically execute), scheduler will be loaded to respond to the API (task $t2$ is moved from *suspended* state to *ready* state). At this moment, the running task $t1$ will be preempted by $t2$ since the priority of $t2$ is higher than $t1$ and the attribute `SCHEDUL` of $t1$ is set to be `FULL` (the context switch happens after the API $ActivateTask(t2)$). Currently, task $t2$ gets run-unit to run, and goes to *suspended* state when the API $TerminateTask()$ is invoked ($TerminateTask()$ is used to terminate the executions of a task, and the terminated task will be moved from *running* state to *suspended* state by scheduler. If the running task is terminated, scheduler then dispatches the head task in the ready queue to run when ready queue is not empty). When task $t2$ terminates itself using API $TerminateTask()$, scheduler will dispatch task $t1$ to *running* state. Then, $t1$ continues its executions from the preempted point, and task

$t4$ is activated by $t1$ using API $ActivateTask(t4)$ (the context switch will not happen, since the priority of task $t4$ is less than $t1$). Finally, task $t4$ will be executed when the *running* task $t1$ terminates itself using API $TerminateTask()$. Thus, according to the conducted symbolic executions, we can get a task execution sequence $\langle t1, t2, t1, t4\rangle$ in the execution path $\pi_1$ and $\pi_2$. However, if the API $ActivateTask(t3)$ is invoked by task $t1$ (the "branch $!(b >= a)$" is chosen to symbolically execute), we will get a different task execution sequence $\langle t1, t3, t4\rangle$ in the execution path $\pi_3$.

### 3.2.2 Execution Characteristics of OSEK/VDX Application

Based on the execution tree of the application shown in Fig. 3.5, we can easily find the following execution characteristics of OSEK/VDX applications,

- Which task is to be run is explicitly determined by OSEK/VDX scheduler according to the ready queue and configuration data of application.
- The APIs invoked from tasks will haphazardly change the scheduling of tasks, e.g., the different service APIs locating at different branches will lead to different task execution sequences, and the context switch of tasks may happen when an API is invoked by running task.

According to the listed execution characteristics, there are several challenges that should be addressed when we apply model checking technique to verify an OSEK/VDX application, e.g.,

- How to precisely construct a model to represent the executions of the application.
- How to deal with OSEK/VDX OS, e.g., how to explicitly perform the scheduling behaviours of OSEK/VDX OS, and how to handle the APIs invoked from tasks.

We have proposed three approaches to verify OSEK/VDX applications in this thesis. The overview of the proposed approaches will be demonstrated in the next chapter.

# Chapter 4

# Overview of Checking Approaches

Based on the execution characteristics of OSEK/VDX applications, we have found, the running task within application is explicitly determined by OSEK/VDX scheduler (deterministic scheduling policy is used to dispatch tasks), and the APIs invoked from tasks will haphazardly affect the scheduling of tasks, e.g., (*i*) the context switch of tasks may happen when an API is invoked, (*ii*) the different APIs locating at different branches will lead to different task execution order. Although there exist many model checking methods [26][50][51] that have been applied to verify concurrent software such as the ANSI-C multi-threaded software and SystemC multi-threaded software, these methods usually perform an approximate verification because the behaviours of scheduler are not taken into account in verification process. If we apply these existing model checking methods to verify OSEK/VDX applications, a lot of unnecessary interleavings of tasks will be superfluously checked in the verification stage, especially these unnecessary interleavings will usually result in a spurious bug. Due to the spurious bug, developers have to spend extra costs judging whether the detected bug is real one or not after completing verification.

In addition, to accurately check concurrent software such as ANSI-C multi-threaded software and SystemC multi-threaded software, several prominent model checking methods [13][37] have been proposed by some senior researchers. In these methods, as to seek a more accurate verification result, the behaviours of scheduler are taken into account in the verification process. Even so, these prominent model checking methods are still unsuitable to accurately verify OSEK/VDX applications because these methods focus on the non-deterministic scheduler based multi-threaded software. In contrast with non-deterministic scheduler, in OSEK/VDX applica-

tions a deterministic scheduler is used to dispatch tasks. As to reduce the checking costs on the OSEK/VDX applications, a more accurate model checking approach should be proposed and applied in the verification process. In this thesis, we describe and develop three approaches that can accurately and automatically verify the safety property of OSEK/VDX applications using model checking technique.

## 4.1 Key Idea of Checking Approaches

The core of model checking consists of two processes, one is to construct a model for target system, and the other is to search the constructed model with given property. As to make model checking accurately verify OSEK/VDX applications, there are two ways that can be considered in the verification of OSEK/VDX applications, e.g.,

1. We can construct an accurate model for target OSEK/VDX application, and then apply existing searching algorithm to check the constructed checking model.

2. We do not desire an accurate model, but reform the existing searching algorithms to make the existing searching algorithms perform an accurate search in the verification process.

In our approaches, we will construct an accurate checking model to achieve the accurate verification based on the model checking technique. Compared with the method for reforming searching algorithms, there is an advantage in this method, e.g., once an accurate model is constructed, we can directly use different searching algorithms to accurately verify OSEK/VDX applications. As we know, in the verification of software, we usually should use different model checking techniques to exhaustively find bugs. Compared with the method for constructing accurate model, the method for reforming the existing searching algorithms will usually spend more costs reforming the different searching algorithms.

To construct an accurate model for a target OSEK/VDX application, the behaviours of OSEK/VDX OS such as scheduler behaviours should be taken into account in the constructing model process. This is because, when an OSEK/VDX application runs on the OSEK/VDX OS, the running task is explicitly determined by OSEK/VDX scheduler and the APIs invoked from tasks will haphazardly affect the scheduling of tasks. Based on this idea, we have proposed three

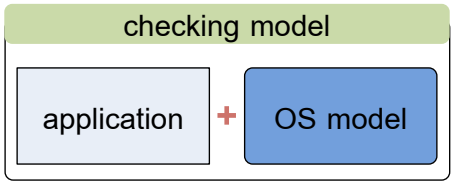



Figure 4.1: The key idea of Spin-based checking approach

approaches that can accurately and automatically verify the developed OSEK/VDX applications using model checking techniques.

## 4.2 Overview of Checking Approaches

In our first approach, we investigate a method based on the existing model checker Spin. In the method, as to make Spin model checker accurately verify OSEK/VDX applications, a checking model which is a combination model of application model and OSEK/VDX OS model is employed to precisely simulate the executions of the OSEK/VDX applications in real OSEK/VDX OS. In the second approach, as to verify a complex application which holds a large number of states, we develop a new technique named execution path generator (EPG) to verify OSEK/VDX applications based on the advanced SMT-based bounded model checking technique. Particularly, in the approach the OSEK/VDX OS model is embedded in EPG (the algorithm of constructing model flat) to respond to the invoked APIs and determine running task in the process of constructing application model. In the last approach, we present a novel method to translate OSEK/VDX applications into sequential models in order to apply the existing model checkers such as `spin` and `cbmc` to efficiently verify OSEK/VDX applications.

### 4.2.1 Spin-based Checking Approach

To accurately verify OSEK/VDX applications using model checking technique, a cheap method is based on the existing model checker, because we do not need to establish a new model checker. As we know, it is too difficult and impractical to establish a new model checker within a short time, since a basic model checker usually consists of several complex modules such as the language parsing, simulation, verification, counterexample generation and display. To finish

all of these modules with reasonable soundness guarantee often takes years of effort, e.g., the established model checkers `spin` and `cbmc` are the result of decades of development. As to apply existing model checkers to accurately verify OSEK/VDX applications, we investigate an approach based on the well-know Spin model checker.

To accurately verify OSEK/VDX applications using Spin model checker, the key work is how to construct an accurate checking model to feed Spin. In our Spin-based checking approach, as shown in Fig. 4.1, we design a checking model which is a combination model of application model and OSEK/VDX OS model to precisely simulate the executions of the target application. In the combination model, all of the tasks within application are designed as `process`es, and the OSEK/VDX OS model as a cooperative `process` is used to dispatch tasks and respond to the APIs invoked from tasks. In particular, the application model and OSEK/VDX OS model are synchronously executed based on the invoked APIs for simulating the executions of the application in real OSEK/VDX OS.

The advantage of the approach is that, it can accurately verify OSEK/VDX applications based on the existing model checker Spin. However, the scalability of this approach is limited, because too many details of OSEK/VDX OS model will be explored in the verification stage, especially the state space explosion may happen if application invokes a lot of APIs (when an API is invoked by running task, OSEK/VDX OS model will be checked once by Spin in the verification process). Based on the shortcoming of the approach, we can easily find that the approach cannot be applied to verify the complex OSEK/VDX applications which will invokes a lot of APIs. In order to efficiently check a complex OSEK/VDX application which holds a lot of states and APIs, we develop a new technique named execution path generator (EPG) based on the advanced SMT-based bounded model checking.

### 4.2.2 SMT-based Bounded Model Checking Approach

The core of our SMT-based bounded model checking approach is execution path generator (EPG), which is a new technique to verify OSEK/VDX applications using bounded model checking technique. In our EPG technique, as to efficiently handle a complex application which holds a large number of states, the advanced SMT-based bounded model checking is adopted to perform the verification. Particularly, in order to avoid OSEK/VDX OS model to be explored in the

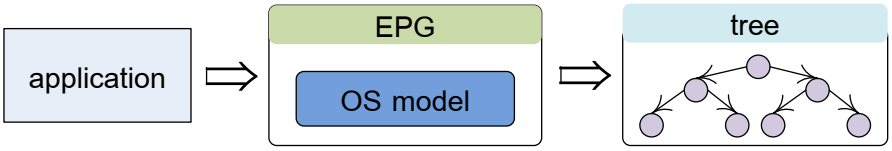



Figure 4.2: The key idea of SMT-based BMC approach

verification stage, as shown in Fig. 4.2, we embed OSEK/VDX OS model in EPG (the algorithm of constructing model flat) to dispatch task and respond to the APIs invoked from tasks.

To verify an OSEK/VDX application using bounded model checking technique, the key work is how to construct a transition system for the application. In OSEK/VDX applications, the APIs invoked from tasks will haphazardly affect the scheduling of tasks, e.g., the context switch of tasks may happen when an API is invoked, and the different APIs location at different branches will lead to different task execution orders. In our EPG technique, like other bounded model checking for concurrent software (e.g., Lucas' method [37] for ANSI-C multi-threaded software), a reachability tree as an intermediate form is constructed for establishing the transition system of application.

The advantage of the approach is that, it is more efficient and scalability than the Spin-based checking approach. This is because, in EPG technique the state-of-the-art SMT-based bounded model checking is employed to perform the verification, which make EPG technique more efficient. Moreover, in EPG technique the OSEK/VDX OS model will not be explored in the verification stage since the OSEK/VDX OS model is embedded in EPG to dispatch tasks and respond to the APIs invoked from tasks, which make EPG technique more scalable. However, we can easily find that the approach is not efficient to check the large-scale applications which hold a lot of branches, because EPG will spend a lot of time exploring execution paths for constructing the reachability tree in the checking process, which will slow down the efficiency of the approach. As to efficiently handle the applications which hold a large number of branches, we propose a novel approach based on the sequentialization idea.

### 4.2.3 Sequentialization-based Checking Approach

The key idea of our sequentialization-based checking approach is that, we firstly translate a given OSEK/VDX application into a corresponding sequential model, and then employ the

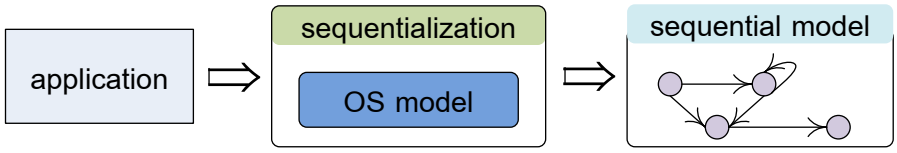



Figure 4.3: The key idea of sequentialization-based checking approach

existing model checker such as `spin` and `cbmc` to verify the sequential model. According to the shortcoming of EPG technique, we have found that EPG technique is not efficient to verify the large-scale applications which hold a lot of branches, because it will spend a lot of time exploring execution paths for establishing transition system. As to efficiently verify the applications which hold a lot of branches, the core work is how to efficiently construct a sequential model for the target application. In our sequentialization-based checking approach, we use an extended directed graph instead of reachability tree to construct the sequential model of OSEK/VDX application. Particularly, as to avoid the behaviours of OSEK/VDX OS to be poured into the sequential model, as shown in Fig. 4.3, like EPG technique we embed OSEK/VDX OS model in translation algorithm flat to dispatch tasks and respond to the invoked APIs. The main process of our sequentialization-based checking approach is that, we symbolically execute application using an extended directed graph and explicitly perform the scheduling behaviours when meeting an API in the sequentialization process.

There are several strengths from extended directed graph that can make our sequentialization-based checking approach more efficient than EPG technique in checking the applications which hold a lot of branches, e.g.,

- Based on the directed graph, we do not need to explore execution paths when meeting branches in the sequentialization process, instead, we can use the combination states to reduce the computation times. E.g., for the example shown in Fig. 4.4, if we use tree to describe the executions of the given example, we have to compute 12 times to construct the tree. However, compared with tree, we just compute 8 times when directed graph is used to describe the executions of the example.

- Based on the directed graph, we do not need to repeatedly compute the same execution behaviours of application in the sequentialization process, instead, we can construct a cycle in directed graph to represent the same execution behaviours. E.g., as shown Fig. 4.5,

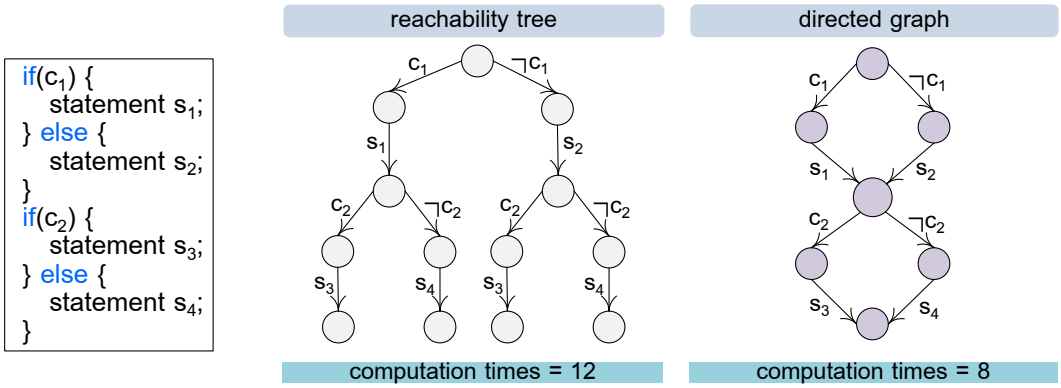



Figure 4.4: An example for tree and directed graph (branch)

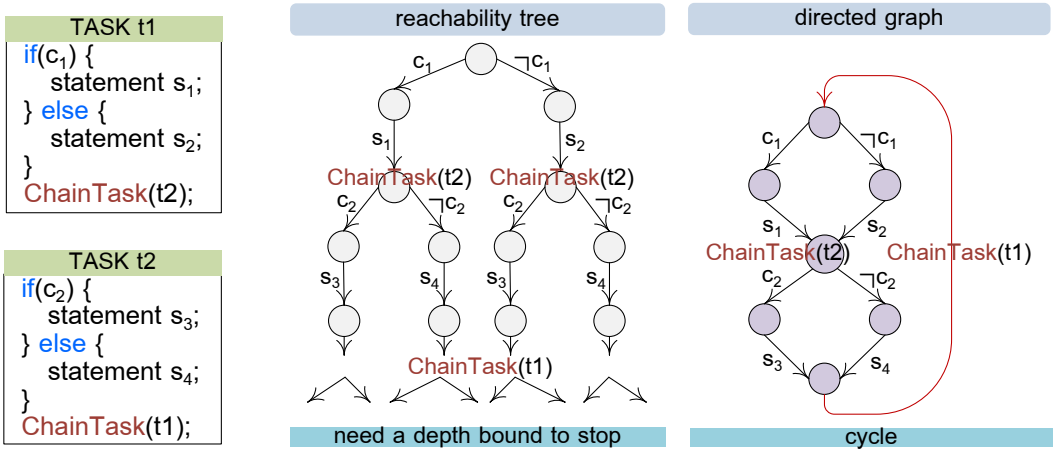



Figure 4.5: An example for tree and directed graph (loop)

there are two tasks $t1$ and $t2$ that will activate each other using API $ChainTask(tk)$. The example holds an infinite execution sequence $t1, t2, \cdots, t1, t2$. If we use tree to describe the executions of the example, we not only need a depth bound to stop the process of constructing tree, but also will spend a lot of computation times to construct the tree under the set depth bound. However, compared with tree, in directed graph we can use a cycle to represent the the same execution behaviours.

Furthermore, based on the sequentialization process of our approach, the existing model checker such as `spin` and `cbmc` can efficiently verify OSEK/VDX applications, because existing model checker just check a sequential model instead of a concurrent model.

# Chapter 5

# Spin-based Checking Approach

As to make model checking more accurate in checking OSEK/VDX applications, in this chapter we will investigate an approach based on the well-known Spin model checker. The key idea of the approach is that a combination model of application model and OSEK/VDX OS model is used to precisely simulate the executions of OSEK/VDX application in real OSEK/VDX OS, and then employ Spin to verify the combination model.

## 5.1 Overview of Spin-based Checking Approach

To accurately check OSEK/VDX applications using Spin model checker, the key work is how to construct a checking model. Based on the execution characteristics of OSEK/VDX applications, we have found that, (*i*) the running task within the application is explicitly determined by OSEK/VDX scheduler according to the ready queue and task configuration data; (*ii*) the APIs invoked from tasks will haphazardly change the scheduling of tasks. As shown in Fig. 5.1, in our approach, we firstly construct a combination model of application model and OSEK/VDX OS model with Promela language to precisely simulate the executions of the OSEK/VDX applications. In the combination model, all of the tasks within application are designed as processes, and the OS model that conforms to OSEK/VDX specification as a process is used to determine the running task and respond to the APIs invoked from tasks. Then, Spin model checker is employed to verify the constructed combination model for obtaining the checking results.

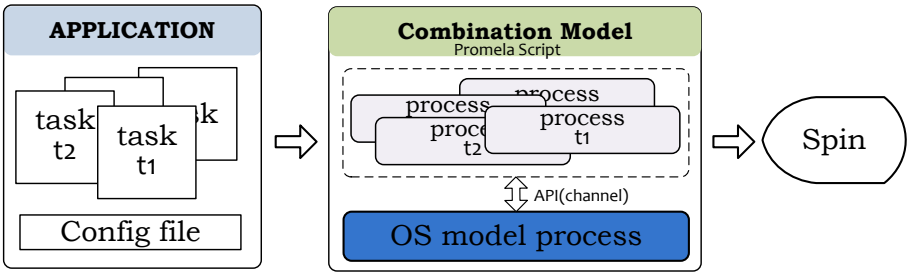



Figure 5.1: The structure of Spin-based checking approach

## 5.2 Combination Model

In our approach, a combination model of application model and OSEK/VDX OS model is established to precisely simulate the executions of the OSEK/VDX applications in real OSEK/VDX OS. The definition of the combination model is as follows,

**Definition 6** *The combination model is a tuple* $\Psi = (\mathcal{OS}, App)$*, where* $\mathcal{OS}$ *is the OSEK/VDX OS model,* $App$ *is the application model.*

In the combination model $\Psi$, $\mathcal{OS}$ is used to conduct the executions of application and respond to the APIs invoked from tasks within application. $App$ is a set consisted of two elements $App$=$\{\Delta, T\}$, where $\Delta$ is the configuration file of application, $T$=$\{t_1, t_2, \cdots\}$ is the finite set of tasks declared in application. To precisely simulate the executions of application, in the combination model $\Psi$, each element $t \in T$ and $\mathcal{OS}$ are design as `process` in `promela` script, and application model $App$ and OS model $\mathcal{OS}$ are synchronously executed via APIs. The execution characteristics of combination model are stated in the following.

When an application runs on the OSEK/VDX OS, the head task in the ready queue will be dispatched to run if the run-unit is idle. The other tasks in the *ready* state, *suspended* state and *waiting* state will not be run until the running chance is given by scheduler. Thus, the first execution characteristic of combination model is as follows.

