Bounded Model Checking for Concurrent Behavior with Scheduler

By Haitao ZHANG

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Toshiaki Aoki

September, 2012

Bounded Model Checking for Concurrent Behavior with Scheduler

By Haitao ZHANG (1010232)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Toshiaki Aoki

and approved by
Associate Professor Toshiaki Aoki
Professor Kokichi Futatsugi
Associate Professor Kazuhiro Ogata and Masato Suzuki

August, 2012 (Submitted)

Contents

List of Figures

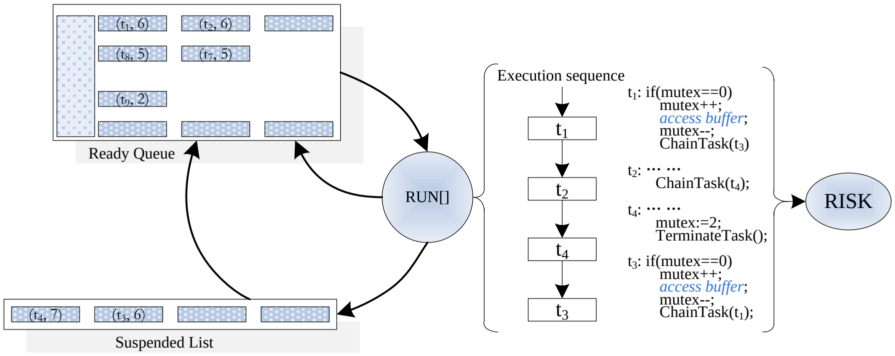
List of Tables

Chapter 1

Introduction

1.1 Motivation

With the advancement of the automobile manufacturing technology, demands for the auxiliary functions of vehicles have also increased sharply and tended to be diversified, which have greatly stimulated the application and development of electronic techniques in automobile industry. However, not all of the electronic parts manufacturers use the same production standard, there exist extremely complex correspondence and cooperation between different electronic parts. OSEK/VDX, as a standard for automobile industry, has been proposed by Germany and France automobile manufacturers and applied in many automobile systems to normalize the correspondence and cooperation. Especially, in OSEK/VDX OS, which task to be run is determined by scheduler, in addition, tasks can send service commands to request scheduler for responding to its particular behaviors, such as terminating itself, activating a task and chaining a task. Thus, there may exist a potential risk which is caused by a unreasonable dispatching, as we can see in figure 1.1, the unreasonable execution sequence t_1, t_2, t_4, t_3 which is conducted by scheduler with service commands leads to $task_1$ and $task_3$ cannot use the semaphore *mutex* to access a global *buffer*. Consequently, how to check the safety property of a multi-task software based on automobile OSEK/VDX OS has become very difficult and crucial.



```
t1: if(mutex==0)
      mutex++;
      access buffer;
      mutex--;
      ChainTask(t3)

t2: … …
     ChainTask(t4);

t4: … …
     mutex:=2;
     TerminateTask();

t3: if(mutex==0)
      mutex++;
      access buffer;
      mutex--;
      ChainTask(t1);
```

Figure 1.1: Example for an unreasonable dispatching sequence

Model checking [1, 2] as a traditional technique, has been applied to checking multi-task software, however, this algorithm suffers from combinatorial state space explosion when verifies complex multi-task software. Recently, a new technique called bounded model checking (BMC) [3, 4] has been proposed to overcome the state explosion problem and has been successfully applied to verifying the multi-task software. Based on BMC, there are many efficient and reliable techniques [5, 6, 7, 8, 9] have been developed and applied in the verification of general multi-task software, however, these techniques just focus on the tasks' current behaviors, the scheduler's behaviors are not considered in verification process. Therefore, these techniques are not able to check the safety property of a software in which tasks are dispatched by a scheduler.

1.2 Thesis purpose and outline

In our article, we propose an approach to check the safety property of multi-task software in which tasks are dispatched by fixed priority scheduler (FPS) based on OSEK/VDX OS [10]. In order to accomplish our research purpose, firstly, we analyze the dispatching behaviors of FPS based on OSEK/VDX OS, and then we use an extended finite state machine to establish a model for FPS and describe tasks' behaviors. Especially, our FPS model can respond to three types of service commands which are sent by tasks in order to realize tasks particular requests, such as terminating a task, activating a task and chaining a task. Furthermore, as to obtain the execution paths of tasks that are dispatched by FPS, we establish a k-step execution tree to represent all of the possible execution paths and propose two strategies to extract execution paths in which BMC is employed to generate the verification conditions (VCs) based on our execution tree. In addition, Yices [12], which is satisfiability modulo theories (SMT) solver and capable of handling large and propositionally complex formulas in a rich combination of theories is used to check the generated VCs with verification property formula and return the verification results. Finally, we implement two types of tools according to our two strategies of extracting execution paths based on execution tree to evaluate our approach. Using our tools, we can directly get the k-step transition system M which is composed of each execution paths VCs based on FPS dispatching, furthermore, the k-step transition system M can be translated into Yices file with our tools. We also carry out some relevant experiments with our tools, results show that our approach can efficiently check the safety property of multi-task software in which tasks are dispatched by FPS.