- *a task* $t \in T$ *can be run iff its ID equals to the running task ID that is computed by OS model* $\mathcal{OS}$*, and the remanent tasks* $A' = T \backslash \{t\}$ *are restrained to execute.*

Once an API is invoked by the application, the OSEK/VDX OS will be loaded to run for responding to the invoked API (at this moment, the executions of the application will be preempted by OSEK/VDX OS). When OSEK/VDX OS has already completed its executions, the

run-unit is released, and then the application will be continued again. According to the described execution characteristics, the following three execution characteristics are held by combination model for simulating the interactive executions between OS model $\mathcal{OS}$ and application model *App*.

- *application model App and OS model $\mathcal{OS}$ are synchronously executed via APIs.*
- *when an API is invoked by running task t, the task t will stop its execution to wait for the executions of OS model $\mathcal{OS}$.*
- *once OS model $\mathcal{OS}$ receives an invoked API from App, OS model $\mathcal{OS}$ will be executed for responding to the invoked API and computing the running task ID. If OS model has completed its executions, application model App will be executed from the stopped point, and then OS model $\mathcal{OS}$ waits for the next API from application model.*

## 5.3 Promela Model of Combination Model

According to the combination model, we can easily simulate the executions of an OSEK/-VDX application. As to conveniently use Spin to check OSEK/VDX applications based on the combination model, we have constructed an OSEK/VDX OS model $\mathcal{OS}$ using `promela` language according to the OSEK/VDX specification. Furthermore, the constructed OS model provides two interface functions for easily constructing the application model. The OS model $\mathcal{OS}$ and the provided interface functions are stated in the following.

### 5.3.1 OS Model

As shown in Fig.5.2, we have developed an OS model based on the OSEK/VDX OS model presented in paper [32], which is a combination of scheduler model, event process model and shared resource process model. The definition of OS model $\mathcal{OS}$ is as follows,

**Definition 7** *The OS model is a tuple $\mathcal{OS} = (S, s_0, D, F, \Sigma)$, where $S$ is the finite set of states ($s_0 \in S$ is the initial state), $D$=\{runTask, readyQueue, suspendList, waitList, evtBitArray, resAccessList\} is the set of data structures, $F$ is the set of functions, $\Sigma \subseteq S \times F \times S$ is the set of transition relations.*

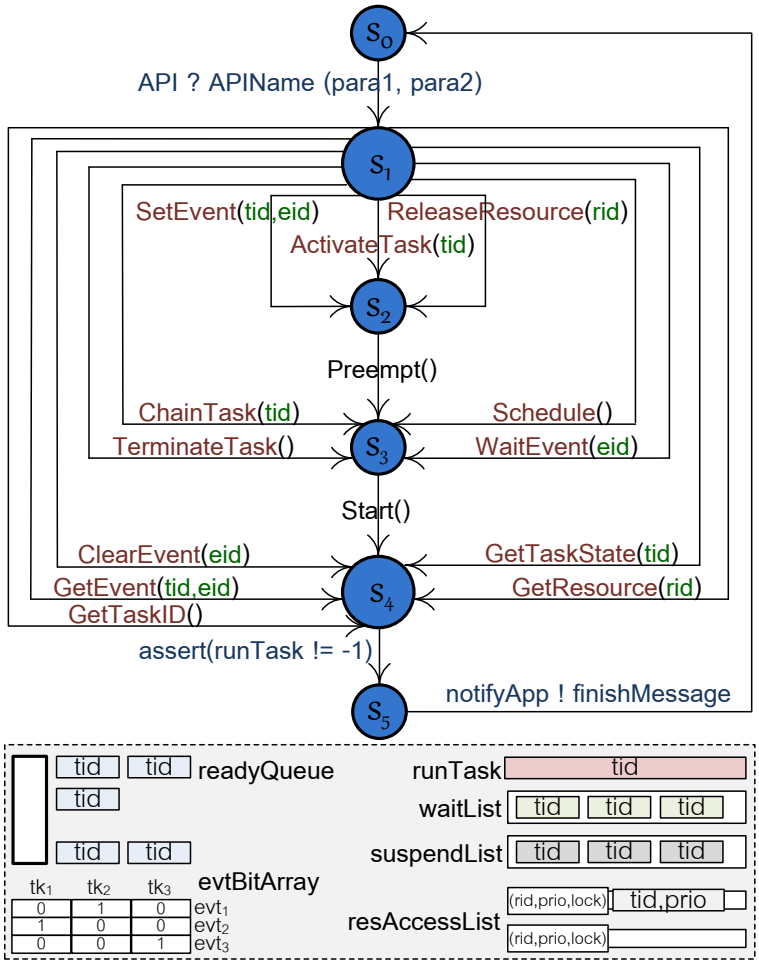



Figure 5.2: OSEK/VDX OS model

In the OS model, $D$ is the set of data structures used to record the states of tasks, synchronization events and shared resources. Where, *runTask* is a variable used to store the running task *tid* ($tid \in \mathbb{N}$ is the identifier of tasks). In OSEK/VDX applications, several tasks can share a same priority in the OSEK/VDX OS. The *readyQueue* is composed of queues with different priorities used to store the *tid* of *ready* tasks. The data structures *suspendList* and *waitList* are used to store the *tid* of tasks in the *suspended* state and *waiting* state, respectively. *evtBitArray* is a matrix used to store the event states of extended tasks ($eid \in \mathbb{N}$ is the identifier of events). *resAccessList* is a set of lists used to indicate the state of resources accessed by tasks ($rid \in \mathbb{N}$ is the identifier of resources). In the function set $F$, `API?APIName(para1,para2)` and `notifyApp!finishMessage` are the synchronization functions used to simulate APIs, the

```
mtype {TerminateTask, ActivateTask,  ChainTask,
       Schedule, SetEvent, GetEvent, ClearEvent,
       WaitEvent, GetResource, ReleaseResource,
       GetTaskState, GetTaskID
};
mtype APIName;
int   para1, para2;
bool  finishMessage;
chan  API = [0] of {mtype, int, int};
chan  notifyApp = [0] of {bool};
```

Figure 5.3: The API functions

```
inline waitForRun(_tid){
  (runTask == _tid);
}
```

Figure 5.4: The interface function `waitForRun`()

```
inline taskAPI(_tid,_APIName,_para1,_para2){
  API ! _APIName(_para1,_para2);
  notifyAPP ? finishMessage;
  (_tid == runTask);
}
```

Figure 5.5: The interface function `taskAPI`()

`promela` scripts for these two functions are shown in Fig. 5.3. Here, `API ? APIName(para1, para2)` is used to receive the invoked APIs from application model (`APIName` is the name of invoked API, `para1` and `para2` are the parameters in APIs). `notifyApp!finishMessage` is used to notify the application model that OS model has already completed its executions. In addition, the assertion `assert`($runTask != -1$) is used to terminate the checking process if there is no running task (where, "–1" represents that running task is idle). The other functions in $F$ such as `ChainTask(tid)` and `TerminateTask`() are the standardized functions defined in OSEK/VDX specification, used to operate the system data $D$ according to the invoked APIs.

### 5.3.2 Interface Functions of OS Model

To conveniently use the constructed OS model to check OSEK/VDX applications, we design two interfaces functions for the application model. The first interface function `waitForRun`()

Source file (.cpp)

```
int buffer = 0;

TASK t1(){
   ActivateTask(t2);
   ActivateTask(t3);
   TerminateTask();
}

TASK t2(){
   buffer++;
   TerminateTask();
}

TASK t3(){
   buffer--;
   TerminateTask();
}
```

Configuration file (.oil)

```
TASK t1{
   TYPE=BASIC;
   SCHEDULE=FULL;
   PRIORITY=5;
   AUTOSTART=TRUE;
 };
TASK t2{
   TYPE=BASIC;
   SCHEDULE=NON;
   PRIORITY=6;
   AUTOSTART=FALSE;
 };
TASK t3{
   TYPE=BASIC;
   SCHEDULE=NON;
   PRIORITY=4;
   AUTOSTART=FALSE;
 };
```

Figure 5.6: Motivating example for Spin-based checking approach

shown in Fig. 5.4 is used to restrain the executions of the tasks whose *tid* are not equal to *runTask*. The second interface function `taskAPI`() shown in Fig. 5.5, which can be invoked by tasks, is used to simulate the behaviors of APIs, in which `API!_APIName(_para1,_para2)` and `notifyAPP?finishMessage` are used to implement the interactive executions between OS model and tasks, (`_tid` == *runTask*) is employed to simulate the context switch of tasks caused by the invoked API (the parameter `_tid` is the host task ID).

### 5.3.3 Example

In this part, we will use an example to show how to use Spin model checker to accurately verify OSEK/VDX applications based on the constructed OSEK/VDX OS model.

As shown in Fig. 5.6, the application holds three tasks, $t_1$, $t_2$ and $t_3$. In the application, only the attribute `AUTOSTART` of $t_1$ is set to be `TRUE`, $t_1$ thus is firstly moved to *running* state by scheduler and then $t_1$ is executed. As shown in Fig. 5.7, when the API $ActivateTask(t_2)$ is invoked by $t_1$, scheduler will be loaded to respond to the API. At this moment, the running task $t_1$ will be preempted by $t_2$, since the priority of $t_2$ is higher than $t_1$ and the attribute `SCHEDUL` of $t_1$ is set to be `FULL`. Currently, task $t_2$ gets run-unit to run, and goes to *suspended* state when the API $TerminateTask()$ is invoked. When $t_2$ is terminated, $t_1$ will be moved to *running* state

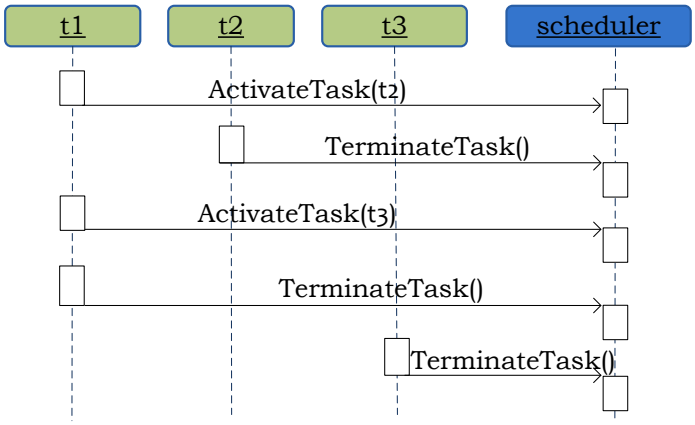



Figure 5.7: The executions of motivating example

again and continue its executions from preempted point. Then, $t_3$ is activated by $t_1$, and will be run when the API $TerminateTask()$ is invoked by $t_1$ ($t_1$ cannot be preempted by $t_3$, because the priority of $t_3$ is lesser than $t_1$). Finally, task $t_3$ is executed when task $t_1$ terminates itself using API $TerminateTask()$.

Based on the constructed OS model and provided interface functions, we can easily construct a combination model to simulate the executions of the motivating example shown in Fig. 5.6. The `promela` model of the motivating example has been presented in Fig. 5.8. Note that, for the configuration file, the constructed OS model uses an `inline` function `configData()` to get the configuration data of tasks, which is shown in Fig. 5.9.

## 5.4 Supported Checking Properties

In our approach, we can use a combination model to precisely simulate the executions of an OSEK/VDX application in real OSEK/VDX OS. Moreover, based on the combination model, Spin can accurately verify the application in the verification process. In this part, we will talk about what kinds of given properties can be checked by our approach in the practical checking process.

**Variable property**: In the practical checking process, we usually want to check whether the executions of an application have already reached a specified state via assertion statements. Based on the combination model, all of the possible executions of application are taken into

```
#include "OSmodel.h"
proctype t1() {
    start:
    waitForRun(t1.tid);
        taskAPI(t1.tid, ActivateTask, t2.tid, -1);
        taskAPI(t1.tid, ActivateTask, t3.tid, -1);
        taskAPI(t1.tid, TerminateTask, -1, -1);
    goto start;
}
proctype t2() {
    start:
    waitForRun(t2.tid);
        buffer=buffer+1;
        taskAPI(t2.tid, TerminateTask, -1, -1);
    goto start;
}
proctype t3() {
    start:
    waitForRun(t3.tid);
        buffer=buffer-1;
        taskAPI(t3.tid, TerminateTask, -1, -1);
    goto start;
}
init {
    run OSModel();  run t1();  run t2(); run t3();
}
```

Figure 5.8: The `promela` model for the motivating example

```
inline configData(){
  task[1].tid=1;
  task[1].priority=5;
  task[1].schedule=true;
  task[1].autostart=true;
  ...
}
```

Figure 5.9: The `inline` function for configuration data

account, and Spin model checker supports the assertions. Thus, our approach can be used to check variable property using assertion statements.

**LTL property**: In addition to assertions, the given property which holds temporal operators is frequently used to check an application in the practical checking process. For instance, we want to check whether the value of a variable will be changed to be zero in the future. Since Spin model checker can accept the given property specified in Linear Temporal Logic (LTL),

our approach can be used to check the LTL property.

**API property**: The API is also an interesting checking point for the OSEK/VDX applications, since APIs perform an important part in the interaction between application and OSEK/VDX OS. In the checking process, we usually want to check whether an API will be invoked by tasks. In our approach, the API is represented as a set {`APIName`,`para1`,`para2`} of variables in `promela`. Therefore, our approach can check the API property.

**OS data property**: When an application runs on the OSEK/VDX OS, it is difficult to judge the execution situations of the application, since the executions of OSEK/VDX applications are conducted by scheduler, and tasks within application can invoke APIs to synchronously execute and access shared resources. As to clearly detect the execution situations of an application, the states of tasks, events and shared resources are often considered as a checking point. To check this type of property (which is named as OS data property in our paper), the data in data structure $D$ of OS model can be accessed by the given property. E.g., we can use the LTL property shown in formula (5.1) to check whether the task *tid* will be run after $ActivateTask(tid)$ is invoked.

$$\begin{aligned} <> ((\texttt{APIName} == ActivateTask \ \&\& \ \texttt{para1} == tid) \\ \&\& \ \texttt{X}(runTask == tid)) \end{aligned} \tag{5.1}$$

**Mutual exclusion property**: Furthermore, the checking process on mutual exclusion property also will be carried out in the practical checking process, since tasks within application can enter a critical section for accessing a shared resource using APIs $GetResource(rid)$ and $ReleaseResource(rid)$. Informally, mutual exclusion contains of two properties, one is *exclusiveness*, and the other is *liveness*. In our approach, the task *tid*s of accessing shared resources are recorded in the shared resource lists $resAccessList$ within the OS model. Thus, our approach can be used to check these two properties. For instance, we can use the LTL properties shown in formulae (5.2) and (5.3) to check the *exclusiveness* property and *liveness* property respectively, where we suppose that task $tk1$ and task $tk2$ will access the same shared resource *rid*. In the formulae (5.2) and (5.3), `IN` represents matching task *tid* in the corresponding shared resource list, $n$ is the number of tasks defined in the application.

$$\begin{aligned} !<> (tk1.tid \ \texttt{IN} \ resAccessList[rid].list[0:n] \ \&\& \\ tk2.tid \ \texttt{IN} \ resAccessList[rid].list[0:n]) \end{aligned} \tag{5.2}$$

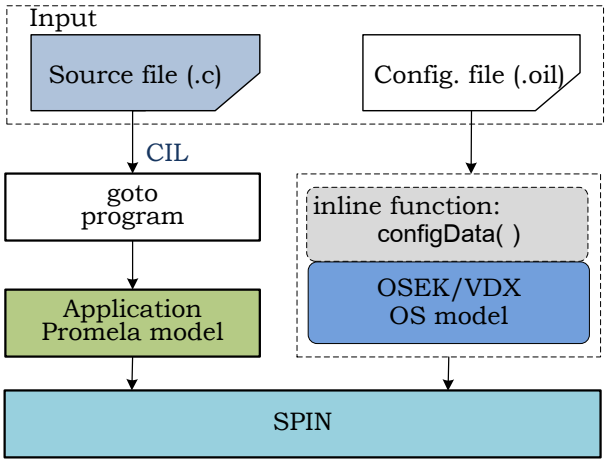



Figure 5.10: The structure of `osek2spin`

$$<>(tk1.tid \texttt{ IN } resAccessList[rid].list[0:n]) \tag{5.3}$$

## 5.5 Implementation

To easily use our Spin-based checking approach to verify an OSEK/VDX application, we develop a tool named `osek2spin`[1] to automatically translate the OSEK/VDX application into `promela` model based on the visual studio 2010. Currently, `osek2spin` supports the C programming language without `pointer`, `struct` and function calls as input language. The key processes of `osek2spin` are shown in Fig. 5.10. In the fist step, the task behaviours described in C programming language are interpreted into `goto` programs based on the C intermediate language (CIL) [25], where complex structures in C programming language such as `for` loop, `while` loop and `switch` are replaced with `goto` statements. In the second step, the obtained `goto` program is translated into the corresponding `promela` model. The data in the configuration will also be automatically passed to the `inline` function `configData()` of OSEK/VDX OS model.

[1] `http://www.jaist.ac.jp/~s1220209/osek-spin.htm`

## 5.6 Advantage and Disadvantage

In the Spin-based checking approach, the checking model is a combination model, consisted of two models, one is application model, and the other is OSEK/VDX OS model. Where, the application model is used to simulate the executions of tasks within application (tasks are designed as `process`es), the OSEK/VDX OS model as a `process` is used to respond to the APIs invoked from tasks and determine the running task. Based on the combination model, the practical execution behaviours of OSEK/VDX application are accurately simulated in the verification process. Thus, the approach is an accurate checking method for OSEK/VDX applications.

However, in the approach, the behaviour of OSEK/VDX OS model will be explored in the verification stage, because the OSEK/VDX OS model is inside the checking model. When an application invokes an API, the OSEK/VDX OS model within the combination model will be loaded to respond to the API. Sequentially, the states of OSEK/VDX OS model will be explored by Spin in the verification stage. If an application invokes a lot of APIs, a large number of states of OSEK/VDX OS model will be explored in the verification stage. The scalability of the approach is limited, since too many details of OSEK/VDX OS model are explored in the verification process, especially the state space explosion will happen if a large number of APIs are invoked by application.

# Chapter 6

# SMT-based Bounded Model Checking Approach

According to the disadvantage of Spin-based checking approach, we find that the approach cannot be used to verify the large-scale OSEK/VDX applications which hold a lot of tasks and APIs, since too many details of OSEK/VDX OS model will be explored in the verification stage. In order to avoid OSEK/VDX OS model to be explored in the verification stage and intend to handle the complex OSEK/VDX applications which hold a large number of states, in this chapter we will show a new approach based on the SMT-based bounded model checking (BMC). In the approach, we develop a constructing model algorithm named execution path generator (EPG) in order to successfully apply BMC to verify OSEK/VDX applications. Particularly, as to put the OSEK/VDX OS model outside the checking model, the OSEK/VDX OS model is embedded in EPG (constructing model algorithm flat) to respond to the APIs invoked from tasks and determine running task in the process of constructing transition system.

## 6.1 Overview of SMT-based Bounded Model Checking Approach

The main processes of SMT-based bounded model checking approach are shown in Fig. 6.1. As to easily check an application, in the first step we use control flow graphs (CFGs) to describe the

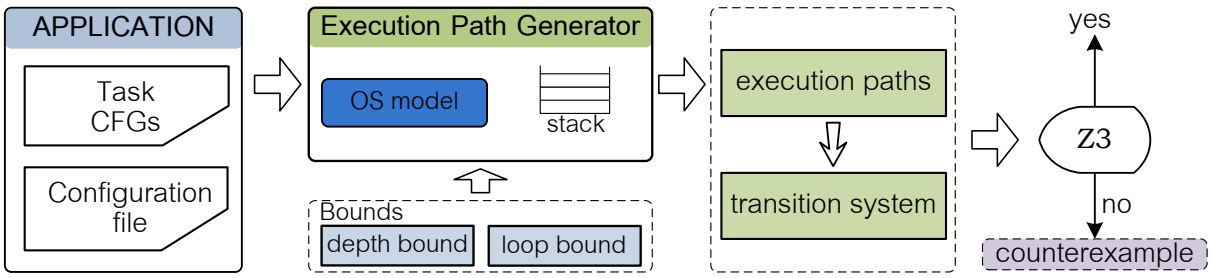



Figure 6.1: The structure of SMT-based bounded model checking approach

behaviors of tasks within the application. Since the APIs invoked from tasks will haphazardly change the scheduling of tasks, e.g., the different APIs locating at different branches will lead to different task execution sequences and the context switch of tasks may happen when an API is invoked by running task, in the second step we develop an execution path generator (EPG) as an intermediate simulator to generate all of the possible execution paths for the target application. In particular, the OSEK/VDX OS model is embedded in the EPG to respond to invoked APIs and determine the running task in the process of constructing execution paths. In the third step, the transition system of the target application will be established based on the generated execution paths. Finally, the SMT solver Z3 [35] is employed to perform the verification for judging whether the constructed transition system satisfies a given property or not, if not, our approach will return a counterexample. Note that, the behaviors of OSEK/VDX OS model will not be explored in the verification stage, since the OS model is embedded in the EPG (constructing model algorithm flat) to respond to invoked APIs and conduct the executions of tasks. In addition, like other bounded model checkers such as `cbmc` [16], the depth bound and loop bound are also supported by our approach.

## 6.2 CFG of Task

To easily check an application, in our approach we use control flow graphs (CFGs) to describe the behaviors of tasks within the application. The definition of CFG of task is as follows,

**Definition 8** *The CFG of a task is a tuple* $\Omega^{tid} = (N^{tid}, n_0^{tid}, n_e^{tid}, \Sigma^{tid}, R^{tid})$.

Source file (.cpp)

```
int x, y;

TASK t1(){
   if(!(x < 0))
      ActivateTask(t2);
   TerminateTask();
}

TASK t2(){
   if(y > x)
      x=y;
   TerminateTask();
}
```

Configuration file (.oil)

```
TASK t1{
   TYPE=BASIC;
   SCHEDULE=FULL;
   PRIORITY=5;
   AUTOSTART=TRUE;
};

TASK t2{
   TYPE=BASIC;
   SCHEDULE=NON;
   PRIORITY=6;
   AUTOSTART=FALSE;
};
```

Figure 6.2: The motivating example for EPG

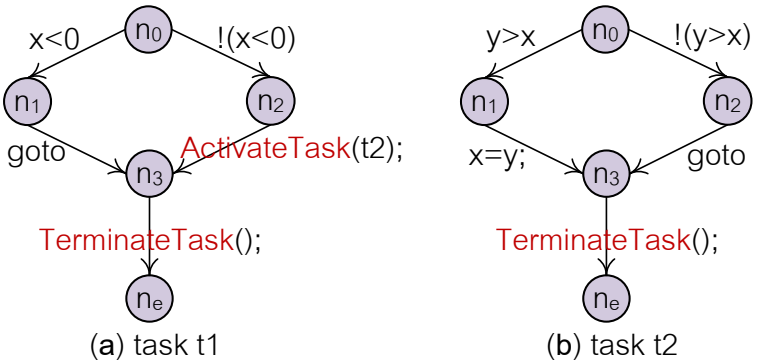



Figure 6.3: The CFGs of tasks

Where, $tid$ is the identifier of tasks. $N^{tid}$ is the set of locations, $n_0^{tid} \in N$ is the start location, $n_e^{tid} \in N$ is the end location. $\Sigma^{tid}$ is the set of statements of task $tid$, the expression of a statement $\alpha \in \Sigma$ is as follows:

$$\alpha ::= condition \,|\, assignment \,|\, goto \,|\, assertion \,|\, \texttt{API}$$

$R \subseteq N^{tid} \times \Sigma^{tid} \times N^{tid}$ is the set of directed edges labelled by task statements. E.g, for the example shown in Fig. 6.2, we can use the CFGs shown in Fig. 6.3 to describe the behaviours of tasks $t1$ and $t2$.

## 6.3 OSEK/VDX OS Model

In OSEK/VDX OS standard, as to let developers easily and confidently implement an OSEK/VDX OS, all of the functions for responding to APIs are specifically stated in the OSEK/VDX OS specification. In our approach, based on the stated functions, as shown in Fig. 6.4, we construct an abstract OS model which is a combination model of scheduler module, synchronization event process module and shared resource process module in EPG to respond to the APIs invoked from application and conduct the executions of tasks within application. The definition of abstract OS model is as follows,

**Definition 9** *The abstract OS model is a tuple* $\mathsf{OS} = (N, n_0, n_\varsigma, F, R, D)$*, where* $N$ *is the set of nodes,* $n_0$ *is the start node,* $n_\varsigma$ *is the end node.* $F$ *is the set of APIs responding functions.* $R \subseteq N \times F \times N$ *is the set of transitions.* $D$=$\{runTask,\ readyQueue,\ suspendList,\ waitList,\ evtBitArray,\ resAccessList\}$ *is the set of data structures.*

In the the set of data structures $D$, $runTask$ is a variable used to store the *tid* of *running* task (*tid* is the identifier of tasks). In the OSEK/VDX OS several tasks can share a same priority, the *readyQueue* is composed of queues with different priorities used to store the *tid*s of tasks in the *ready* state. The data structures *suspendList* and *waitList* are used to store the *tid*s of tasks in the *suspended* state and *waiting* state, respectively. In OSEK/VDX application the extended tasks can hold synchronization events, $evtBitArray$ is a matrix used to store the event states of extended tasks (*eid* is the identifier of events). Moreover, tasks within the OSEK/VDX application can access shared resources according to the priority ceiling protocol, $resAccessList$ is composed of lists used to indicate the state of resources accessed by tasks (*rid* is the identifier of resources). In addition, in the constructed OS model, the trace $\varpi$ which starts from node $n_0$ and ends at node $n_\varsigma$ represents the responding behaviors of the OS model for an invoked API.

In order to call the constructed OS model to respond to the invoked APIs and compute running task in the process of generating execution paths, we define two interface functions `StartTask`() and `RespondAPI`(`API`) for the constructed OS model. Where, the interface function `StartTask`() is used to request OS model to dispatch the head task in *readyQueue* to *running* state, the interface function `RespondAPI`(`API`) is used to call the corresponding trace $\varpi$ to respond to the invoked APIs and compute the data within $D$ of OS model. For instance, if the interface function `RespondAPI`(`API`) is called to respond to the invoked API $ActivateTask(t1)$,

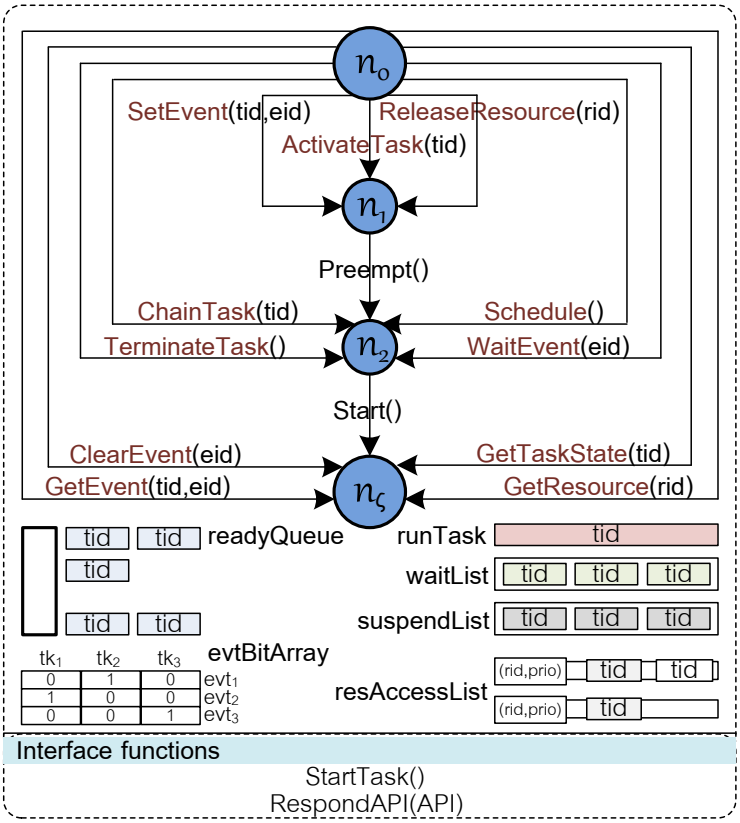



Figure 6.4: OSEK/VDX OS model in EPG

the function $ActivateTask(t1)$ of OS model is firstly called to active task $t1$, and then the function $Preempt()$ is called to judge whether the rescheduling point happens (if the rescheduling point happens, the currently running task will be moved to $readyQueue$, and $runTask$ will be set to null by the $Preempt()$ function). Finally, the function $Start()$ is called to dispatch the head task in $readyQueue$ to run if $runTask$ is null.

## 6.4 EPG and Transition System

The challenge of verifying a target system using BMC is how to construct a transition system to reflect the executions of the target system. Based on the execution characteristics of OSEK/VDX applications, we found that the APIs invoked from tasks will haphazardly change the scheduling of tasks, e.g., the different APIs locating at different branches will lead to different task execution sequences and the context switch of tasks may happen when an API is invoked by running task.

In our approach, as to precisely construct a transition system for an OSEK/VDX application and avoid the behaviors of OS model to be poured in the transition system, we explore all of the execution paths of the application and embed OS model in the algorithm flat to compute running task and respond to the APIs invoked from application. We have developed constructing model algorithm named execution path generator (EPG) to implement this idea.

### 6.4.1 EPG

The key process of EPG is to symbolically execute OSEK/VDX application based on the scheduling of the embedded OSEK/OSEK OS model. EPG consists of two modules, one is execution path constructor, and the other is OS model. The execution path constructor is used to construct the execution paths according to the executions of currently running task. The OS model is employed by the EPG to respond to invoked APIs and compute the running task when execution path constructor meets an API in the process of constructing execution paths. The main processes of EPG are as follow,

1. Execute the interface function `StartTask`() of OS model to compute the *running* task.
2. Construct an execution path along the trace of *running* task CFG, and map the explored statements in the execution path.
3. If meet branches, select one branch to continue, and push the other branches and the current data in $D$ of OS model into `stack`.
4. If the explored statement is an API, execute the interface function `RespondAPI(API)` of OS model to respond to the API and compute the data in $D$ of OS model.
5. If *running* task is null, pop an element from `stack`. Then, map the popped OS data to $D$ of OS model, and construct the next execution path according to the popped branch position.
6. Repeat 2, 3, 4 and 5 until `stack` is empty.

The details of EPG are shown in Algorithm 1. In Algorithm 1, an element of stack is a tuple $elem$=($pcs$, $osd$, $i$, $(n, n')$). Where, $pcs$ which is an array is used to record the current positions

**Algorithm 1** : **Execution Paths Generator**

**Input:** task CFGs $\Omega^1$,$\Omega^2$,$\cdots$, and configuration file
**Output:** Execution paths $\pi_1, \pi_2, \cdots$

1: $pcs := [n_0^1, \ldots, n_0^m]$, where $m$ is the number of tasks
2: $i$:=0, where $i$ is the index of state of execution path
3: $j$:=1, where $j$ is the index of execution paths
4: initialize a *stack*
5: initialize $D$ within OS model according to application config. file
6: call the interface function of OS model `StartTask()`
7: **while** $runTask$ of $D$ != $null$ **do**
8: $tid := runTask$ of $D$ within OS model
9: $\Delta := \{(n, n') \in \Omega^{tid} | n = pcs[tid]\}$
10: $(n, n')$:= one of the element of $\Delta$
11: $\Delta$:=$\Delta \setminus \{(n, n')\}$
12: **if** $|\Delta| > 0$ **then**
13: $D \rightarrow osd$, the operator "$\rightarrow$" represents mapping the data within $D$ into $osd$
14: **for all** $(n, n') \in \Delta$ **do**
15: $elem$:=$(pcs, osd, i, (n, n'))$, $stack.push(elem)$
16: **end for**
17: **end if**
18: **if** the statement $\alpha$ mapped in edge $(n, n')$ is an API **then**
19: **if** $\alpha$ is $TerminateTask()$ or $ChainTask(tid')$ **then**
20: $pcs[tid']$:=the start node $n_0$ of task $tid'$
21: **end if**
22: call the interface function of OS model `RespondAPI(`$a$`)` to respond to the invoked API and compute $D$
23: **end if**
24: create a new state $s_{i+1}$ for the execution path $\pi_j$, and map the task statement $\alpha$ in the edge $\langle s_i, s_{i+1} \rangle$
25: $i$++
26: update $pcs[tid]$ with target node $n'$ of edge $(n, n')$
27: **if** $runTask = null$ **then**
28: $output(\pi_j)$
29: **if** $stack.empty() = true$ **then**
30: break
31: **end if**
32: $(pcs, osd, i, (n, n'))$:=$stack.pop()$
33: $osd \rightarrow D$, the operator "$\rightarrow$" represents mapping the values within $osd$ into $D$ of OS model
34: $\pi_{sub}$:=$GetSubpath(\pi_j, i)$
35: $j$++
36: $\pi_j$ :=$\pi_{sub}$
37: **end if**
38: **end while**
39: **return**

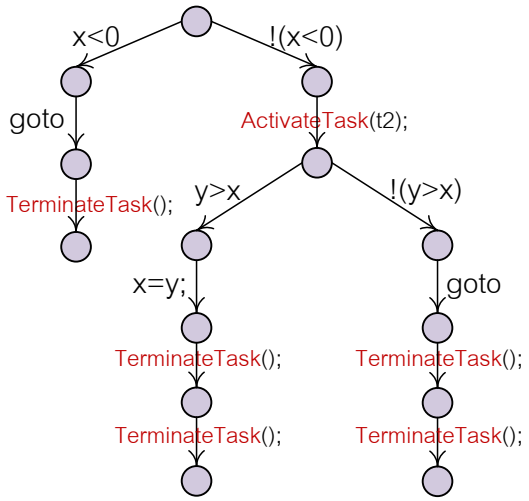



Figure 6.5: The execution paths for motivating example

of task CFGs, *osd* which is the set of values is used to record the data within $D$, $i$ which is variable is used to record the position of branches, $(n, n')$ is one of the branches. Since stack is used to construct the execution paths, the execution path $\pi_j$ and the next execution path $\pi_{j+1}$ will hold the same sub-path which starts from initial state and ends at the position of branch popped from stack. The function $GetSubpath(\pi_j, i)$ is used to extract the same sub-path from the execution path $\pi_j$ for constructing the next execution path $\pi_{j+1}$. $\Delta$ which is a set is used to store the edges whose previous node $n$ is equal to current position of *running* task.

Based on the EPG, all of the execution paths with respect to the target application can be generated. The definition of generated execution path $\pi$ is as follows,

**Definition 10** *An execution path* $\pi$ *is the task statement sequence* $\pi = s_0 \xrightarrow{a} s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3 \xrightarrow{a} \cdots$, $s$ *is the state of execution path,* $\alpha \in \bigcup \Sigma^{tid}$ *is the statement of tasks.*

E.g., for the example shown in Fig. 6.2, EPG will generate three execution paths, which have been illustrated in Fig. 6.5.

### 6.4.2 Transition System

When all of the possible execution paths of an OSEK/VDX application are generated by EPG, we then can construct a corresponding transition system for the application. Here, we use function $[\![\pi_j]\!]$ to convert an outputted execution path $\pi_j$ into the conjunctive normal form (CNF)

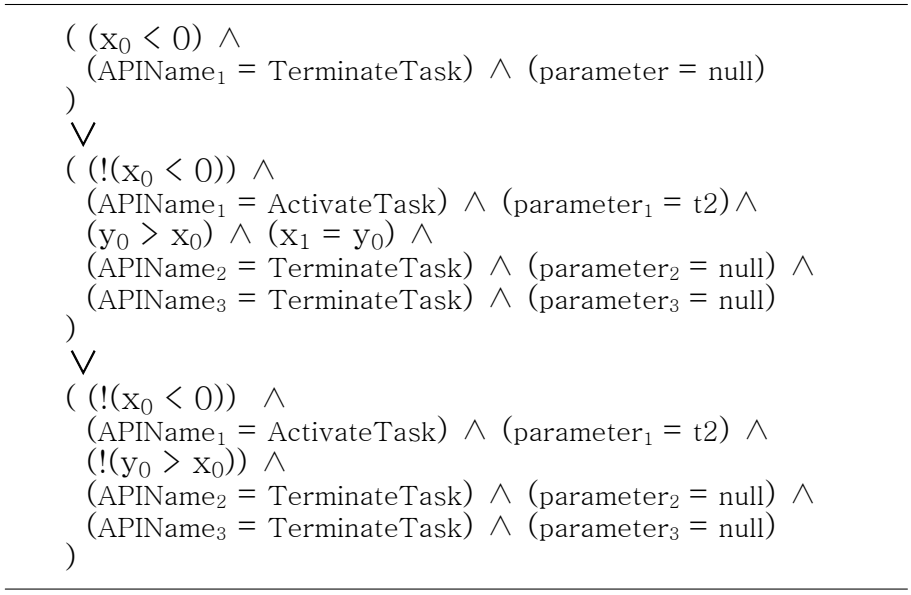



Figure 6.6: The transition system for motivating example

expression in SMT-LIB format. The CNF expression for an execution path $\pi_j$ is defined in the formula (6.1), where $\mathcal{L}(s_i,s_{i+1})$ represents a task statement $\alpha$ mapped in the edge $(s_i,s_{i+1})$ of execution path $\pi_j$. The conversion processes of function $[\![\pi_j]\!]$ are as follow,

- if $\mathcal{L}(n, n')$ is a condition statement, such as "$x > y$", then
  $[\![\mathcal{L}(s, s')]\!] \Rightarrow (x > y)$
- if $\mathcal{L}(n, n')$ is an assignment statement, such as "$x = y + z$", then
  $[\![\mathcal{L}(s, s')]\!] \Rightarrow (x' = y + z)$
- if $\mathcal{L}(n, n')$ is an `API`, such as $ChainTask(tk)$, then
  $[\![\mathcal{L}(s, s')]\!] \Rightarrow ((\texttt{APIName}' = ChainTask) \wedge (\texttt{parameter}' = tid))$, where `APIName` and `parameter` are the variables used to record the name and parameter of invoked API.
- if $\mathcal{L}(n, n')$ is `goto` statement, then
  $[\![\mathcal{L}(s, s')]\!] \Rightarrow$
- if $\mathcal{L}(n, n')$ is a assertion statement, such as "$assert(x > y)$", then
  $[\![\mathcal{L}(s, s')]\!] \Rightarrow (assert(!(x > y))$

According to the formula (6.1), we can obtain the transition system when we translate all of the generated execution paths into CNF expression. The expression of transition system $[\![M]\!]$ is defined in the formula (6.2), where $I(s_0)$ which is the initial function is used to initialize each global variable and local variable declared in the target application, $w$ is the number of execution paths. E.g., the transition system of the motivating example is shown in Fig. 6.6. Finally, we can pass the constructed transition system and a given property to SMT solver for checking whether the transition system holds the given property.

$$[\![\pi_j]\!] := \bigwedge_{i=0}^{|\pi_j|-1} [\![\mathcal{L}(s_i,s_{i+1})]\!] \tag{6.1}$$

$$[\![M]\!] := I(s_0) \wedge (\bigvee_{j=1}^{w} [\![\pi_j]\!]) \tag{6.2}$$

# 6.5 Bounds

## 6.5.1 Depth Bound

In OSEK/VDX applications, tasks can invoke the API $ActivateTask(tid)$ or $ChainTask(tid)$ to activate a suspended task. This activation process will possibly result in an infinite task execution sequence in an application. As shown in Fig. 6.7, task $tk_a$ invokes API $ChainTask(tk_b)$ to terminate itself and activate task $tk_b$. Similarly, task $tk_b$ invokes $ChainTask(tk_a)$ to terminate itself and activate task $tk_a$. We can find that the application holds an infinite task execution sequence $tk_a,\ tk_b,\ tk_a,\ \cdots,\ tk_a,\ tk_b$, which means, the application holds an infinite execution path. For this type of applications, EPG cannot stop itself when we use it to generate execution paths. In order to terminate the executions of EPG, a bound for limiting the depth of infinite execution paths is supported by our approach. In the process of generating execution paths using EPG, when EPG meets the bound, it will stop to construct the current execution path and then go to construct the next execution path.