1.3 Thesis structure

Our article is structured as follows. In chapter 2, we give out some technical background of model checking, bounded model checking and SMT solver Yices. In chapter 3, we review the history of OSEK/VDX, besides, operating system standard of OSEK/VDX is also presented in this chapter. Based on the analysis of chapter 3, in chapter 4, we establish a model for fixed priority scheduler with extended finite state machine. In order to model and obtain all of the tasks' execution paths under FPS's dispatching, in chapter 5, we illustrate a new approach to represent and gain all of the execution paths based on a execution tree in which BMC is employed to generate the verification conditions for a transition system M. Then the implementation processes of our approach is shown in chapter 6. As to evaluate our approach, in the chapter 7, we carry out some relevant experiments with our tools, the experiments results are also shown in this part. In the last two chapters of our article, we firstly summary our approach and talk about our future work based on current work in chapter 8. Finally, we discuss some related work for our research in chapter 9.

Chapter 2

Background

2.1 Model checking

Generally, in software and hardware design of complex systems, most of the time and effort are spent on verification other than construction. Techniques are sought to reduce and ease the verification efforts while increasing their coverage. Formal methods [11] offer a large potential to obtain an early integration of verification in the design process, to provide more effective verification techniques, and to reduce the verification time. Model checking as a traditional verification technique, it is a formal verification technique and based on models to describe a possible system behaviors in a mathematically precise and unambiguous manner and has three fundamental features.

1. Automatic checking
   It does not rely on complicated interaction with the user for incremental property proving. If a property does not hold, the model checker generates a counterexample trace automatically.

2. Finite states
   The systems being checked are assumed to be finite. Typical examples of finite systems, for which model checking has successful been applied, are digital sequential circuits and communication protocols.

3. System properties specified by temporal logic
   The system which to be checked whose verification properties are specified by temporal logic. Thus, model checking can be summarized as an algorithmic technique for checking temporal properties of finite systems.

Usually, independent of the concrete design language, *finite state machine* or *Kripke structure* [13] is employed to describe a system's behaviors. In our article, we adopt *finite state machine* to describe systems behaviors. In the following part, firstly, we will talk about the general notion of finite state machine, and then a specific definition of finite state machine will be defined for our research.

2.1.1 General notion of finite state machine

Model checking is a technique for the automated verification of finite state-based systems. The proof of a property is entirely carried out by the machine. In case the property does not hold, the model checker will construct a counter-example suitable for failure diagnosis. In mathematical terms, the considered systems are represented as finite state-based transition graphs (finite state machine, FSM). A *finite State machine* consists of a finite set of states, a set of initial states (a subset of the set of states), a transition relation (states are accessible from the current state), a function mapping each state to the atomic propositions holding in this state. Especially, we use an example to illustrate the process about how to use *finite State machine* to construct a model for a system. For instance, a simple cruise control system [14] (figure 2.1) has several characteristics, e.g., it is controlled by three buttons: *resume*, *on*, *off*. When the engine is running and *on* is pressed, the cruise control system records the current speed and maintains the car at this speed. When the accelerator, brake or *off* is pressed, the cruise control system disengages but retains the speed setting. If resume is pressed, the system accelerator or de-accelerator forces the car back to the previously recorded speed.

Figure 2.1: A simple cruise control system

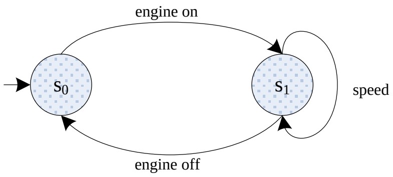


Figure 2.2: The state machine for a simple cruise control system

We can model the various process of the system as state machines according to the description of a simple cruise control system. A state machine for the process responsible for obtaining the current speed is given in figure 2.2. Starting from state s_0, it indicates that once the engine is switched on, it transits to state s_1 and can then repeatedly obtain a speed reading until the engine is switched off, when it returns to state s_0.

2.1.2 A specific definition of finite state machine in our thesis

In our article, we define a specific finite state machine for describing tasks' and scheduler behaviors based on general notion of finite state machine. In our article, a finite state machine M is a tuple,

Definition 1 (*Finite state machine*) a finite state machine M=(S, s_0, T, V), where S is the finite set of states, s_0 is initial state, V is finite set of variables and V=V_{global} $\cup \bigcup u_j$, (where V_{global} is the set of global variables, u_j is one of local variables of a task and j is index of local variables), $T \subseteq S \times C \times A \times S$ is transition relation set.

In the transition relation set T, C is the set of guard functions and $c \in C$ is a Boolean expression, the grammar of c is defined as follows:

$$c ::= \mathit{true} | v_i \sim v_j | c \vee c | c \wedge c \tag{2.1}$$

Here, the symbol $\sim \in \{==, >, \geq, <, \leq, \neq\}$, i, j are index of variables and $v_i, v_j \in V$.