## 6.5.2 Loop Bound

Like other bounded model checker such as `cbmc`, in our approach, the computation of variables is performed by the back-end solver Z3 rather than EPG. Thus, the break-conditions of loops

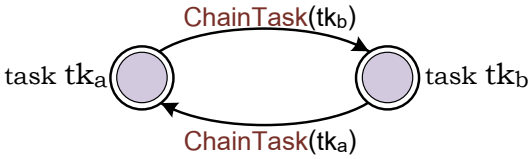



Figure 6.7: Infinite execution path

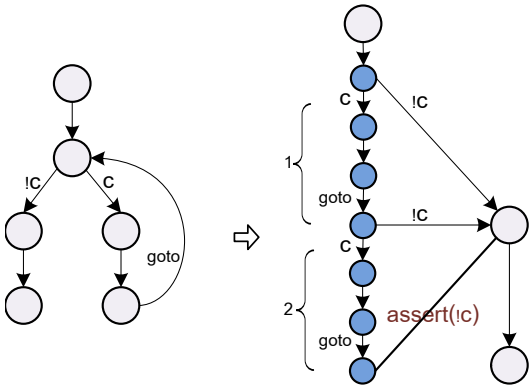



Figure 6.8: Unfolding loop according to loop bound

cannot be determined in the process of constructing execution paths. Although the bound for limiting the depth of execution paths can terminate the executions of EPG when the given application holds loops, a lot of redundant behaviors within loop body will be continuously translated into transition system. In order to eliminate the redundant transitions of loops from transition system, a bound for limiting the execution times of loops is considered in our approach. According to the set bound, loops will be unfolded in the host task CFG before using EPG to generate execution paths. As shown in Fig. 6.8, the loop will be unfolded two times when we set the loop bound to be 2. Especially, as to judge whether a loop has been unfolded enough, the assertion which holds the negative break-condition will be inserted into the end of loop body. In the process of unfolding loops, the Tarjan algorithm [5] is used to find loops from tasks in our approach.

## 6.6 Supported Checking Properties

Based on the EPG and bounds, we can construct a transition system under bounds for OSEK/VDX application. In this part, we will talk about what kinds of given properties can be checked by

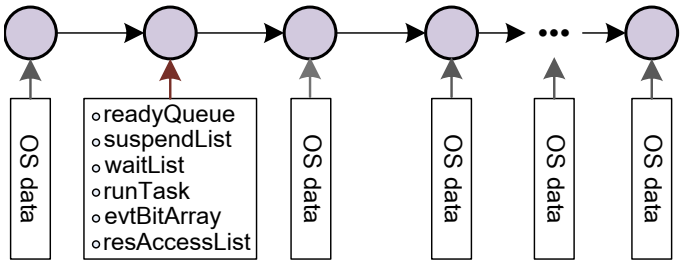



Figure 6.9: Mapping OS data of OS model to nodes of execution path

our approach.

**Variable property**: In the practical checking process, we usually use assertions to check whether the executions of target application have already reached a specified state. Based on the expression of transition system, we can find that all of the executions of application have been translated into the corresponding transition system. Thus, our approach can be used to check variable property using assertion statement. In addition to assertions, the given property which holds temporal operators is frequently used to check an application in the practical checking process. For instance, we want to check whether the value of a variable will be changed to be zero in the future. To check such type of property, the property specified in Linear Temporal Logic (LTL) can be accepted by our approach. The conjunctive expression $\psi$ of transition system and given property $f$ specified in LTL is defined in the formula (6.3), where $k$ is the state number of the longest execution path. The process about how to unfold the given property $f$ specified in LTL has been stated in paper [14].

$$[\![\psi_k]\!] := I(s_0) \wedge \bigvee_{j=1}^{w} ( \bigwedge_{i=0}^{|\pi_j|-1} [\![\mathcal{L}(s_i, s_{i+1})]\!]) \wedge \neg f_k \tag{6.3}$$

**API property**: The API is also an interesting checking property for the OSEK/VDX applications, since APIs perform an important part in the interaction between application and OSEK/VDX OS. In the checking process, we usually want to check whether an API will be invoked by tasks. In our approach, the API statements within tasks are mapped in the execution paths by EPG. Therefore, our approach can be used to check API property.

**OS data property**: When an application runs on the OSEK/VDX OS, it is difficult to judge the execution situations of the application, since the executions of OSEK/VDX applications are conducted by scheduler, and tasks within application can invoke APIs to synchronously execute

and access shared resources. As to clearly detect the execution situations of an application, the states of tasks are often considered as a checking point. To check this type of property (which is named as OS data property in our paper), the transition system should hold the data within $D$ of OS model. However, in the process of constructing transition system, only executions of application are translated into transition system. In order to improve our approach to check OS data property, as shown in Fig.6.9, the OS data of OSEK/VDX OS model is mapped into each state of execution paths by EPG when constructing execution paths. Based on the mapping process, the OS data property can be checked by our approach, e.g., we can check whether the running task will be moved into ready queue after $ActivateTask(tid)$ is invoked.

**Mutual exclusion property**: In addition, the checking process on mutual exclusion property also will be carried out in the practical checking process, since tasks within application can enter a critical section for accessing a shared resource using APIs $GetResource(rid)$ and $ReleaseResou\text{-}rce(rid)$. Informally, mutual exclusion contains two properties, one is $exclusiveness$, the other is $liveness$. In our approach, as to check these two properties, the task $tid$ of accessing shared resources is recorded in $resAccessList$ of OS model, and the recorded data is mapped in the states of execution paths by EPG. For instance, we can use the given properties $f = !\mathrm{F}(tk_1@Res_1 \wedge tk_2@Res_1)$ and $f = \mathrm{F}(tk_1@Res_1)$ to check the $exclusiveness$ property and $liveness$ property of target application respectively, where we suppose that task $tk_1$ and task $tk_2$ will access the same shared resource $Res_1$. In the given properties, @ represents matching task ID in the corresponding shared resource list, F is the $Liveness$ temporal operator in LTL.

## 6.7 Verification of Transition System using Z3

Z3, a high-performance theorem prover, has been widely applied in the SMT-based bounded model checking. In our approach, Z3 as the back-end solver is employed to check whether the constructed transition system holds the given property, if not, to provide a counter-example. There are two strategies for checking constructed transition system using Z3, first one is just invoking Z3 one time to check whole transition system, the second is invoking Z3 many times to check. In our approach, we use second strategy to perform the checking process. When EPG outputs an execution path, we will invoke Z3 to check the transition system which is built on the one execution path, and we stop the checking process when Z3 has found a bug, or has

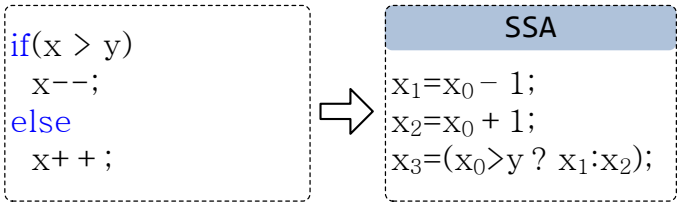



Figure 6.10: Static single assignment (SSA)

systematically checked all of execution paths.

On the face of this strategy, it seems to be naive: (*i*) a lot of iterative transitions will be checked by Z3, since the generated execution paths probably hold many same sub-paths. (*ii*) We need to invoke Z3 several times to check the obtained transition system, which might slow down the checking process. However, there are several advantages that can make this strategy worthwhile. First, if the target application contains errors in the preceding execution paths, the errors will often be found within a shorter time compared with first strategy, since only a small part of the whole transition system is checked in the practical checking process (the experience on the real application can be found in the paper [47]). Second, we do not need to construct all of the execution paths in advance, instead, we only store one execution path in the memory when we check an target application. Third, and most important, this strategy can improve the checking ability of our approach to handle highly complex application, since just the small parts of the target application are checked by our approach in each checking process.

## 6.8 Reduction of Execution Paths

In bounded model checking, static single assignment (SSA) [18] as an efficient and important technique is usually employed to combine the branches in the process of constructing transition system, as shown in Fig. 6.10. In OSEK/VDX applications, since APIs invoked from tasks will dynamically change the scheduling of tasks, e.g., the different APIs locating at different branches will result in different task execution orders and the invoked APIs may lead to the context switch of tasks. Thus, in our EPG technique, we explore all of the possible execution paths to construct the corresponding transition system. However, if the given application holds a lot of branches, our approach will check a large number of execution paths, which will slow

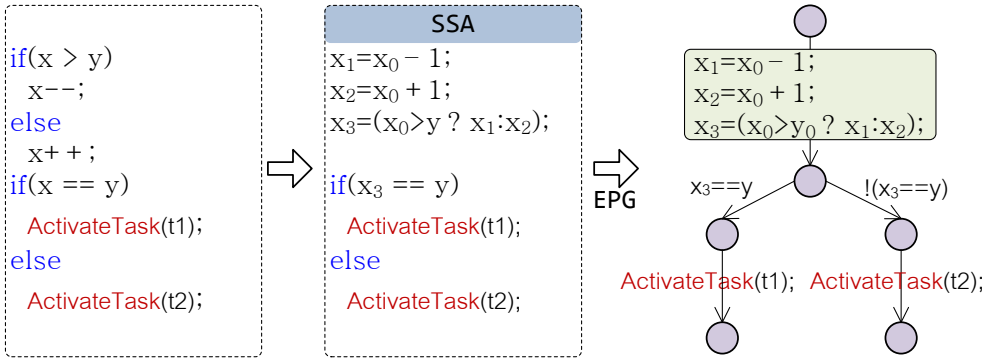



Figure 6.11: Example for reducing execution paths

down the performance of our approach.

In order to improve the performance of our approach, we propose an optimization strategy to combine the branches which do not hold APIs for reducing execution paths based on the SSA technique. In the checking process, we firstly carry out the optimization strategy to combine the branches which do not hold APIs, and then use EPG to generate execution paths. As shown in Fig. 6.11, if the SSA technique is used in the checking process, EPG will generate two execution paths for the example; otherwise, EPG will generate four execution paths for the example.

## 6.9 Implementation

We have implemented a corresponding tool named `osek-bmc`[1] according to the proposed approach. `osek-bmc` consists of five modules, implemented on the Visual Studio 2010 with 9400 lines of C++ code. The structure of `osek-bmc` is shown in Fig. 6.12. In the first module, as to conveniently applied our approach to verify an OSEK/VDX application, we develop a front-end interpreter to interpret the behaviors of tasks within the application into the corresponding CFGs based on the C intermediate language (CIL) [25] (the interpreter for now can accept the main characteristics of C programming language except `pointer`, `struct` and function calls). The second module is to unfold loops within tasks according to the set loop bound. The third module is used to combine the branches without APIs based on the SSA technique. The fourth module is to implement the functionality of EPG. The last module is to invoke SMT solver Z3

[1] `http://www.jaist.ac.jp/~s1220209/osek-bmc.htm`

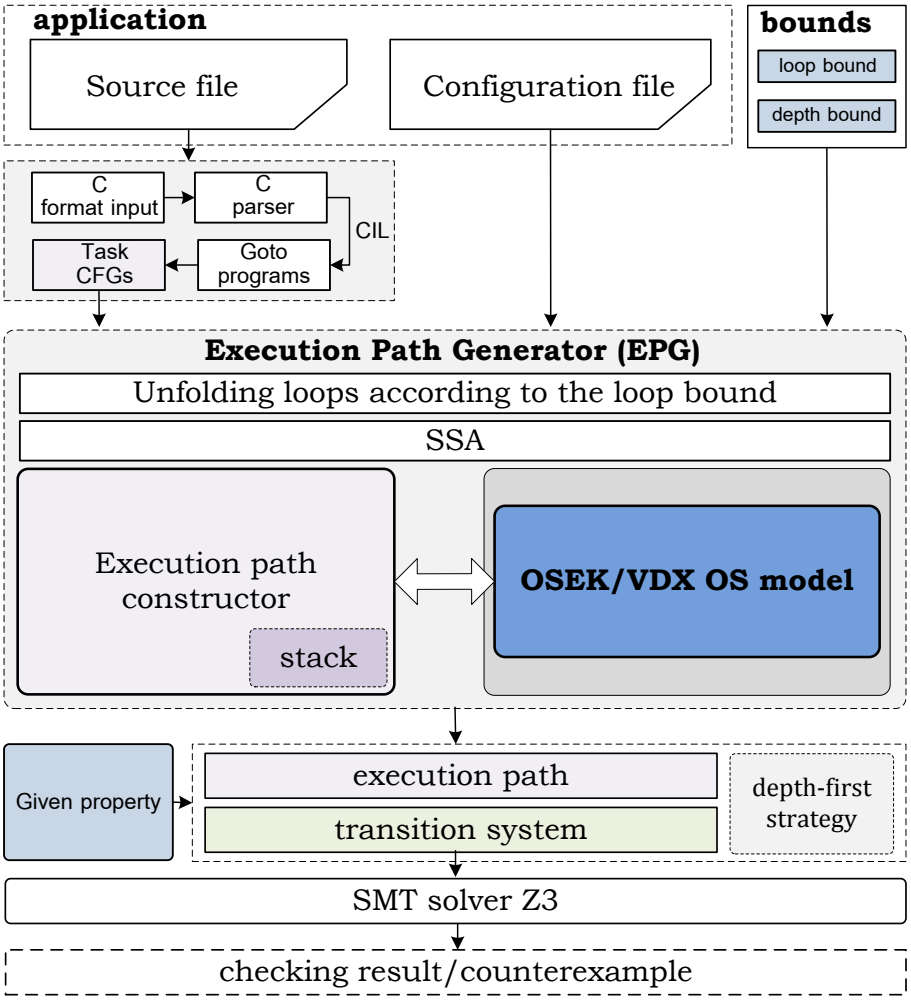



Figure 6.12: The structure of `osek-bmc`

to check whether the constructed transition system satisfies a given property or not, if not, a counterexample will be reported.

Furthermore, in OSEK/VDX applications, tasks often get a value from a sensor. However, it is difficult to determine the value of a sensor in the verification, since the value of sensors is usually in a range. As to conveniently simulate the values of a sensor, a function `random(low,high)` is supported by `osek-bmc` to implement the non-deterministic behaviours.

## 6.10 Advantage and Disadvantage

In our SMT-based checking approach, we present a new method named EPG to accurately and efficiently verify the OSEK/VDX applications which hold a lot of tasks, APIs and states. In the approach, as to avoid OSEK/VDX OS model to be explored in the verification stage, we embed OSEK/VDX OS model in EPG (constructing model algorithm flat) to respond to the APIs invoked from tasks and determine the running task. Moreover, as to handle the complex applications which hold a large number of states, the state-of-art SMT solver Z3 is used to carry out the verification. The approach holds the following advantages,

- EPG: we can construct a transition system without the behaviors of OSEK/VDX OS model for the target OSEK/VDX application based on the generated execution paths, since OSEK/VDX OS model is embedded in EPG to determine the running task and respond to the APIs invoked from tasks.
- SMT-based BMC: it can make our approach more efficient and scalable in checking the complex applications which hold a large number of states, because the state-of-the-art SMT solver Z3 is used to carry out the verification.

However, the approach is not efficient to handle the applications which hold a lot of loops and branches with APIs. This is because, in our EPG technique the transition system of an OSEK/VDX application is constructed based on the execution paths. If the application holds a lot of loops with APIs, it will spend a lot of time exploring execution paths for constructing transition system under the set loop bound, which will slow down the performance of our EPG technique.

# Chapter 7

# Sequentialization-based Checking Approach

According to the disadvantages of EPG technique, we found that the approach will spend a lot of time verifying the OSEK/VDX applications which hold a lot of loops and branches with APIs, since in EPG technique the transition system of application is constructed based on the execution paths. If an application holds many loops with APIs, EPG technique will spend a lot of time exploring execution paths for constructing transition system under the set loop bound, which will slow down the performance of the approach. To efficiently check the applications which hold a lot of loops with APIs, the key work is how to avoid a large number of execution paths to be explored in the verification process.

According to the technique characteristics of explicit state model checking and bounded model checking, we find that these model checking techniques can efficiently check a sequential software even though the sequential software holds a lot of loops. This is because, these model checking techniques will not explore execution paths in the verification stage, i.e., in the explicit state model checking technique sequential software is explicitly executed, and in the bounded model checking technique SSA technique is used to combine branches after the process of unfolding loops. Based on above discussions, if we can efficiently translate OSEK/VDX applications into sequential models, that means, we can efficiently verify OSEK/VDX applications using model checking techniques such as explicit state model checking technique and bounded model checking technique. However, there is a challenge that is how to efficiently translate

Source file (.cpp)

```
int x=0, y=0;

TASK t1(){
   while(x < 2){
      ActivateTask(t2);
      x++;
   }
   TerminateTask();
}

TASK t2(){
   y++;
   TerminateTask();
}
```

Configuration file (.oil)

```
TASK t1
{
   TYPE=BASIC;
   SCHEDULE=FULL;
   PRIORITY=5;
   AUTOSTART=TRUE;
};
TASK t2
{
   TYPE=BASIC;
   SCHEDULE=NON;
   PRIORITY=6;
   AUTOSTART=FALSE;
 };
```

Figure 7.1: The motivating example for DGC

OSEK/VDX applications into the corresponding sequential models.

Actually, we have sequentialized OSEK/VDX applications in our EPG technique. E.g., for the motivating example shown in Fig. 7.1, based on the the execution paths which are generated by EPG, we can construct an execution tree illustrated in Fig. 7.2 to represent the sequential model of the application (where, we set the loop bound to 2). There are two advantages in the execution tree method, $(i)$ the execution tree does not hold the behaviours of OSEK/VDX OS, since the OSEK/VDX OS model is embedded in the constructing algorithm to dispatch tasks and respond to the APIs invoked from tasks; $(ii)$ the constructed execution tree can accurately represent the sequential executions of the OSEK/VDX application, since the execution paths of the execution tree can accurately reflect all of the different task execution orders caused by the APIs locating at different branches. However, there are some disadvantages in the execution tree method, e.g., first, the constructed execution tree holds a lot of the same sub-paths (as shown in Fig. 7.2, the sub-path from ① to ② is held by the both execution path $\pi_2$ and $\pi_3$); second, the method cannot stop its execution if the given application holds loops, because it just symbolically executes the application, the computation on variables are not computed in the process of constructing execution tree. third, and most important, this method is not efficient because in the sequentialization process this method will spend lot of time exploring execution paths.

As to efficiently sequentialize OSEK/VDX applications which hold a lot of branches and same

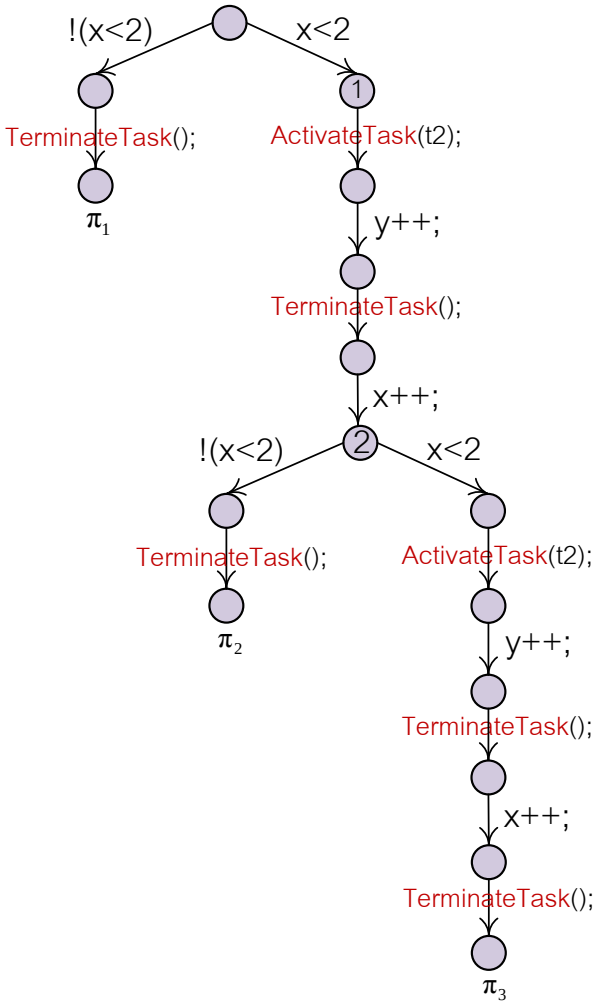



Figure 7.2: The executive tree for motivating example

execution behaviours such as loops into sequential models, we propose a novel approach in this chapter based on the advantages of execution tree. In the approach, we use an extended directed graph instead of execution tree to compute the sequential model of OSEK/VDX application. There are several strengths that can make directed graph more efficient than execution tree to compute the sequential model of OSEK/VDX application, e.g., $(i)$ based on the directed graph, we do not need to explore execution paths when meeting branches in the sequentialization process, instead, we can use the combination states to reduce the computation times; $(i)$ based on the directed graph, we do not need to repeatedly compute the same execution behaviours of application in the sequentialization process, instead, we can construct a cycle in directed graph

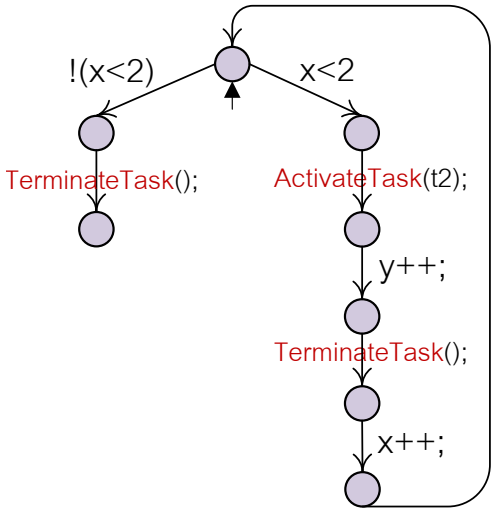



Figure 7.3: The extended directed graph for motivating example

to represent the same execution behaviours.