In addition, A is a finite set of actions, let $a \in A$ be an action which is a calculation expression, we use $a(v)$ to represent a calculation over a variable $v \in V$. The grammar of $a(v)$ is defined as follows:

$$a(v) ::= < v >< eqs >< cxep >; \tag{2.2}$$

$$cxep ::= v_i | v_i \circ v_j | v_i \circ z | z \circ v_i | cxep \circ cxep; \tag{2.3}$$

$$eqs ::= \text{“} := \text{”}; \tag{2.4}$$

Here, the symbol $\circ \in \{+, -, *, /\}$ and $z \in Z$ (Z is integer set).

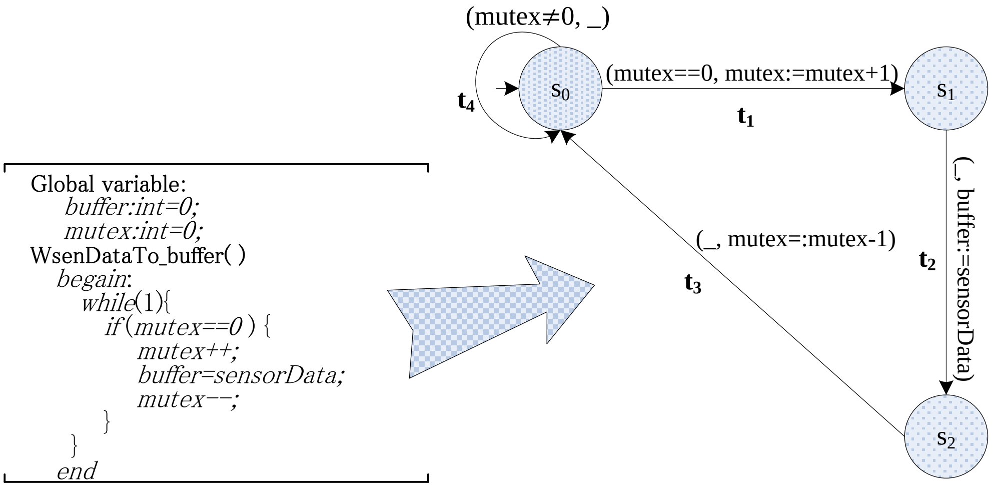
Each element t of T is denoted by $t(s, c, a, s')$, where s is source state, s' is target state. If and only if a transition relation t whose guard function c is *true*, the transition relation t can be executed, otherwise *finite state machine* will stay in current states s. Furthermore, the action a can be performed if and only if the transition relation t is executed, we use formula 2.5 and 2.6 to illustrate the calculating process of action $a(v)$ over variables set V.

$$\exists v \in V, v' := a(v) \tag{2.5}$$

$$\forall v_{other} \in V \setminus \{v\}, v^{'}_{other} := v_{other} \tag{2.6}$$

Note that in above definition, for a transition $t \in T$, if $c \in C$ or $a \in A$ is null, we use symbol “_” instead of it, similarly, if both $c \in C$ and $a \in A$ are null, we use symbol “_” instead both of them. For the case of the guard function c, if c is equal to “_”, it means that c is true, if “_” appears in the position of a, it means there is no action.

Example 1 Figure 2.3 shows a *finite state machine* that consists of three states: s_0, s_1 and s_2, where s_0 is initial state. According to the *definition 1*, we can infer that variable set $V = \{buffer, mutex, sensorData\}$ and transition relation set $T = \{t_1, t_2, t_3, t_4\}$. Intuitively, this example models a system which starts from s_0 and moves to s_1 with the guard function “$mutex == 0$”, simultaneously, the transition from s_0 to s_1 can also cause the action $a(mutex) : mutex := mutex + 1$ to be performed. Since the guard function in transition relation t_2 is “_”, which indicates the guard function is true, the transition relation $t_2(s_1, \_, buffer := sensorData, s_2)$ can be executed and the action $a(buffer) : buffer := sensorData$ is performed. The action $a(mutex) : mutex := mutex - 1$ also can be performed after the transition relation t_3 is executed. Particularly, the transition relation t_4 is used to indicate the negative case of the guard function “$mutex == 0$”.



```
Global variable:
    buffer:int=0;
    mutex:int=0;
WsenDataTo_buffer( )
  begain:
    while(1){
       if(mutex==0){
          mutex++;
          buffer=sensorData;
          mutex−−;
       }
    }
  end
```

Figure 2.3: A finite state machine

Usually, since a multi-task software includes a lots tasks and these tasks can communicate with each other using synchronization event, in our research, each task which includes synchronization events will be described as an *extended finite state machine* and the whole software is modeled as an combinatorial *extended finite state machine* via synchronization events. We named the *finite state machine* which includes synchronization events as *E-FSM*, the definition of *E-FSM* is as follows:

Definition 2 *(E-FSM)* An *E-FSM* $M_i = (S_i, S_i^0, T_i, E_i, V_i)$, where i means the index of task, S_i, s_i^0 and V_i are same as *definition 1*, E_i denotes the finite set of events, an event $e \in E$ is a synchronization event with suffix "?" or "!" (where, symbol "!" represents request and symbol "?" represents respondence), $T \subseteq S \times C \times E \times A \times S$ is transition relations set.