The key idea of the approach is to symbolically execute the application using extended directed graph and explicitly perform the scheduling behaviours of OSEK/VDX OS in the sequentialization process. E.g., for the motivating example shown in Fig. 7.1, we can use the extended directed graph illustrated in Fig. 7.3 to represent the corresponding sequential model. Based on the constructed execution tree and extended directed graph, we can easily find that the extended directed graph is more efficient than execution tree to represent the sequential model of OSEK/VDX application, since we compute 16 times to construct the execution tree (where, we suppose that one state spends once computation). Compared with execution tree, we just spend 8 times constructing the extended directed graph. Moreover, based on this idea, in our approach we have developed a directed graph constructor (DGC) to efficiently sequentialize OSEK/VDX applications.

## 7.1 Overview of Sequentialization Approach

The structure of our approach is shown in Fig. 7.4. As to automatically translate an OSEK/VDX application into the corresponding sequential program, in the first step we develop a front-end interpreter to interpret the behaviors of tasks within the application into corresponding control

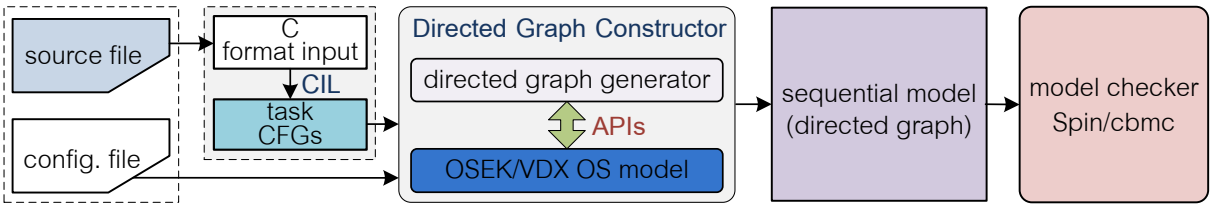



Figure 7.4: The structure of DGC

flow graphs (CFGs) based on the C intermediate language (CIL) [25]. In the second step, the directed graph constructor (DGC) is employed to construct a directed graph for representing the sequential program of OSEK/VDX application. Particularly, the constructed OSEK/VDX OS model shown in SMT-based bounded model checking approach is embedded in the DGC (translation algorithm flat) to respond to the APIs invoked from tasks and determine the running task when the directed graph generator meets an API in the process of constructing directed graph. In the last step, the model checker such as Spin and `cbmc` will be employed to verify the obtained sequential model.

## 7.2 CFG of Task

Like our EPG technique, in our sequentialization-based checking approach, we also use CFG to describe the behaviours of tasks. The definition of CFG of task is as follows,

**Definition 11** *The CFG of a task is a tuple* $\Omega^{tid}=(N^{tid}, n_0^{tid}, n_e^{tid}, \Sigma^{tid}, R^{tid})$.

Where, $tid$ is the identifier of tasks. $N^{tid}$ is the set of locations, $n_0^{tid} \in N$ is the start location, $n_e^{tid} \in N$ is the end location. $\Sigma^{tid}$ is the set of statements of task $tid$, the expression of a statement $\alpha \in \Sigma$ is as follows:

$$\alpha ::= condition \,|\, assignment \,|\, goto \,|\, assertion \,|\, \texttt{API}$$

$R \subseteq N^{tid} \times \Sigma^{tid} \times N^{tid}$ is the set of directed edges labelled by task statements. For example, the CFGs for the motivating example illustrated in Fig. 7.1 are shown in Fig. 7.5.

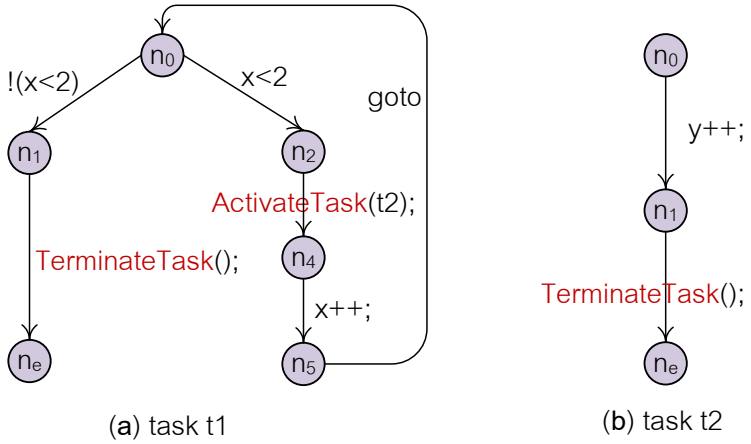



Figure 7.5: The CFGs for motivating example

## 7.3 DGC and Sequential Model

### 7.3.1 DGC

The key idea of the approach is to symbolically execute the application using extended directed graph and explicitly perform the scheduling behaviours of OSEK/VDX OS in the sequentialization process. In the other words, in the sequentialization process, we do not compute the task statements such as condition statements and assignment statements, instead, we symbolically execute application using an extended directed graph under the scheduling of OSEK/VDX OS. The definition of the extended directed graph is as follows,

**Definition 12** *The extended directed graph is a tuple* $\mathcal{G}=(V, v_0, v_e, E)$. $V$ *is the set of nodes, and a node* $v \in V$ *is a tuple* $v = (pcs, osd)$, *where* $pcs=[n^1,...,n^m]$ *is an array used to record the current locations of tasks* $t_1,...,t_m$ ($m$ *is the number of tasks*), *osd which is a set of values used to store the data within* $D$ *of OS model.* $v_0 \in V$ *is the start node, and* $v_e \in V$ *is the end node.* $E \subseteq V \times \bigcup \Sigma^{tid} \times V$ *is the set of directed edges labelled by task statements.*

The details about how to construct an extended directed graph for an OSEK/VDX application are stated as follow,

**Condition and assignment statements**: The condition statements and assignment statements are not computed in the sequentialization process. As shown in Fig. 7.6, we just symbolically execute condition statements and assignment statements.

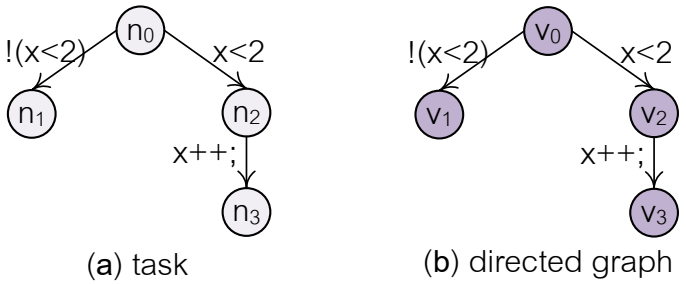



Figure 7.6: Process: condition and assignment statements

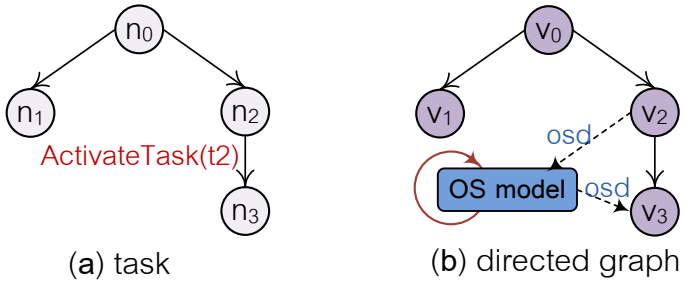



Figure 7.7: Process: API statement

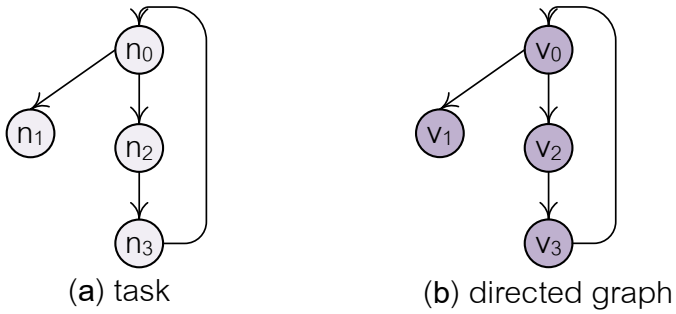
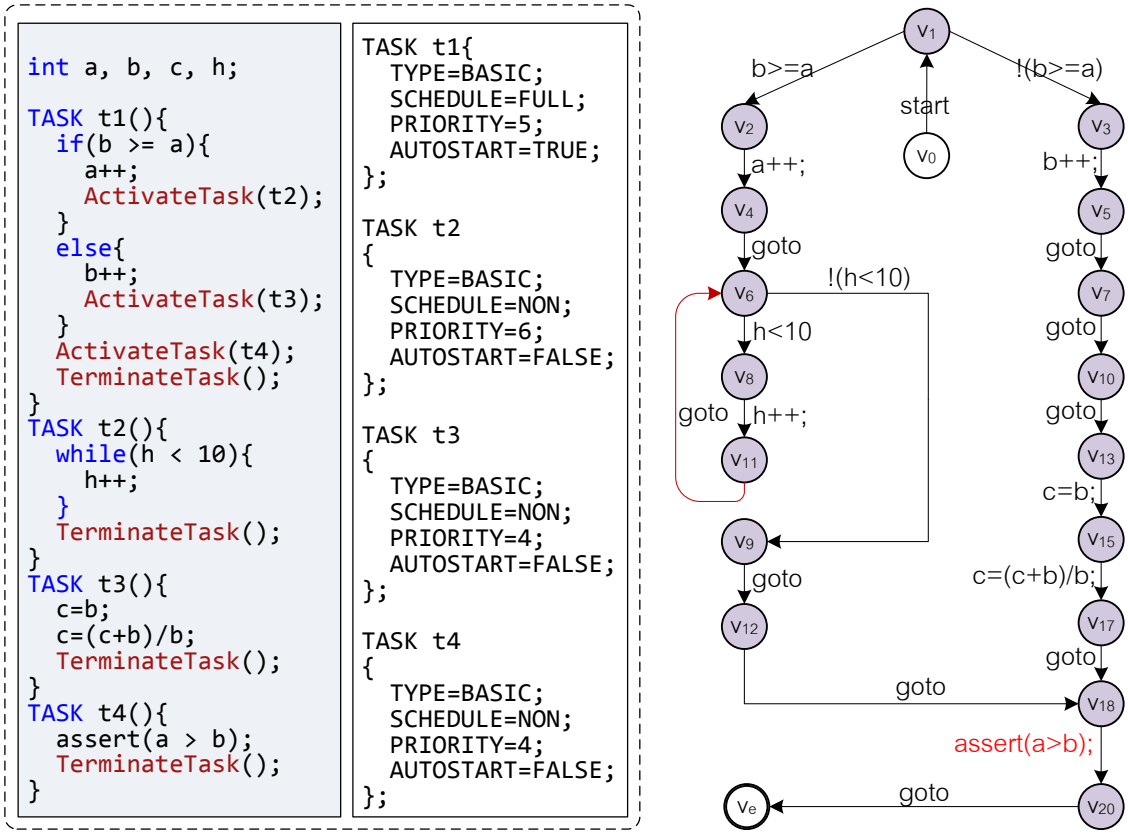



Figure 7.8: Process: loop

**API statement**: In the sequentialization process, when we meets an API, as shown in Fig. 7.7, we firstly pass the OS data *osd* of node to the embedded OSEK/VDX OS model; and then call the OS model to respond to the invoked API; finally, record the OS data of OS model in *osd* of node.

**loop**: In the sequentialization process, when we meets a loop, as shown in Fig. 7.8, we just use a cycle in directed graph to represent the loop in tasks (note that, loops in tasks are only computed on time in the sequentialization process).




(**a**) OSEK/VDX application (**b**) extended directed graph

Figure 7.9: Example for extended directed graph

Based on the above ideas, we develop a directed graph constructor (DGC) to construct a directed graph for representing the sequential model of OSEK/VDX application. The details about how to construct extended directed graph are stated in Alg. 2.

In Alg. 2, $\Theta$ which is a set is employed to store the transitions of directed graph, where an element $\gamma$ of $\Theta$ is a tuple $\gamma{=}(v, \alpha, v')$ (here, $v' \in V$ is the successor node of node $v \in V$, $\alpha \in \bigcup \Sigma^{tid}$ is the task statement). Moreover, for two elements $\gamma_i{=}(v_i, \alpha_i, v_i')$ and $\gamma_j{=}(v_j, \alpha_j, v_j')$, if and only if $v_i = v_j$ and $\alpha_i = \alpha_j$ and $v_i' = v_j'$, $\gamma_i$ is equal to $\gamma_j$. Note that, the API statement is replaced with *goto* statement in the process of constructing directed graph. The structure of Alg. 2 is as follows,

- The fragment (1-3) is the initial part.
- The fragment (5-10) is used to construct the start transition of directed graph.

**Algorithm 2** : Directed Graph Constructor

**Input:** task CFGs, configuration file of application
**Output:** directed graph

initialize $D$ of OS model with application config. file
initialize $pcs := [n_0^1, \ldots, n_0^m]$ with task start locations
$i := 1$, $j := 0$, where $i$ and $j$ are the index of nodes

call the interface function `StartTask`() of OS model to compute the running task
$D \rightarrow osd$, where the operator "$\rightarrow$" represents mapping the data within $D$ into $osd$
$v_0 := (pcs, osd)$
$v_1 := v_0$
$\gamma := (v_0, start, v_1)$ and set $\gamma$ as unexplored element
$\Theta := \Theta \cup \gamma$

**while** $\exists \gamma \in \Theta$ is an unexplored element **do**
  $(v, \alpha, v') := \gamma$ and set $\gamma$ as explored element
  $osd := osd$ within node $v'$
  $pcs := pcs$ within node $v'$
  $j :=$ the index of node $v'$
  $tid := runTask$ within $osd$
  **if** $tid =$ null **then**
    update the index of $v'$ of $\gamma$ with $e$, goto 12
  **end if**
  $\Delta := \{(n, n') \in \Omega^{tid} | n = pcs[tid]\}$

  **for all** $(n, n') \in \Delta$ **do**
    $i$ ++
    $v_j := v'$
    $\alpha :=$ the task statement mapped in $(n, n')$
    $pcs[tid] :=$ the target location $n'$ of $(n, n')$

    **if** $\alpha$ is an API **then**
      **if** $a$ is $TerminateTask()$ or $ChainTask()$ **then**
        $pcs[tid] :=$ the start location $n_0$ of task $tid$
      **end if**
      $osd \rightarrow D$, where the operator "$\rightarrow$" represents mapping the data within $osd$ into $D$ of OS model
      **call the interface function** `ResponseAPI`$(\alpha)$ **of OS model to respond to the service API** $\alpha$ **and compute the data within** $D$ **of OS model**
      $D \rightarrow osd$
      $\alpha := goto$
    **end if**

    $v_i := (pcs, osd)$
    $\gamma' := (v_j, \alpha, v_i)$

    **if** $\exists \gamma'' \in \Theta = \gamma'$ **then**
      $j :=$ the index of node $v$ within $\gamma''$
      update the index of $v'$ of $\gamma$ with $j$
      goto 12
    **end if**

    set $\gamma'$ as unexplored element, $\Theta := \Theta \cup \gamma'$
  **end for**
**end while**
**return**

```
v0:   if(b >= a)          goto v2;
      if(!(b >= a)        goto v3;
v2:   a++;                goto v6;
v6:   if(h < 10)          goto v8;
      if(!(h < 10))       goto v18;
v8:   h++;                goto v6;
v3:   b++;                goto v13;
v13:  c=b;                goto v15;
v15:  c=(c+b)/b;          goto v18;
v18:  assert(a > b);
```

Figure 7.10: C model for OSEK/VDX application sequential model

- The fragment (13-21) is used to compute the new transitions of directed graph.
- The fragment (29-37) is used to call the interface function `RespondAPI(API)` of OS model to respond to the invoked service API and compute the data within $D$ of OS model.
- The fragment (42-46) is used to avoid redundant transitions to be inserted into the directed graph.

### 7.3.2 Sequential Model

Based on the DGC, we can easily construct an extended directed graph to represent the sequential model of an OSEK/VDX application, e.g., as shown in Fig. 7.9, DCG will construct an extended directed graph for the shown application. Note that, int the constructed extended directed graph, the APIs in the original application are replaced with `goto` statements.

In the extended directed graph, the relationships between task statements are clearly specified by directed edges, we thus can easily compile the directed graph into the input language of beck-end model checker using `goto` statement, such as the input language `promela` of Spin and C language of `cbmc`. E.g., for the the extended directed graph shown in Fig. 7.9, we can use `goto` statements to compile the extended directed graph into the C language model which is illustrated in Fig. 7.10. A similar work for translated finite state machine into systemC has been presented in paper [49].

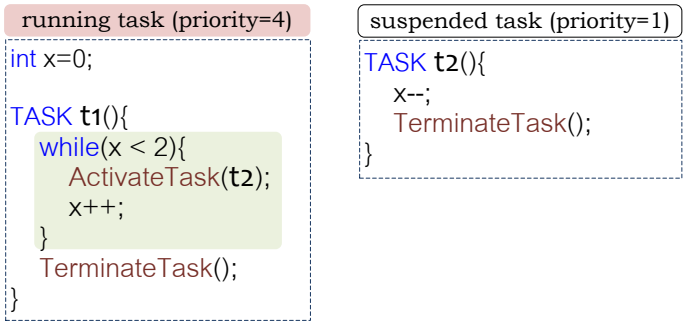



Figure 7.11: Example for multi-activations of tasks

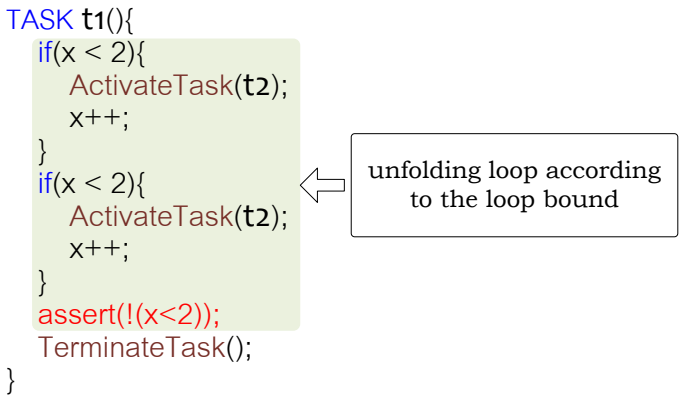



Figure 7.12: Unfolding loop for multi-activations of tasks

## 7.4 Multi-activations of Tasks

In OSEK/VDX applications, if a task is multiply activated by running task, the task will be directly moved to ready state after finished its executions until the activation times of the task have been used up. E.g., as shown in Fig. 7.11, task $t2$ is multiply activated 2 times by the currently running task $t1$ using a loop. In the example, since the priority of task $t2$ is lesser than task $t1$, $t2$ will not be executed although $t2$ has been activated. When running task $t1$ is terminated by scheduler, task $t2$ will be executed 2 times.