In the transition relations set T, the definitions of C and A are the same as *definition 1*, we also use $t(s, c, e, a, s')$ to represent each element t of T. The difference between $t(s, c, a, s')$ and $t(s, c, e, a, s')$ is transition relation $t(s, c, e, a, s')$ includes a synchronization event $e \in E$. If a transition relation t from source state s goto target state s' (this transition relation can be executed iff the c is true), both e and a will be performed.

In order to describe the whole execution behaviors of a software in which tasks use synchronization events to synchronization execution, we use combinatorial *E-FSM* to represent the combination of all tasks *E-FSM*, the definition of combinatorial *E-FSM* is as follows:

Definition 3 (*Combinatorial E-FSM*) A *combinatorial E-FSM* of m *E-FSMs* is a *extended finite state machine* $CM = (S, S^0, T, E, V)$, where $S = \bigcup_{i=1}^{m} S_i$, $s_0 = (s_1^0, s_2^0, ..., s_m^0)$, $V = \bigcup_{i=1}^{m} V_i$, each global transition relation $t \in T$, such as,

$$t((s_1^0, s_2^0, ..., s_m^0), c, e, a, (s_1^{'}, s_2^{'}, ..., s_m^{'})) \tag{2.7}$$

Here, $c = \bigwedge_{i=1}^{m} c_i$, $c = \bigcup_{i=1}^{m} a_i$ and $e = e_i?||e_j!$ (where, i, j are index of translation relation of task and symbol "||" means an enabled synchronization event). Especially,

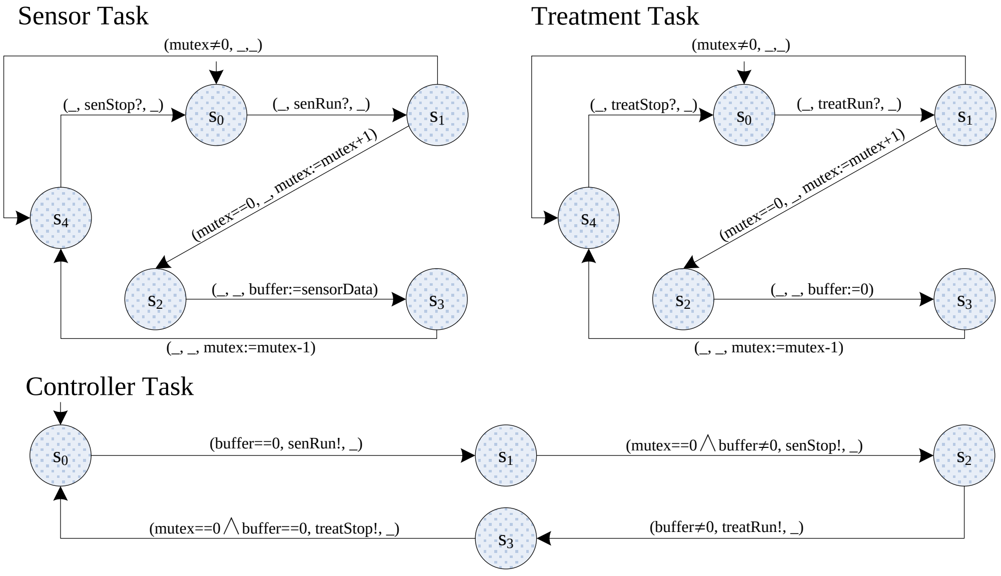


Figure 2.4: Extended finite state machine for sensor task, treatment task and controller task

$\forall s_i \in S, s_i^{'} = s_j$ iff when the translation relation $t(s_i, c, e, a, s_j)$ can be executed, for the case of $\forall s_r \in S \setminus \{s_i\}$, $s_r^{'} = s_r$.

Note that for a transition relation t, if it cannot be synchronously executed, it will remain in the same location when a synchronization transition relation is performed.

Example 2 The system consists of three *E-FSMs*: *sensor* task, *treatment* task and *controller* task. These three tasks' *E-FSMs* are shown in figure 2.4. As an example of *combinatorial E-FSM*, *controller* task is used to control *sensor* task and *treatment* task with synchronization events: *senRun*, *senStop*, *treatRun* and *treatStop*. In our example, *sensor* task and *treatment* task use a mutual semaphores *mutex* to mutually access the global *buffer*. The whole system which is modeled by these three tasks with synchronization events is illustrated in figure 2.5. As we can see, in figure 2.5, the transition relation $t_1((0,0,0), buffer == 0, senRun, \_, (1,0,1))$ is a synchronization transition between *sensor* task and *controller* task with synchronization event *senRun*, where the (s_i, s_j, s_r) is a states tuple. Obviously, in the states tuple, the first bit represents *sensor* task's current state, the second bit represents *treatment* task's current state and last bit represents *controller* task's current state.

Usually, the purpose of constructing a model for a system is to verify whether the system satisfies verification properties formulae or not. For this purpose, in our article, we use verification properties to search or detect all of the possible execution paths which exist in a system model. Each path π in a system model M is a sequence of transition relation $\pi = \{t_1, t_2, ..., t_n\}$, e.g., in the figure 2.5, the system consists of four paths: $\pi_1 = \{t_1, t_2, t_4, t_5, t_6, t_7, t_9, t_{10}\}$, $\pi_2 = \{t_1, t_2, t_4, t_5, t_6, t_8, t_{10}\}$, $\pi_3 = \{t_1, t_3, t_5, t_6, t_7, t_9, t_{10}\}$, $\pi_4 = \{t_1, t_3, t_5, t_6, t_8, t_{10}\}$, for each transition relation $t_i \in T$, $0 \leq i \leq |\pi|$ ($|\pi|$ is length of path π which can be either finite or infinite). Furthermore, in model checking, in order to capture nesting and mutual dependency of properties, temporal logic [15, 16, 17] is used

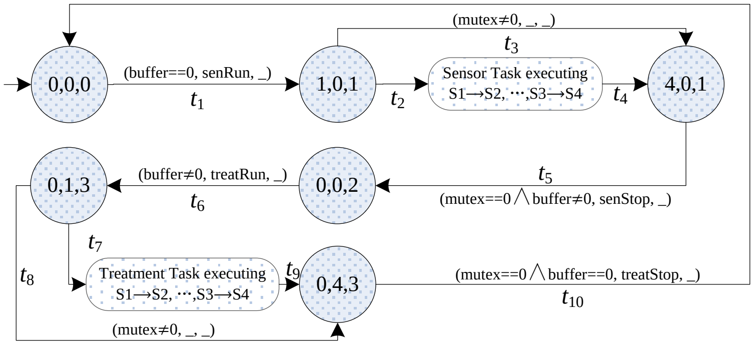


Figure 2.5: The whole system which is modeled by three tasks with synchronization events

as a specification language for verification property.

Temporal logic is an extension of classical logic, we concentrate on propositional linear temporal logic (PLTL, or LTL for short) as an extension of propositional logic in our article. Propositional logic LTL inherits Boolean variables and Boolean operators such as negation $\neq$, conjunction $\wedge$, implication $\rightarrow$, and so on. In addition, for the connection operators, LTL has temporal operators, such as *next* time operator $\boldsymbol{X}$, *globally* operator $\boldsymbol{G}$, simplest *liveness* operator $\boldsymbol{F}$, binary temporal operators *until* ($\boldsymbol{U}$) and *release* ($\boldsymbol{R}$).

Model checking, as a traditional technique, although has been successfully applied to checking multi-task software, will suffer from combinatorial state space explosion when verifys a multi-task software. Recently, a new technique called bounded model checking (BMC) has been proposed to overcome the state explosion problem and successfully applied to verify the multi-task software. In next part we will talk about this new technique BMC.

2.2 Bounded model checking

Bounded model checking (BMC) is a SAT-based [18, 19] technique for symbolic model checking. Compared to BDD-based [20, 21] model checking, it offers the advantage of handling the verification of large state spaces, albeit for a smaller fragment of the language. The main idea of BMC is to avoid the full state space generation and look for witnesses of an existential specification on suitable subsets of the full model. Once a sub-model is selected, the formula to be checked as well as the considered sub-model to be translated into propositional formulae and a propositional satisfiability problem are solved via a specialized SAT solvers. If the test is positive, the specification will hold on the sub-model as well as the whole model and give the checked particular existential syntax, Otherwise, a larger sub-model will be selected and the whole procedure will be run again.

In bounded model checking, we usually construct a Boolean formula that is satisfiable if and only if the underlying state transition system can realize a finite sequence of state transitions that reaches certain states of interest. If such a path segment cannot be found at a given length k, the search is continued for larger k. The procedure is symbolic, i.e., symbolic Boolean variables are utilized. Thus, when a check is done for a specific path segment of length k, all path segments of length k are being examined. The Boolean formula that is formed is given to a satisfiability solving program and if a satisfying assignment is found, the assignment is a witness for the path segment of interest.

There are several advantages of bounded model checking. SAT tools [22], e.g., PROVER [23], SATO [24] and GRASP [25], do not require exponential space and large designs can be checked very fast, since the state space is searched in an arbitrary order. BDD based model checking usually operates in breadth first search consuming much more memory. Further, the procedure is able to find paths of minimal length, which helps the user understand the examples that are generated. Lastly, the SAT tools generally need far less by hand manipulation than BDDs. Usually the default case splitting heuristics are sufficient. However, although there have been attempts to extend SAT-based BMC to the verification of multi-task software, the main challenge remains the classical state space explosion in which the number of tasks grows exponentially. An important observation is that checking the transition system which is generated by BMC according to a verification system with SMT [27] instead of SAT. The advantages of SMT compares with SAT is SMT not only can check more VCs of a transition system and faster than SAT but also SMT solvers produce unsatisfiable cores that allow us to remove logic that is not relevant to a given property. Especially, the propositional formula created by BMC is formed as follows:

- a transition system M
- a temporal logic formula f
- a user-supplied bound k

We construct a propositional formula $[[M, f]]_k$ which will be satisfiable if and only if the formula f is valid along one of execution path of M. We form the formula $[[M, f]]_k$ on state transition system M, bound k and formula f without rolled transition relation is as follows:

$$[[M, f]]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge [[\neg f]]_k \tag{2.8}$$

Where $I(s_0)$ is the characteristic function of the set of initial states, and $T(s_i, s_{i+1})$ is the characteristic function of the transition relation. $[[\neg f]]_k$ is a formula that will be true if and only if the formula $\neg f$ is valid along a path of length k. For the formula f which is expressed by LTL, we can use following translation processes to translate verification formula f into BMC's verification semantic formula.