In our sequentialization approach, if a task is multiply activated by a loop of running task and the currently running task will be preempted by the activated task, our approach can precisely handle the multi-activations, because the multi-activations of tasks are like function calls. However, there is a limitation in our sequentialization approach, that is, our approach does not support the multi-activations of tasks shown in Fig. 7.11. E.g., for the example, in

our approach the task $t2$ will be activated only once rather than two times. This is because, in our approach we do not judge the execution times of loops, instead, loops are just symbolically executed on time, that is, we do not know how many times the task is activated by the loop. As to solve this limitation, before using DGC to generate extended directed graph, we can set a bound for the loop, and unfold the loop according to the set bound. The idea benefits from bounded model checking [11][14]. For instance, as shown in Fig. 7.12, for the loop shown in Fig. 7.11, we can unfold the loop two times according to the loop bound. In particular, in order to judge whether the loop is unfolded enough or not, we can insert an assertion which holds the negative condition of the loop in the end of unfolded loop body.

## 7.5 Implementation

We have developed a corresponding tool named `autoC`[1] according to the proposed approach. Currently, `autoC` support C programming language without `pointer`, `struct` as input language. Furthermore, `autoC` can output two types of sequential model, C sequential model and `promela` sequential model. The structure of `autoC` is shown in Fig. 7.13. `autoC` consists of four modules, implemented on the Visual Studio 2010 with 4000 lines of C++ code. The first module is to automatically interpret the behaviors of tasks into the corresponding CFGs. The second module is to extract the configuration data from configuration file of application. The third module is to implement the functionality of DGC. The last module is to compile the constructed extended directed graph into the C sequential model and `promela` sequential model.

## 7.6 Advantage and Disadvantage

As to efficiently verify OSEK/VDX applications using the existing model checkers such as Spin and `cbmc`, in our sequentialization-based checking approach, we present a novel method that can accurately and efficiently translate OSEK/VDX applications into sequential models. In the approach, we symbolically execute application using an extended directed graph under the scheduling of OSEK/VDX OS. There are several advantages in our approach, e.g.,

[1] `http://www.jaist.ac.jp/~s1220209/autoC.htm`

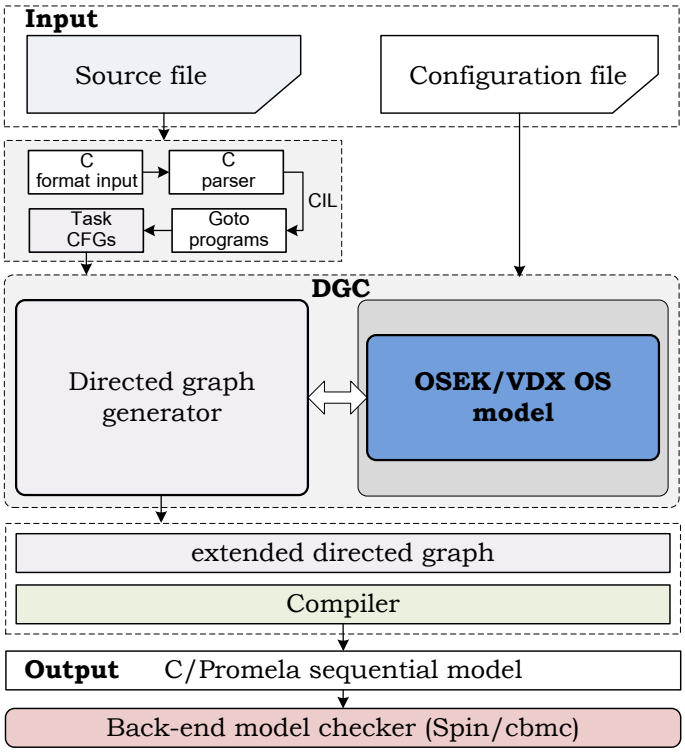



Figure 7.13: The structure of `autoC`

- The constructed sequential model (extended directed graph) can accurately reflect the executions of the OSEK/VDX application, since the embedded OSEK/VDX OS model is employed to respond to the invoked APIs and determine the running task in the sequentialization process.

- The sequential model does hold the behaviours of OSEK/VDX OS, because OSEK/VDX OS model is embedded in the DGC (translation algorithm flat).

- The same execution behaviours of application such as loops will not be repeatedly computed in the sequentialization process, since an extended graph is used to symbolically execute application.

- Our sequentialization-based approach can efficiently improve the efficiency and scalability of existing model checkers in checking OSEK/VDX applications, because the existing model checkers just verify a sequential model instead of a concurrent model.

However, there is a shortcoming in our approach, that is, our approach is not efficient to sequentialize the applications in which the APIs locating at branches will always lead to the different task execution orders. This is because, in our approach, as to accurately sequentialize an OSEK/VDX application, all of the different task execution orders are represented by the different sub-paths in the extended directed graph. For this type of applications, our approach will spend a lot of time exploring execution paths in the sequentialization process and finally construct a tree structure, which will slow down the performance of our approach.

# Chapter 8

# Experiment and Discussion

In this thesis, based on the model checking technique, we have presented three approaches to automatically verify OSEK/VDX applications. In this chapter, we will carry out two types of experiments to comprehensively evaluate the proposed checking approaches. In the first experiment, we will conduct several experiments to show the accuracy of our approaches. In the second experiment, we will show the efficiency and scalability of our approaches based on several experiments. Furthermore, based on the experiment results, the advantage and disadvantage of our approaches will be also discussed in this chapter.

## 8.1 Accuracy

As we know, accuracy is an very important criterion in the verification of software, since an inaccurate checking result will usually result in the extra costs, i.e., developers will spend extra time to judge whether the detected bug is a real one or not after completing verification. However, in the verification of OSEK/VDX applications, the existing model checking methods cannot achieve an accurate verification, because a lot of unnecessary interleavings of tasks within application will be checked in the verification stage. To accurately verify OSEK/VDX applications, we developed three approaches to construct an accurate model in order to make model checking techniques perform an accurate verification. Compared with the existing model checking methods for concurrent software, an accurate OSEK/VDX application model will be constructed under the explicitly scheduling of OSEK/VDX OS. Based on the constructed accu-

rate model, the unnecessary interleavings of tasks will not be checked in the verification stage. That is, our approaches can accurately check the OSEK/VDX applications using model checking technique, especially the spurious bugs will not be found by our approaches in the verification process. As to show the accuracy of our approaches, we will demonstrate several experiments in this section.

### 8.1.1 Experiment

In experiments, we adopt well-known Spin model checker instead of existing model checking methods for concurrent software to verify OSEK/VDX applications. In the checking method, OSEK/VDX scheduler is not taken into account in the verification process, and all of the tasks within the application are designed `process`es. In our checking approaches, we select sequentialization-based checking approach as comparison object, where we firstly use our approach to sequentialize OSEK/VDX application, and then use Spin to verify the sequentialized application. Furthermore, in order to show the accuracy of our approaches, the number of explored states and spurious bug are considered as the investigation points.

As to comprehensively evaluate the accuracy, the OSEK/VDX applications which hold different task number and API number are selected as our benchmarks. Moreover, as to really represent the execution behaviours of an OSEK/VDX application, the non-preemptive scheduling behaviour (e.g., nonpremt1_safe and nonpremt2_safe shown in Table 1), full-preemptive scheduling behaviour (e.g., fullpremt1_safe and fullpremt2_safe), mix-preemptive scheduling behaviour (e.g., mixpremt1_bug and mixpremt2_bug) are also taken into account in the selected benchmarks. Note that, the given property as assertion is inserted in the selected benchmarks.

All of the experiments are conducted on the Intel Core(TM)i7-3770 CPU with 32G RAM, and we set the time limit and memory limit to 600 seconds and 1GB, respectively. In addition, the "`C compiler`" of Spin is set to "-DVECTORSZ=16384 -DBITSTATE", and the max depth is set to "20,000,000". The experiment results have been listed in Table 1. In the result table, `#`$t$ is the number of tasks, `#API` is the number of times of invoked APIs, `#`$s$ is the number of explored states. According to the experiment results shown in Table 8.1, we can find the following results,

- The checking method Spin without scheduler checks more states than our sequentialization-based checking approach.

Table 8.1: Accuracy

| | | | Spin without Scheduler | | autoC+Spin | |
|---|---|---|---|---|---|---|
| benchmark | #t | #API | #s | result | #s | result |
| nonpremt1_safe | 2 | 3 | 17 | spurious | 5 | true |
| nonpremt2_safe | 3 | 5 | 33 | spurious | 10 | true |
| fullpremt1_safe | 2 | 3 | 17 | spurious | 5 | true |
| fullpremt2_safe | 3 | 5 | 33 | spurious | 10 | true |
| mixpremt1_bug | 3 | 5 | 33 | spurious | 10 | false |
| mixpremt2_bug | 4 | 7 | 69 | spurious | 13 | false |

- The checking method Spin without scheduler will usually find a spurious bug from checked benchmarks.
- Based on the sequentialization process of our approach, the back-end model checker Spin can accurately verify all of the benchmarks.

### 8.1.2 Discussion

#### Comparison to Spin without Scheduler

In the checking Spin method without scheduler, the checking model of benchmark holds all of the possible interleavings of tasks, because the running task is arbitrarily determined by Spin model checker in the verified stage. E.g., for the benchmark "nonpremt1_safe" shown in Fig. 8.1, Spin will construct a checking model illustrated in Fig. 8.2 to demonstrate all of the possible interleavings of tasks. According to the practical executions of the benchmark, we can easily find that there are several unnecessary interleaving of tasks in the constructed checking model. Moreover, due to the unnecessary interleavings, Spin will usually find a spurious bug in the verification stage.

However, in contrast with the checking method Spin without scheduler, in our approach the embedded OSEK/VDX OS model is used to respond to the APIs and determine the running task in the process of translating a benchmark into sequential model. Thus, the constructed sequential model in our approach is smaller than the checking method Spin without scheduler. Substantially, the constructed sequential model is an accurate model for OSEK/VDX applica-

Source file (.cpp)

```
int cnt = 0;

TASK t1(){
    if(cnt > 0)
        ActivateTask(t2);
    assert(cnt == 0);
    TerminateTask();
}

TASK t2(){
    cnt--;
    TerminateTask();
}
```

Configuration file (.oil)

```
TASK t1{
    TYPE=BASIC;
    SCHEDULE=FULL;
    PRIORITY=5;
    AUTOSTART=TRUE;
};

TASK t2{
    TYPE=BASIC;
    SCHEDULE=NON;
    PRIORITY=3;
    AUTOSTART=FALSE;
};
```

Figure 8.1: Benchmark: nonpremt1_safe

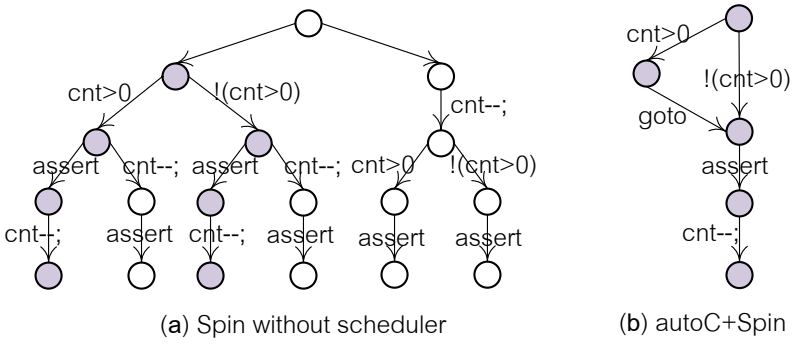



Figure 8.2: The checking model for benchmark nonpremt1_safe

tion, since all of the possible executions of application is poured into the sequential model under the explicitly scheduling of OSEK/VDX OS. Furthermore, the back-end model checker Spin can accurately verify the benchmark based on the constructed sequential model. E.g., for the same benchmark "nonpremt1_safe" shown in Fig. 8.1, our sequentialization-based checking approach will construct a sequential model shown in Fig. 8.2 and verified the sequential model using Spin model checker.

### Advantage and Disadvantage of Our Approaches

To accurately verify OSEK/VDX applications, as shown in Fig. 8.3, in our approaches we firstly developed an OSEK/VDX OS model according to the OSEK/VDX specification, and then employ the developed OS model to simulate the executions of target application in order to

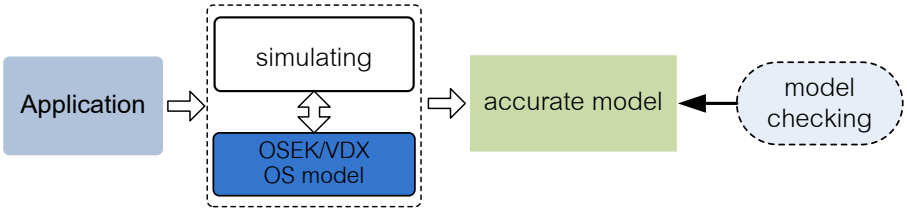



Figure 8.3: Key idea of our approaches

construct an accurate model. Based on accurate checking model, the explicitly executions of the application will be checked in the verification process. There is a advantage in our approaches, that is, we can apply model checking techniques to accurately verify OSEK/VDX applications based on the accurate checking model constructed by our approach. Moreover, developers do not need to spend extra costs judging whether the detected bug is a spurious bug or not when completing verification.

However, there is a disadvantage in our approaches, that is, our approaches deeply rely on the OSEK/VDX OS model. To accurately verify OSEK/VDX applications using our approaches, we have to develop an OSEK/VDX OS model according to the specification of OSEK/VDX OS. Furthermore, as to achieve an accurate checking result, we should spend extra costs ensuring whether the developed OSEK/VDX OS model straitly conforms to the OSEK/VDX OS specification. Even so, our approaches still hold the superiority in the verification of OSEK/VDX applications, e.g., once a reliable OSEK/VDX OS model is implemented, our approaches can be employed to accurately verify many of developed OSEK/VDX applications, which will significantly reduce the costs on the verification of OSEK/VDX applications.

As we know, it is difficult to develop correct OSEK/VDX OS model according to the OSEK/VDX specification. In our work, we have tested the developed OSEK/VDX OS model with at least 1000 experimental applications. Moreover, as to exhaustively find bugs from the developed OSEK/VDX OS model, the different program structures and scheduling behaviours such as sequential structure, branch structure, loop structure, non-preemptive behaviours, full-preemptive behaviours, mix-preemptive behaviours, synchronization behaviours and accessing shared resource behaviours have been carefully taken into account in the testing applications. Furthermore, we intend to apply the methods stated in papers [32][33][53] to ulteriorly check the developed OSEK/VDX OS model. We leave it as our future work.

## 8.2 Efficiency and State-space

In this experiment, we will make a comparison for our three approaches. In the comparison, we will investigate three points including explored state number, memory consumption and time consumption to shown the efficiency and scalability of our approaches.

### 8.2.1 Experiment

An OSEK/VDX application usually consists of tasks, APIs, loops, synchronization events and shared resources. In experiment, as to comprehensively evaluate the efficiency and State-space of our approaches on the realistic applications, the applications which hold different task number, API number, loop number and different scheduling behaviours are selected as our benchmarks. Furthermore, the applications which hold the non-preemptive scheduling behaviour (e.g., msgp1-msgp4 shown in Table 8.2), full-preemptive scheduling behaviour (e.g., token1-token4), mix-preemptive scheduling behaviour (e.g., rw_var1-rw_var4), synchronous behaviour (e.g., sync1-sync4), and accessing shared resource behaviour (e.g., acc_res1-acc_res4) are selected as benchmarks. The benchmarks used in the experiments and the prototype tools corresponding to the Spin-based checking approach, EPG technique and sequentialization-based checking approach are available at `http://www.jaist.ac.jp/~s1220209/autoC.htm`. All of the experiments are conducted on the Intel Core(TM)i7-3770 CPU with 32G RAM, and we set the time limit and memory limit to 600 seconds and 1GB, respectively.

In addition, in the experiment assertion as given property is checked by each approach. Moreover, as to make each approach checks the same state space, in the sequentialization-based checking approach, the well-known Spin is adopted as back-end model checker, and the max depth of Spin is set to "20,000,000". In the EPG technique, the max depth is set to "20,000,000", and the loop bound is set to 10. In the Spin-based checking approach, the "`C compiler`" of Spin is set to "-DVECTORSZ=16384 -DBITSTATE", and the max depth is set to "20,000,000".

The experiment results have been listed in Table 8.2. In the result table, #$t$ is the number of tasks, #$l$ is the number of loops, `#API` is the number of times of invoked APIs, #$s$ is the number of explored states. #$\widehat{s}$ is the number of times of computing states by `autoC`, "MB" and "time" are the memory consumption and time consumption, measured in Mbyte and second. M.O. and T.O. represent that the checking approach runs out of memory and time, respectively.

Table 8.2: Comparison: Spin-based approach, EPG technique and sequentialization-based approach

| benchmark | size | | | original version | | | | | | autoC | | | Spin | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Spin-based approach | | | EPG | | | | | | | | |
| | #$t$ | #$l$ | #API | #s | MB | time | #s | MB | time | #$\widehat{s}$ | MB | time | #s | MB | time |
| 1 msgp1_safe | 6 | 0 | 11 | 4652 | 17.7 | 0.21 | 89 | 2.17 | 0.29 | 48 | 3.190 | 0.162 | **36** | 0.02 | 0 |
| 2 msgp2_bug | 8 | 0 | 14 | 25253 | 81.1 | 0.71 | 55 | 2.18 | 0.35 | 66 | 3.646 | 0.239 | **38** | 0.02 | 0.01 |
| 3 msgp3_safe | 10 | 0 | 17 | 103835 | 303 | 2.67 | 157 | 2.19 | 0.49 | 84 | 4.102 | 0.276 | **60** | 0.04 | 0.01 |
| 4 msgp4_bug | 18 | 0 | 35 | - | - | T.O. | 145 | 2.22 | 0.57 | 148 | 5.724 | 0.576 | **99** | 0.09 | 0.02 |
| 5 token1_safe | 4 | 4 | 61 | 20501 | 2.45 | 0.10 | 2470 | 2.26 | 43.4 | 39 | 2.962 | 0.120 | **1008** | 0.50 | 0.11 |
| 6 token2_bug | 6 | 6 | 101 | 34371 | 126 | 1.12 | 2283 | 2.23 | 139 | 54 | 3.342 | 0.177 | **1624** | 0.93 | 0.16 |
| 7 token3_safe | 9 | 9 | 161 | 46990 | 138 | 1.26 | 6417 | 2.41 | 192 | 82 | 4.052 | 0.271 | **2607** | 1.79 | 0.27 |
| 8 token4_bug | 13 | 13 | 241 | - | M.O. | - | - | - | T.O. | 125 | 5.134 | 0.427 | **3904** | 2.78 | 0.39 |
| 9 rw_var1_safe | 5 | 5 | 101 | 13541 | 54.7 | 0.44 | 3259 | 2.30 | 78.4 | 49 | 3.214 | 0.164 | **1308** | 0.75 | 0.13 |
| 10 rw_var2_safe | 9 | 9 | 161 | - | M.O. | - | 6459 | 2.41 | 450 | 89 | 4.226 | 0.283 | **2588** | 1.71 | 0.26 |
| 11 rw_var3_safe | 13 | 13 | 241 | - | M.O. | - | - | - | T.O. | 129 | 5.239 | 0.431 | **3868** | 2.56 | 0.38 |
| 12 rw_var4_safe | 13 | 12 | 13 | - | M.O. | - | 940 | 2.34 | 13.4 | 111 | 4.783 | 0.398 | **390** | 0.24 | 0.04 |
| 13 sync1_safe | 5 | 4 | 14 | 2443 | 11.6 | 0.15 | 336 | 2.22 | 4.02 | 47 | 3.164 | 0.163 | **134** | 0.11 | 0.02 |
| 14 sync2_safe | 8 | 7 | 23 | 24159 | 86.8 | 0.78 | 585 | 2.27 | 7.14 | 80 | 4.001 | 0.270 | **230** | 0.14 | 0.02 |
| 15 sync3_bug | 11 | 10 | 32 | 210841 | 612 | 5.37 | 458 | 2.31 | 10.7 | 101 | 4.533 | 0.381 | **322** | 0.02 | 0.03 |
| 16 sync4_safe | 12 | 11 | 42 | 382329 | 998 | 9.62 | 917 | 2.33 | 11.9 | 124 | 5.115 | 0.428 | **358** | 0.20 | 0.03 |
| 17 acc_res1_safe | 2 | 3 | 4 | 13907 | 56.5 | 0.46 | 3491 | 2.32 | 88.8 | 57 | 3.416 | 0.179 | **1308** | 0.75 | 0.13 |
| 18 acc_res2_safe | 9 | 9 | 320 | - | M.O. | - | 6779 | 2.41 | 509 | 105 | 7.631 | 0.377 | **2588** | 1.71 | 0.26 |
| 19 acc_res3_safe | 13 | 13 | 480 | - | M.O. | - | - | - | T.O. | 153 | 5.846 | 0.578 | **3868** | 2.56 | 0.38 |
| 20 acc_res4_bug | 12 | 12 | 25 | - | M.O. | - | 552 | 2.13 | 13.9 | 119 | 4.986 | 0.409 | **389** | 0.24 | 0.03 |

According to the experiment results shown in Table 8.2, we can find the following results,

(*i*) The Spin-based checking approach can accurately verify the OSEK/VDX applications, but the scalability of this approach is limited, e.g., for the benchmarks (lines 8, 10, 11, 12, 18, 19 and 20), it will run out of memory or time.

(*ii*) EPG technique is more efficient and scalable than Spin-based checking approach in checking the applications which hold a lot of tasks and APIs, e.g., lines 4 and 18. However, it is not efficient to check the applications which hold a large number of loops with APIs, e.g., for the benchmark (lines 8, 11 and 19), it will run out of time.

(*iii*) The sequentialization approach (`autoC`) can efficiently sequentialize the given benchmarks. Moreover, based on the sequentialization process of our approach, the back-end model checker Spin can successfully verify all of the benchmarks which hold a lot of tasks, loops and APIs with the less cost in terms of states, time and memory compared with the Spin-based checking approach and EPG technique.

### 8.2.2 Discussion

#### Spin-based checking approach

In the Spin-based checking approach, the checking model is a combination model included two models, one is application model, and the other is OSEK/VDX OS model. Where, the application model is used to simulate the executions of tasks within application (tasks are designed as `process`es), the OSEK/VDX OS model as a cooperative `process` is used to dispatch tasks and respond to the APIs invoked from tasks. Based on the combination model, the practical execution behaviours of OSEK/VDX application are accurately simulated in the verification process. Thus, the approach is an accurate checking method for OSEK/VDX applications.

However, in the approach the behaviour of OSEK/VDX OS model will be explored in the verification stage, because the OSEK/VDX OS model is inside the checking model. When an application invokes an API, the OSEK/VDX OS model within the combination model will be loaded to respond to the API. Sequentially, the states of OSEK/VDX OS model will be explored once by Spin in the verification stage. If an application invokes a lot of APIs, a large number of states from OSEK/VDX OS model will be explored in the verification stage. The scalability of

the approach is limited, because too many details of OSEK/VDX OS model are explored in the verification process, especially the state space explosion will happen if a large number of APIs are invoked by application.

### EPG technique

In the EPG technique, compared with Spin-based checking approach, we embed OSEK/VDX OS model in EPG (constructing model algorithm flat) to dispatch tasks and respond to the APIs invoked from tasks in the checking process in order to avoid OSEK/VDX OS model to be explored in the verification stage. Moreover, as to handle the complex applications which hold a large number of states, the state-of-art SMT solver Z3 is used to carry out the verification. The following advantages make EPG technique more efficient and scalable in checking OSEK/VDX applications in contrast with Spin-based checking approach,

- EPG: we can construct a transition system without the behaviors of OSEK/VDX OS model for the target OSEK/VDX application based on the generated execution paths, since OSEK/VDX OS model is embedded in EPG to dispatch tasks and respond to the APIs invoked from tasks.

- SMT-based BMC: it can make our approach more efficient and scalable in checking the complex applications which hold a large number of states, because the state-of-the-art SMT solver Z3 is used to carry out the verification.

However, the approach is not efficient to handle the applications which hold a lot of loops with APIs. This is because, in our EPG technique the transition system of an OSEK/VDX application is constructed based on the execution paths. If the application holds a lot of loops with APIs, it will spend a lot of time exploring execution paths for constructing transition system under the set loop bound, which will slow down the performance of our EPG technique.

### Sequentialization-based checking approach

In the sequentialization-based checking approach, we symbolically execute application using an extended directed graph under the scheduling of OSEK/VDX OS in order to accurately and efficiently translate OSEK/VDX applications into sequential models. There are several

Table 8.3: Scalability and Efficiency of `autoC` and `cbmc`

| | | | | | autoC | | cbmc | |
|---|---|---|---|---|---|---|---|---|
| benchmark | #$t$ | #$l$ | #API | #loc | MB | time | MB | time |
| CCS | 21 | 25 | 241 | 969 | 6.84 | 0.826 | 8.43 | 1.21 |
| WCS | 18 | 5 | 171 | 1000 | 5.04 | 0.572 | 6.79 | 1.22 |
| TCS | 25 | 9 | 1281 | 1144 | 8.76 | 1.196 | 7.69 | 1.28 |
| ABS | 25 | 7 | 1141 | 1124 | 8.84 | 1.224 | 9.23 | 1.34 |

advantages that make the checking approach more efficient and scalable in checking OSEK/VDX applications, e.g.,

- The constructed sequential model (extended directed graph) can accurately reflect the executions of the OSEK/VDX application, since the embedded OSEK/VDX OS model is employed to respond to the invoked APIs and determine the running task in the sequentialization process.
- The sequential model does hold the behaviours of OSEK/VDX OS, because OSEK/VDX OS model is embedded in the DGC (translation algorithm flat).
- The same execution behaviours of application such as loops will not be repeatedly computed in the sequentialization process, since an extended graph is used to symbolically execute application.
- The sequentialization-based approach can efficiently improve the efficiency and scalability of existing model checkers in checking OSEK/VDX applications, because the existing model checkers just verify a sequential model instead of a concurrent model.

Compared with Spin-based checking approach, the behaviours of OSEK/VDX OS model will not be checked in the verification stage. Moreover, in contrast with EPG technique, in the sequentialization-based checking approach we do not need to explore execution paths, instead, we use an extended directed graph to symbolically execute an application under the scheduling of embedded OSEK/VDX OS model in order to avoid the same execution behaviours of application such as loops to be repeatedly computed in the sequentialization process. These efforts make the approach more efficient and scalable than the EPG technique and Spin-based checking approach.

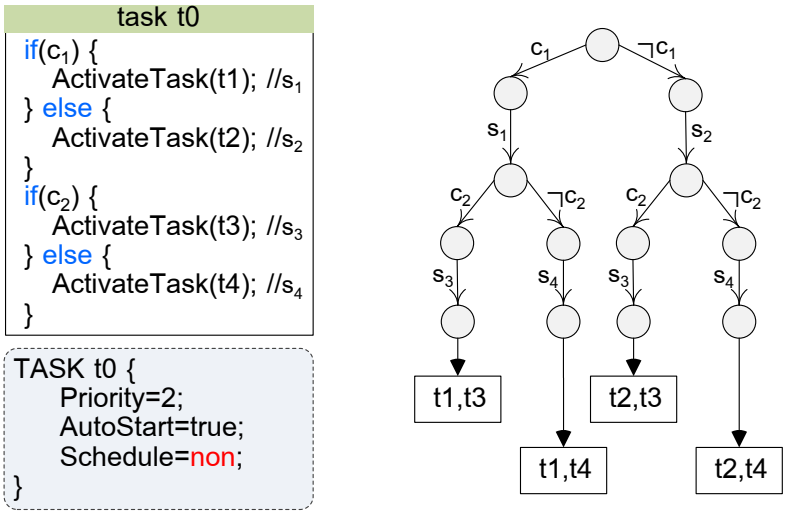



Figure 8.4: The worst-case of sequentialization approach

Furthermore, we have used `autoC` to sequentialize many large-scale experimental OSEK/VDX applications, and verified the sequentialized applications with the well-known model checker `cbmc`. The performances of `autoC` and `cbmc` indicate that `autoC+cbmc` can be considered as a practical method to verify the large-scale OSEK/VDX applications. In order to demonstrate the scalability and efficiency of `autoC` and `cbmc`, in the experiments four applications that are used to model the cruise control system (CCS), windows control system (WCS), temperature control system (TCS) and antilock brake system (ABS) are selected as benchmarks.

All of the experiments are conducted on the Intel Core(TM)i7-3770 CPU with 32G RAM, and the max depth and loop bound of `cbmc` are set to 20,000,000 and 10, respectively. The experiment results have been listed in Table 8.3. In the result table, `#`$t$ is the number of tasks, `#`$l$ is the number of loops, `#API` is the number of times of invoked APIs, `#loc` is the number of lines of code, "MB" and "time" are consumption on the memory and time, measured in Mbyte and second. According to the experiment results, it can be seen that the sequentialization-based approach and `cbmc` can be considered as a practical method to verify the OSEK/VDX applications with industrial complexity.

However, compared with EPG technique, there is a disadvantage or worst-case in our sequentialization approach. As shown in Fig. 8.4, our approach is not efficient to sequentialize the applications in which the APIs locating at branches will always lead to the different task exe-

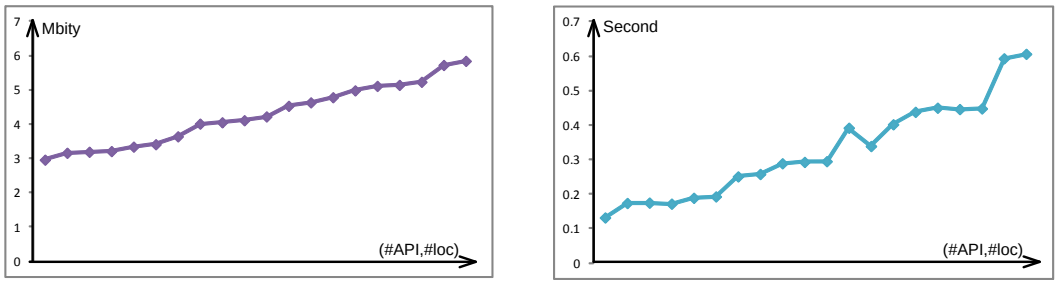



Figure 8.5: The memory and time consumptions of sequentialization approach

cution orders. This is because, in our approach, as to accurately sequentialize an OSEK/VDX application, all of the different task execution orders are represented by the different sub-paths in the extended directed graph. For this type of applications, our approach will spend a lot of time exploring execution paths in the sequentialization process, and finally construct a tree structure which is the same as the tree in our EPG technique.

In addition, as shown in Fig. 8.5, in order to indicate the efficiency and scalability of our sequentialization approach for OSEK/VDX applications, we have evaluated the memory and time consumptions of our approach based on the experiment results shown in Table 8.2. In our sequentialization approach, there are two crucial points that can affect the performance of our approach. First one is the scale of application (i.e., lines of code), since our approach will spend more costs sequentializing more large-scale application. Second one is APIs locating at branches. This is because, as shown in Fig. 8.4, the APIs locating at branches will lead to the different executions of the application. To sequentialize this type of applications, our approach will spend more costs exploring sub-paths. In the shown figures, we use (`#API`, `#loc`) to represent $X$-axis, where `#API` is number of times of invoked APIs, `#loc` is the number of lines. $Y$-axis is the time consumption or memory consumption. Note that, the parameters `#API` and `#loc` in Fig. 8.5 are simultaneously increased, and the loops in the application are unfolded as branch structures.

# Chapter 9

# Related Work

With the development of OSEK/VDX OS standard, OSEK/VDX is not only widely adopted by automobile manufacturers to implement a vehicle-mounted OS, but also employed by other electronics manufacturers to develop an embedded system such as [1][44][46][48]. Currently, with the continuously increasing complexity in the development process, how to ensure the reliability of the developed OSEK/VDX OS and its applications is becoming a challenge for developers.

## 9.1 Model Checking on OSEK/VDX OS

In the scope of verifying developed OSEK/VDX OS, there are many invaluable model checking methods, e.g., Chen and Aoki have proposed a method [32] to generate the highly reliable test-cases for checking whether the developed OS conforms to the OSEK/VDX OS standard based on the Spin model checker. As to support an environment of OSEK/VDX OS for the model checking, an UML-based method for producing Promela scripts of OSEK/VDX OS is proposed in paper [33]. In addition, for the Trampoline [1] which is an open source RTOS developed based on the OSEK/VDX OS standard, Choi presented a method [53] to convert the Trampoline kernel into formal models, and in the method an incremental verification approach is proposed to carry out the verification. Furthermore, a CSP-based approach for checking the code-level OSEK/VDX OS is addressed in the paper [52].

These existing works are different from our work, because these works focus on the verification of developed OSEK/VDX OS. Compared with the existing works, our work concentrates

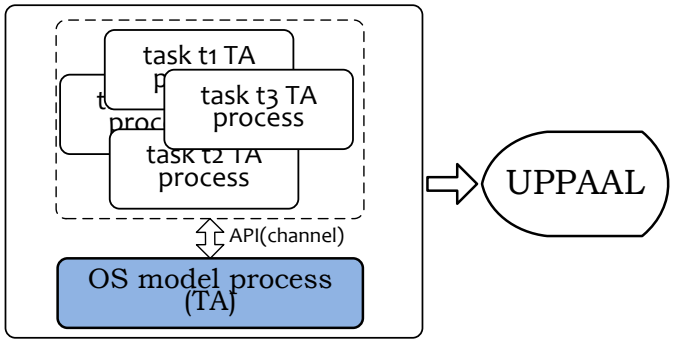



Figure 9.1: Timing property of OSEK/VDX application

on the verification of developed OSEK/VDX applications.

## 9.2 Model Checking on OSEK/VDX Applications

To the developed OSEK/VDX applications, Libor has proposed a method [36] to check the timing property based on the UPPAAL. In the method, as shown in Fig. 9.1, all of the tasks within the application are represented as timed automaton (TA), and the task timed automaton are designed `process`es. Moreover, an OSEK/VDX OS as a cooperative `process` is used to conduct the executions of tasks and respond to the API invoked from tasks.

Our Spin-based checking approach is similar to this method in using a combination model of application model and OSEK/VDX OS model to simulate the executions of application. However, our Spin-based checking approach focuses on the safety property. In addition, compared with this method, in our EPG technique and sequentialization-based checking approach, we embedded OSEK/VDX OS model in the constructing model algorithm flat to conduct the executions of tasks and respond to the API invoked from tasks, which is different from this method.

## 9.3 Model Checking on Concurrent Software

Model checking as an exhaustive verification technique has been widely applied to check concurrent software. E.g., Cimatti has proposed a method [13] to verify SystemC multi-threaded software based on the predicate abstraction [22]. In the method, as shown in Fig. 9.2, since a

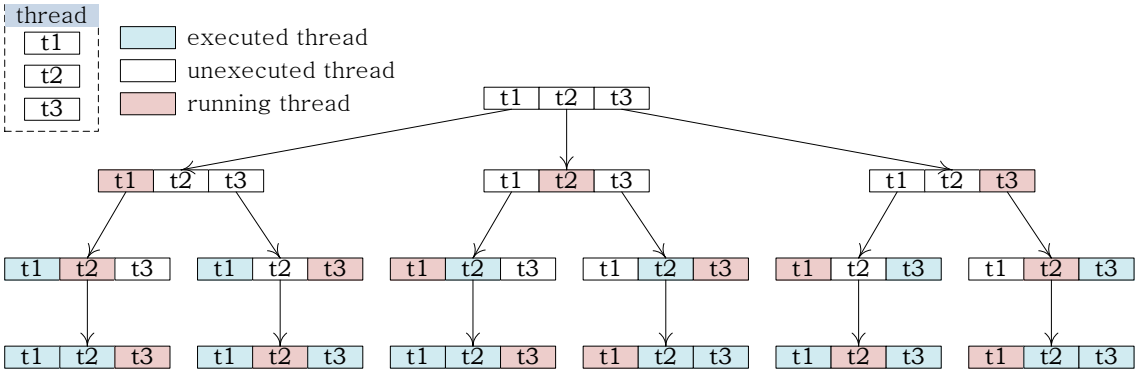



Figure 9.2: All of the interleavings for three threads $t1$, $t2$ and $t3$

non-deterministic scheduler is adopted by SystemC to dispatch threads, all of the possible interleavings of runnable threads are checked in the verification stage in order to completely check the target software. Particularly, in this method, in order to reduce the unnecessary interleavings of threads caused by synchronization events, a scheduler that conforms to SystemC standard is embedded in constructing model algorithm flat to determine runnable threads in the verification process.

Our EPG technique is similar to this method in embedding scheduler model in constructing model algorithm flat, but we handle deterministic scheduling software, and use SMT-based BMC to perform the verification, which are different from ESST technique. In addition, the method is not suitable to check OSEK/VDX applications because the deterministic scheduler is adopted by OSEK/VDX OS to dispatch tasks within the application. Compared with this method, our EPG technique is more accurate in checking OSEK/VDX applications.

In the scope of verifying multi-threaded software using bounded model checking technique, Lucas presented a SMT-based BMC method to check ANSI-C multi-threaded software in paper [37]. In the method, like the method proposed by Cimatti, all of the possible interleavings of threads are taken into account in the verification stage, because in ANSI-C the non-deterministic scheduler is used to dispatch threads. In addition, in the method, two functions named "`pthread_mutex_lock()`" and "`pthread_cond_wait()`" are used to simulate mutex lock operation and synchronous behaviours between threads. In the verification, these two functions can be called by threads to implement the mutual and synchronous behaviours.

Our EPG technique is similar to this method in using SMT-based BMC to perform the

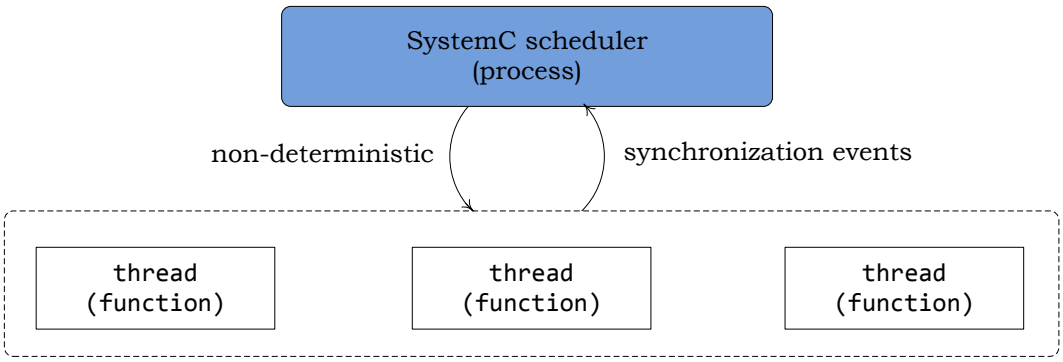



Figure 9.3: Sequentialization process for SystemC multi-threaded software

verification. However, our work handles deterministic scheduling software, and the interactive behaviours between tasks and OS such as mutual and synchronous behaviours will not be checked in the verification stage (the interactive behaviours between tasks and OS are directly processed by embedded OSEK/VDX OS model in our approach). Moreover, like the method proposed by Cimatti, the method is also not suitable to check OSEK/VDX applications because the deterministic scheduler is adopted by OSEK/VDX OS to dispatch tasks within the application.

## 9.4 Sequentialization-based Model Checking on Concurrent Software

Sequentialization-based model checking method has been successfully applied to verify concurrent software. E.g., Campana has proposed a method to translate the SytemC multi-threaded software into the corresponding sequential software based on the Spin model checker in paper [19]. In the method, as shown in Fig. 9.3, all of the threads are designed as functions. Particularly, as to simulate the executions of the threads, the SytemC scheduler as a `process` is used to conduct the executions of functions. In addition, Inverso also shown an approach [42] to translate the multi-threaded software that conforms to the POSIX standard [30] into a non-deterministic sequential program. Unlike the method proposed by Campana, in the method a function named "`CS()`" is used to simulate the context switch of threads instead of scheduler.