- $[[p]]_k^i := p(s_i)$
- $[[\neg p]]_k^i := \neg p(s_i)$
- $[[f \vee g]]_k^i := [[f]]_k^i \vee [[g]]_k^i$
- $[[f \wedge g]]_k^i := [[f]]_k^i \wedge [[g]]_k^i$
- $[[\boldsymbol{G} f]]_k^i := [[f]]_k^i \wedge [[\boldsymbol{G} f]]_k^{i+1}$
- $[[\boldsymbol{F} f]]_k^i := [[f]]_k^i \vee [[\boldsymbol{F} f]]_k^{i+1}$

- $[[f\boldsymbol{U}g]]_k^i := [[g]]_k^i \vee ([[f]]_k^i \wedge [[f\boldsymbol{U}g]]_k^{i+1})$
- $[[f\boldsymbol{R}g]]_k^i := [[g]]_k^i \vee ([[f]]_k^i \wedge [[f\boldsymbol{R}g]]_k^{i+1})$
- $[[\boldsymbol{X}f]]_k^i := [[f]]_k^{i+1}$

Note that the base case of above translation processes is:

$$[[f]]_k^{k+1} := 0 \tag{2.9}$$

Example 3 In order to show the process about how to use BMC to check a system, we illustrate an example based on *example 2*. In our example, we set bound k to be 2, and the verification formula we used f is as follows:

$$\boldsymbol{G}(\mathit{buffer} == 0 \vee \mathit{buffer} == 1) \tag{2.10}$$

In order to get all of the VCs of transition system M under bound k according to *example 2*, we only use $c \wedge a$ which exist in a transition relation $t \in T$ to represent each transition relation t of a path π, where $c \in C$ and $a \in A$. The VCs ϕ_i of each transition relation t is as follows:

$\phi_0 : T(s_0, s_1) \longmapsto (buffer_0 == 0) \wedge (buffer_1 := buffer_0 \wedge mutex_1 := mutex_0 \wedge \mathit{sensorData}_1 := \mathit{sensorData}_0)$

$\phi_1 : T(s_0, s_1) \longmapsto [(mutex_1 == 0) \wedge (buffer_2 := buffer_1 \wedge mutex_2 := mutex_1 + 1 \wedge \mathit{sensorData}_2 := \mathit{sensorData}_1)] \vee [(mutex_1 \neq 0) \wedge (buffer_2 := buffer_1 \wedge mutex_2 := mutex_1 \wedge \mathit{sensorData}_2 := \mathit{sensorData}_1)]$

Hence, we can obtain the transition system M under bound k with ϕ_0 and ϕ_1, the transition system M is as follows:

$$[[M]]_2 := I(s_0) \wedge \bigwedge_{i=0}^{1} \phi_i \tag{2.11}$$

In addition, we can translate the verification formula f into BMC's verification semantic formula according to the translation processes of verification formula f, the BMC's semantic formula f is as follows:

$$[[\neg f]]_2 := \neg(p(s_0) \wedge (p(s_1) \wedge (p(s_2)) \tag{2.12}$$

Here, the symbol $p(s_i)$ means checking verification formula f in k-step VCs of a transition system. Each $p(s_i)$ is as follows:

$\phi_0 : p(s_0) \longmapsto (buffer_0 == 0 \vee buffer_0 == 1)$
$\phi_1 : p(s_1) \longmapsto (buffer_1 == 0 \vee buffer_1 == 1)$
$\phi_2 : p(s_2) \longmapsto (buffer_2 == 0 \vee buffer_2 == 1)$

Thus we can get the combinatorial VCs of $[[M, f]]_k$ according to the formula (2.8), the combinatorial VCs is as follows:

$$[[M, f]]_2 := I(s_0) \wedge \bigwedge_{i=0}^{1} \phi_i \wedge (\neg(\bigwedge_{j=0}^{2} \phi_j)) \tag{2.13}$$

In order to get the verification results, we translate the combinatorial VCs of $[[M, f]]_k$ into Yices file and use Yices to return the verification results in our article. In next part, we will talk about SMT solver Yices.

2.3 SMT solver Yices

Yices is an SMT [28] solver developed at SRI International and a capable solver for handling large and propositionally complex formulas in a rich combination of theories as well as it can be downloaded free of charge at http://yices.csl.sri.com/. Especially, Yices integrates an efficient SAT solver based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm with specialized theory solvers that handle the first-order theories, a core theory solver handles equalities and uninterpreted functions. It is complemented by satellite solvers for other theories such as arithmetic, bit vectors, array and particular data-type. The main components of Yices are depicted in figure 2.6.

In Yices basic use, Yices reads a verification problem file which is described by Yices language and checks whether the verification problem is satisfiable or not. Then, Yices can output the verification result. Although Yices has its own input language, it also accepts specifications written in the SMT-LIB notation and supports all the theories currently defined in SMT-LIB [29]. In our article, Yices is adopted as our solver to check the combinatorial VCs and return the verification result. Especially, we use an example to show the process about how to use Yices to check a combinatorial VCs of $[[M, f]]_k$ in this part.

Example 4 In the *example 3*, we have obtained the combinatorial VCs of $[[M, f]]_k$ based on the *example 2* with BMC. In the formula (2.13), function $I(s_0)$ is used to indicate variables initial values. In our example, we set the initial values of global variable *buffer* and *mutex* as 0 respectively, and the initial value of local variable *sensorData* of *Sensor Task* as 1. Hence, we can use the following codes which are described by Yices language to represent the function $I(s_0)$.

Part 1: function $I(s_0)$
(*define buffer0::int*) (*define mutex0::int*) (*define sensorData0::int*)
(*assert* (*and* (= *buffer0* 0) (=*mutex0* 0) (= *sensorData0* 1)))

For the $\varphi_0 : T(s_0, s_1)$ and $\varphi_1 : T(s_1, s_2)$:

Part 2: $\varphi_0 : T(s_0, s_1)$ and $\varphi_1 : T(s_1, s_2)$
(*define buffer1::int*) (*define mutex1::int*) (*define sensorData1::int*) ;;*k*=1

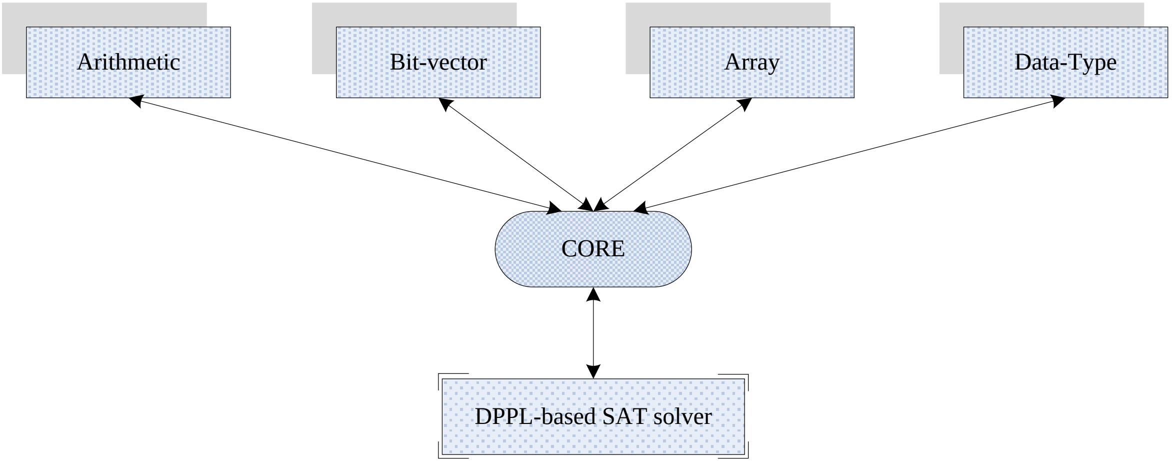


Figure 2.6: Yices architecture

(*assert* (*and* (= *buffer0* 0) (= *buffer1* *buffer0*) (= *mutex1* *mutex0*) (= *sensorData1* *sensorData0*)))
(*define* *buffer2::int*) (*define* *mutex2::int*) (*define* *sensorData2::int*) ;;*k*=2
(*assert* (*or* (*and* (= *mutex1* 0) (= *buffer2* *buffer1*) (= *mutex2* (+ *mutex1* 1))...) (*and* (/= *mutex1* 0) (= *buffer2* *buffer1*) (= *mutex2* *mutex1*)...))

For the verification formulae ϕ_0, ϕ_1 and ϕ_2:

Part 3: verification formula
(*assert* (*not* (*and* (*or* (= *buffer0* 0) (= *buffer0* 1)) (*or* (= *buffer1* 0) (= *buffer1* 1)) (*or* (= *buffer2* 0) (= *buffer2* 1)))))

We can get the Yices file which is combined by *part 1*, *part 2* and *part 3* for combinatorial VCs of $[[M, f]]_k$, then we can use Yices to check this file and return verification result. (For more details about how to use Yices to check a system, visit the Yices website: http://yices.csl.sri.com/.)

Chapter 3

OSEK/VDX

3.1 History of OSEK/VDX

OSEK, a German acronym for *Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug*, which translates roughly to open system and their corresponding interfaces for automotive controllers, was initiated in Germany in May 1993 by the automobile companies BMW, Daimler Benz, Opel and Volkswagen; the major automotive suppliers Bosch and Siemens; and the Institute of Industrial Information Technology at the university of Karlsruhe, German. VDX, an acronym for Vehicle Distributed eXecutive, was initiated in France about the same time by the French automotive companies PSA and Renault. In 1994, the two consortia merged to form the OSEK/VDX consortium and created the OSEK/VDX steering committee. Since that time, other companies have joined as members of the Technical Committee to assist in developing the technical standard.