In our sequentialization-based checking approach, the scheduler model is embedded in the translation algorithm flat to conduct the executions of application, the constructed sequential

model of application does not hold the behaviours of scheduler. In the verification stage, the behaviour of scheduler will not be checked. In contrast with our approach, in the shown existing works scheduler or function "`CS()`" is a part of sequential model. In the verification stage, the behaviours of scheduler or function "`CS()`" will be checked. Moreover, these two methods are not suitable to sequentialize OSEK/VDX applications, because they focus on the non-deterministic scheduler software. In contrast to non-deterministic scheduler, in OSEK/VDX applications the deterministic scheduler is adopted by OSEK/VDX OS to dispatch tasks.

# Chapter 10

# Conclusion

OSEK/VDX as a standard of automobile OS has been widely adopted by many automobile manufacturers to design and implement a vehicle-mounted OS, such as BMW, Opel and TOYOTA. Based on the OSEK/VDX OS, more and more applications are developed and deployed in vehicles to assist drivers to control vehicles, such as the cruise control system and temperature control system. However, with the growing functionalities in vehicles and increasing complexity in the development, how to straitly ensure the reliability of the developed OSEK/VDX applications is becoming a challenge for developers.

When an application runs on OSEK/VDX OS, the running task within application is explicitly determined by OSEK/VDX scheduler (deterministic scheduling policy is used to dispatch tasks), and the APIs invoked from tasks will haphazardly affect the scheduling of tasks. To ensure the reliability of OSEK/VDX applications, model checking as an efficient and exhaustive verification technique for concurrent software can be applied to verify OSEK/VDX applications for detecting the subtle and logic errors. Although there exist many model checking methods that have been successfully applied to verify concurrent software such as SystemC and ANSI-C multi-threaded software, these methods usually perform an approximate verification since the behaviours of scheduler are not taken into account in verification process. If we apply these existing model checking methods to verify OSEK/VDX applications, it is too imprecise since a lot of unnecessary interleavings of tasks will be superfluously checked in the verification stage, especially these unnecessary interleavings will usually result in a spurious bug. As a result of the spurious bug, developers have to spend extra costs judging whether the detected bug is real

one or not after completing verification. As to reduce the checking costs, a more accurate model checking approach should be used in the verification of OSEK/VDX applications. Furthermore, to accurately check concurrent software such as ANSI-C multi-threaded software and SystemC multi-threaded software, several prominent model checking methods have been proposed by some senior researchers. In these methods, as to seek a more accurate verification result, the behaviours of scheduler are taken into account in the verification process. Even so, these prominent model checking methods are still unsuitable to accurately verify OSEK/VDX applications because these prominent methods focus on the non-deterministic scheduler based concurrent software. In contrast with non-deterministic scheduler, in OSEK/VDX applications a deterministic scheduler is used to dispatch tasks. In this thesis, we described and developed three approaches that can accurately and automatically verify the safety property of OSEK/VDX applications using model checking technique. To the best of our knowledge, our work is first to apply model checking technique to accurately verify the multi-tasks/threads software which is dispatched by a deterministic scheduler.

## 10.1 Checking Approaches

To accurately verify OSEK/VDX applications using model checking technique, a cheap method is based on the existing model checker, because we do not need to establish a new model checker. Thus, in our first approach we investigate a method based on the existing model checker Spin. In the method, as to make Spin model checker accurately verify OSEK/VDX applications, a checking model which is a combination model of application model and OSEK/VDX OS model is employed to precisely simulate the executions of the OSEK/VDX applications. We have conducted many experiments using this method. The experiment results show that, although our Spin-based checking approach can accurately verify OSEK/VDX applications, the scalability of this approach is limited because too many details of OSEK/VDX OS model will be explored in the verification stage, especially the state space explosion may happen if application invokes a lot of APIs.

In the second approach, as to verify a complex application which holds a large number of states and APIs, we develop a new technique named execution path generator (EPG) to verify OSEK/VDX applications based on the advanced SMT-based bounded model checking technique.

Particularly, in the approach the OSEK/VDX OS model is embedded in EPG (constructing model algorithm flat) to dispatch tasks and respond to the invoked APIs in the process of constructing transition system of application. In OSEK/VDX applications, the APIs invoked from tasks will haphazardly affect the scheduling of tasks, e.g., the context switch of tasks may happen when an API is invoked, and the different APIs location at different branches will lead to different task execution orders. In our EPG technique, like other bounded model checking for concurrent software (e.g., the method [37] for ANSI-C multi-threaded software), a reachability tree as an intermediate form is constructed for establishing the transition system of application. Based to the proposed approach, we established new bounded model checker named `osek-bmc` for OSEK/VDX applications, and carried out many experiments using the established model checker. The experiment results show that our EPG technique is more scalable than Spin-based checking approach. However, it is not efficient to check the large-scale applications which hold a lot of branches, because EPG will spend a lot of time exploring execution paths for constructing the reachability tree in the checking process, which will slow down the efficiency of the approach.

As to efficiently handle the applications which hold a large number of branches, in the last approach we present a novel method to translate OSEK/VDX applications into sequential models in order to apply the existing model checkers such as `spin` and `cbmc` to efficiently verify OSEK/VDX applications. As to efficiently construct a sequential model for OSEK/VDX application. In our sequentialization-based checking approach, we use an extended directed graph instead of reachability tree to construct the sequential model of OSEK/VDX application. Particularly, as to avoid the behaviours of OSEK/VDX OS to be poured into the sequential model, like EPG technique we embed OSEK/VDX OS model in translation algorithm flat to dispatch tasks and respond to the invoked APIs. There are several strengths from extended directed graph that can make our sequentialization-based checking approach more efficient than EPG technique in checking the applications which hold a lot of branches, e.g., ($i$) based on the directed graph, we do not need to explore execution paths when meeting branches in the sequentialization process, instead, we can use the combination states to reduce the computation times; ($ii$) based on the directed graph, we do not need to repeatedly compute the same execution behaviours of application in the sequentialization process, instead, we can construct a cycle in directed graph to represent the same execution behaviours. We have implemented a tool named `autoC` according to the proposed approach, and evaluated the proposed approaches based on a

series of experiments. The experiment results show that our sequentialization-based checking approach is more efficient and scalable than EPG technique and Spin-based checking approach. Furthermore, we have used `autoC` to sequentialize many experimental OSEK/VDX applications which hold about 1000 lines of `C` code, and verified the sequentialized applications using the well-known bounded model checker `cbmc`. The performances indicate that `autoC` and `cbmc` can be considered as a practical method to verify the OSEK/VDX applications with industrial complexity.

## 10.2 Merits and Drawbacks of Checking Approaches

To accurately verify OSEK/VDX applications, in our approaches the OSEK/VDX OS model as a cooperative component is employed to dispatch tasks and respond to the APIs invoked from tasks. Based on the OSEK/VDX OS model, the explicitly executions of the application are checked in the verification process. There is a key merit in our approaches, that is, developers do not need to spend extra costs judging whether the detected bug is a spurious bug or not. To check a realistic OSEK/VDX application, there are several arguments that can be considered as criteria to apply our approaches in the verification of OSEK/VDX applications.

- If the given application holds a few tasks and does not invoke a lot of APIs, the Spin-based checking approach is the best choice, since the approach is based on the existing model checker Spin. Compared with EPG technique and sequentialization-based checking approach, we do not need to establish a new model checker and translator.

- If the given application holds a lot of tasks, loops and APIs, the sequentialization-based checking approach is an outstanding checking method compared with Spin-based checking approach and EPG technique. This is because, although we should establish a translator compared with Spin-based checking approach, the approach can successfully verify the given application. In addition, in contrast with EPG technique, we just need to implement a translator rather than a new model checker.

- Compared with sequentialization-based checking approach, EPG technique is not efficient for the application which holds a lot of branches and loops. Moreover, a new model checker needs to be established in contrast with Spin-based checking approach. Even so, there is

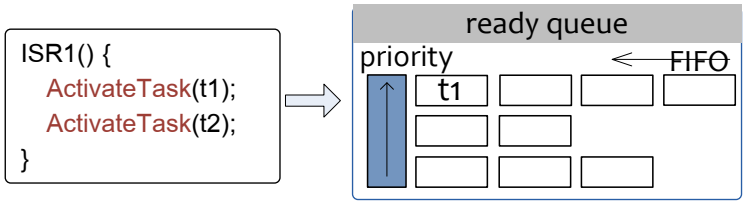



Figure 10.1: The ISR with APIs

a merit that makes EPG technique worthwhile, that is, the approach can be considered as generator to generate the test-cases for the path coverage and condition coverage, since all of the execution paths within the given application will be explored in the verification process, and the back-end SMT solver Z3 will efficiently give a test-case for the specified execution path.

However, there is a drawback in our approaches, that is, our approaches deeply rely on the OSEK/VDX OS model. To accurately verify OSEK/VDX applications using our approaches, we have to develop an OSEK/VDX OS model according to the specification of OSEK/VDX OS. Furthermore, as to achieve an accurate checking result, we should spend extra costs ensuring whether the developed OSEK/VDX OS model straitly conforms to the OSEK/VDX OS specification. Even so, our approaches still hold the superiority in the verification of OSEK/VDX applications, e.g., once a reliable OSEK/VDX OS model is implemented, our approaches can be employed to accurately verify many developed OSEK/VDX applications, which will significantly reduce the costs on the verification of developed applications. In addition, in our work we have developed an OSEK/VDX OS model and tested the developed OSEK/VDX OS model with at least 1000 experimental applications. We believe that the developed OSEK/VDX OS model can make our approaches trustworthy and serviceable in the verification of OSEK/VDX applications.

## 10.3 Future Work

In the future, we intend to extend our approaches to verify the OSEK/VDX applications which hold interrupt service routines (ISRs). There are several challenges that should be addressed when we applied model checking to verify the application with ISRs, e.g.,

- How to simulate the occurrence positions of ISRs. In OSEK/VDX applications, the occurrence positions of ISRs are difficult to be statically determined in the verification process, because the ISRs are triggered by external environment.

- How to simulate the occurrence times of ISRs. In OSEK/VDX applications, the occurrence times of ISRs cannot be statically determined in the verification process, since an ISR will not happen or happen many times.

- Furthermore, how to simulate the ISRs with APIs. In OSEK/VDX applications, as shown in Fig. 10.1, ISRs can invokes APIs, and the invoked APIs will dynamically change the scheduling of tasks. The deterministic scheduling policy in OSEK/VDX applications will be changed to non-deterministic scheduling policy by the APIs invoked from ISRs.

# Publications


[1] Haitao Zhang, Toshiaki Aoki, and Yuki Chiba: A Spin-based Approach for Checking OSEK/VDX Applications, the international workshop FTSCS in ICFEM, pp.1-17 (2014)

[2] Haitao Zhang, Toshiaki Aoki, Kenro Yatake, Min Zhang, and Hsin-Hung Lin: An Approach for Checking OSEK/VDX Applications, 13th Quality Software International Conference (QSIC), pp.113-116 (2013)

[3] Haitao Zhang, Toshiaki Aoki, Hsin-Hung Lin, Min Zhang, and Kenro Yatake: SMT-Based Bounded Model Checking for OSEK/VDX Applications, 20th Asia-Pacific Software Engineering Conference (APSEC), pp.307-314 (2013)

[4] Haitao Zhang, Toshiaki Aoki, and Yuki Chiba: Yes! You Can Use Your Model Checker to Verify OSEK/VDX Applications, IEEE International Conference on Software Testing, Verification and Validation (ICST), full paper (10 pages) (2015)

[5] Haitao Zhang, Toshiaki Aoki, and Yuki Chiba: Verifying OSEK/VDX Applications: A Sequentialization-based Model Checking Approach, accepted by IEICE Transactions (2015)

[6] Haitao Zhang, Toshiaki Aoki, and Yuki Chiba: Verifying OSEK/VDX Applications: A SMT-based Bounded Model Checking Approach, conditionally accepted by IEICE Transactions (2015)

# References


[1] Trampoline. http://trampoline.rts-software.org/

[2] Edmund M. Clarke, E. Allen Emerson, et al.: Model Checking: Algorithmic Verification and Debugging. Commun. ACM 152(11), 74–84 (2009)

[3] Edmund M. Clarke, Orna Grumberg and David E. Long: Model Checking and Abstraction. ACM Trans pp. 1512–1542 (1994)

[4] Edmund M. Clarke, William Klieber, et al.: Model Checking and the State Explosion Problem. Tools for Practical Software Verification 7682, 1–30 (2012)

[5] R. E. Tarjan: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (Jun 1972)

[6] A. Biere, A. Cimatti, E. Clarke, Y. Zhu: Symbolic model checking without BDDs. TACAS'99, Lecture Notes in Computer Science 1579, 193207 (1999)

[7] A. Cimatti, E. Clarke, F: NuSMV: a new symbolic model verifier. International Conference on Computer-Aided Verification (CAV) pp. 495–499 (1999)

[8] Adrian Balint, Anton Belov, Marijn Heule, et al.: SAT Competitions. http://www.satcompetition.org/

[9] Alan Burns and Andy Wellings: Real-Time Systems and Programming Languages (4th Edition). Addison Wesley Longmain, New York, NY, USA (Aug 2009)

[10] Alessandro Armando: Building SMT-Based Software Model Checkers: An Experience Report. 7th International Symposium 5749(1), 1–17 (2009)

[11] Alessandro Armando, Jacopo Mantovani and Lorenzo Platania: Bounded model checking of software using SMT solvers instead of SAT solvers. STTT 11(1), 69–83 (2009)

[12] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya and Marco Roveri: Kratos Homepage. https://es-static.fbk.eu/tools/kratos/ (2010)

[13] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya and Marco Roveri: Verifying SystemC: A software model checking approach. FMCAD pp. 51–59 (2010)

[14] Armin Biere, Edmund M. Clarke and Yunshan Zhu: Bounded Model Checking. Advances in Computers 58(11), 117–148 (2003)

[15] Bruttomesso, R., Pek E., Sharygina N., and Tsitovich A.: The OpenSMT Solver. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) pp. 150–153 (2012)

[16] CBMC: http://www.cprover.org/cbmc/ (2004)

[17] Clark Barrett, Aaron Stump and Cesare Tinelli: SMT-LIB. http://www.smtlib.org/ (2013)

[18] Cytron, R., Ferrante, J.,Rosen,B.K.,Wegman,M.N., and Zadeck, F.K.: An efficient method of computing static single assignment form. POPL pp. 25–35 (1989)

[19] D. Campana, A. Cimatti, I. Narasamdya, and M. Roveri: An Analytic Evaluation of SystemC Encodings in Promela. 18th International SPIN Workshop pp. 90–107 (2011)

[20] Dutertre Bruno: Yices 2.2. Computer-Aided Verification (CAV) 8559, 737–744 (2014)

[21] E. Clarke, K. McMillan, S. Campos, V. Hartonas-Garmhausen: Symbolic model checking. 8th International Conference, CAV '96 pp. 419–422 (1996)

[22] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav: SATABS: SATBased Predicate Abstraction for ANSI-C. TACAS pp. 570–574 (2005)

[23] E. M. Clarke, E. A. Emerson and A. P. Sistla: Automatic verification of finite-state concurrent systems using temporal logic specifications. In ACM Transactions on Programming Languages and Systems pp. 244–246 (1986)

[24] Edmund Clarke, Armin Biere, Richard Raimi, Yunshan Zhu: Bounded Model Checking Using Satisfiability Solving. Formal Methods in System Design 19, 7–34 (2001)

[25] George Necula: CIL. http://kerneis.github.io/cil/ (2001)

[26] Gerard J.Holzmann: The Spin Model Checker: Primer and Reference Manual. Lucent Technologies Inc., Bell Laboratories, Boston, USA (Sep 2003)

[27] Harald Zankl, Aart Middeldorp: Satisfiability of Non-linear (Ir)rational Arithmetic. 16th International Conference, LPAR-16 pp. 481–500 (2010)

[28] Himanshu Jain and Edmund M. Clarke: Efficient SAT Solving for Non-clausal Formulas Using DPLL, Graphs, and Watched Cuts. Proceedings of the 46th Annual Design Automation Conference (6), 563–567 (2009)

[29] Isil Dillig, Thomas Dillig, Kenneth L. McMillan, Alex Aiken: Minimum Satisfying Assignments for SMT. 24th International Conference, CAV pp. 394–409 (2012)

[30] ISO/IEC: Information technology-Portable Operating System Interface (POSIX) Base Specifications. Issue 7, ISO/IEC/IEEE 9945:2009, ISO (2009)

[31] Jeremy Morse, Lucas Cordeiro, Denis Nicole and Bernd Fischer: ESBMC Homepage. http://www.esbmc.org/ (2009)

[32] Jiang Chen, Toshiaki Aoki: Conformance Testing for OSEK/VDX Operating System Using Model Checking. 18th APSEC pp. 274–281 (2011)

[33] Kenro Yatake, Toshiaki Aoki: Automatic Generation of Model Checking Scripts Based on Environment Modeling. 17thl SPIN pp. 58–75 (2010)

[34] Lemieux, J.: Programming in the OSEK/VDX Environment. CMP, Suite 200 Lawrence, KS 66046, USA (2001)

[35] Leonardo de Moura and Grant Passmore: Z3 Homepage. http://z3.codeplex.com/

[36] Libor Waszniowski, Zdenk Hanzlek: Formal verification of multitasking applications based on timed automata model. Real-Time Systems pp. 39–65 (2008)

[37] Lucas Cordeiro and Bernd Fischer: Verifying multi-threaded software using smt-based context-bounded model checking. ICSE' 11 3(9), 331–340 (May 2011)

[38] M. Pradella, A. Morzenti, et al.: Refining Real-Time System Specifications through Bounded Model- and Satisfiability-Checking. ASE' 08 (9), 119–127 (2008)

[39] Mukul R. Prasad, Armin Biere, Aarti Gupta: A survey of recent advances in SAT-based formal verification. International Journal on Software Tools for Technology Transfer 7, 156–173 (2005)

[40] Niklas Een, Niklas S.: An Extensible SAT-solver. SAT pp. 502–518 (2003)

[41] Niklas En, et al.: An Extensible SAT-solver. 6th International Conference, SAT 2003 (502-518)

[42] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, Gennaro Parlato: Lazy-CSeq: A Lazy Sequentialization Tool for C. 20th TACAS pp. 398–401 (2014)

[43] OSEK/VDX Group: OSEK/VDX operating system specification 2.2.3. http://portal.osek-vdx.org/

[44] Robert W. Kramer and Jon C. Martin: nxtOSEK/JSP. http://lejos-osek.sourceforge.net/ (2007)

[45] Sentovich, E. M: A brief study of BDD package performance. In Proceedings of the Formal Methods on Computer-Aided Design pp. 389–403 (1996)

[46] Sergio Rey Calvete: FreeOSEK. http://opensek.sourceforge.net/ (2008)

[47] Shaz Qadeer and Jakob Rehof: Context-bounded model checking of concurrent software. TACAS 2005, LNCS 3440 pp. 93–107 (2005)

[48] supported by Evidence: A free of charge open-source OSEK/VDX Hard Real Time Operating System (RTOS). http://erika.tuxfamily.org/drupal/ (2002)

[49] Tareq Hasan Khan, Ali Habibi, et al.: A Tool for Converting Finite State Machines to SystemC. Technical Report, Concordia University, Canada (2007)

[50] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, Yichen Xie: Zing: A Model Checker for Concurrent Software. 16th International Conference Computer Aided Verification (CAV) pp. 484–487 (2004)

[51] Wachter, B., Kroening, D., Ouaknine, J.: Verifying multi-threaded software with impact. Formal Methods in Computer-Aided Design (FMCAD) pp. 210–217 (2013)

[52] Yanhong Huang, Yongxin Zhao, et al.: Modeling and Verifying the Code-Level OSEK/VDX Operating System with CSP. 5th TASE pp. 142–149 (2011)

[53] Yunja Choi: Safety Analysis of Trampoline OS Using Model Checking: An Experience Report. Software Reliability Engineering (ISSRE) pp. 200–209 (Nov 2011)