The original motivation for the standers was to resolve the problems of increasing software content in automobiles, duplication of effort in the areas of operating systems and communication networks, lack of qualified software engineers, and a desire for high-quality products. The goal was to develop a standard API that could reduce the amount of duplicated effort and increase the amount of code reuse within the vehicle. The results were the four standards in existence today: Operation System (OS), communication (COM), Network Management (NM) and OSEK/VDX Implementation language (OIL). Although originally intended for the automobile environment, the specifications have been carefully developed to meet the requirements of a small embedded system with inter-processor communication.

3.2 Operating system standard of OSEK/VDX

3.2.1 Task

Tasks within the OSEK/VDX have a number of attributes that affect both the operation of the system and the size of the code. A task, either basic or extended, has a statically defined priority, might or might not be preemptive, and might be able to suspend execution while waiting for an event. The combination of these and other attributes creates a conformance class, as defined in OSEK/VDX specification.

In our research, we focus on the basic task which has three types of state: *running* state, *suspended* state and *ready* state. A basic task runs to completion unless preempted by a higher priority task or an interruption. The task, through other API service, can disable preemption and interruption. Lower priority tasks are inhibited while a basic of higher priority runs; however, other tasks of the same priority are also inhibited. The OSEK/VDX does not allow round-robin scheduling of tasks at the same priority level, as is found in some larger system. For time-sliced scheduling, it would make the systems performance impossible to predict, which is undesirable in a real-time safety critical environment. Furthermore, a basic task can send three types of service commands to scheduler for terminating itself, activating a task or chaining a task. These three types service commands are as follow:

- TerminateTask: Terminate a task which is in *running* state, the task will be dispatched from *running* state into *suspended* state by scheduler;
- ActivateTask: Activate a task which is in *suspended* state by scheduler. If the task is activated, its state will be changed to *ready* state;
- ChainTask: If scheduler receives this service command which is sent by a *running* task, scheduler will activate a task and then terminate the *running* task;

Since OSEK/VDX is a statically defined OS, all tasks must be defined at compile time. When the OS starts, a basic task whether will be in the suspended or ready state depends on the configure file. If a task is defined as an *autostart* task, then it starts in the ready state; otherwise, it starts in the suspended state.

3.2.2 Priority

As with any ROTS, tasks in an OSEK/VDX OS have a priority, which is statically defined and cannot be changed dynamically by the application and 0 is the lowest priority, and no maximum is defined in the specification. Especially, OSEK/VDX allows several tasks share a same level priority.

3.2.3 Scheduling

In OSEK/VDX, task switch within an OSEK/VDX is performed by fixed priority scheduler (FPS). Furthermore, fixed priority scheduler uses one of three possible policies to dispatch task, such as non-preemptive, fully preemptive or mixed preemptive. In order to manage configure files and states of tasks, a *ready queue*, a *suspended list* and a *running unit* are used as store structures to save tasks' configure files according to tasks' states in scheduler. In the dispatching process, which task to be run is determined by scheduler according to the priority and state of task. If and only if a task whose state is *ready* and priority is higher than other *ready* state' tasks', the task will be dispatched by scheduler to occupy the *running unit* for running. Particularly, in order to incarnate several tasks that share the same level priority, in the *ready queue*, each priority has a *queue* structure which is used as store structure for saving tasks' configure files, the structure of *ready queue* is shown in figure 3.1. We can see in the figure 3.1, $task_1$, $task_3$ and $task_7$ share the same level priority 6. In addition, tasks can send service commands to call scheduler for terminating itself, activating a task and chaining a task. Especially, we use an example to illustrate the scheduling behaviors of FPS.

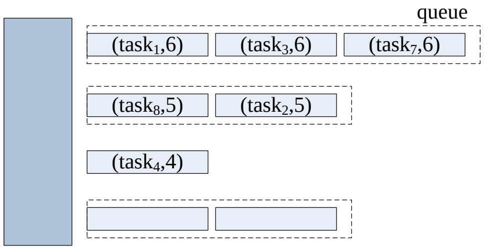
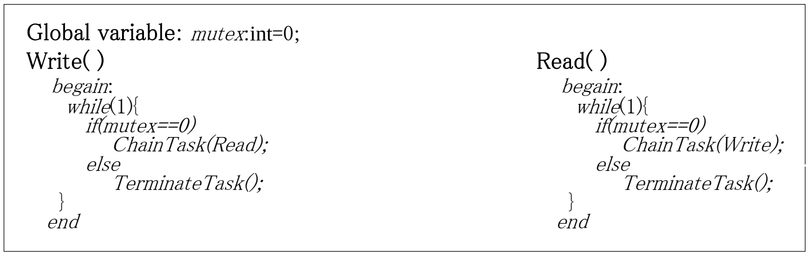


Figure 3.1: The state machine for a simple cruise control system



```
Global variable: mutex:int=0;
Write( )                                Read( )
  begain:                                 begain:
    while(1){                               while(1){
      if(mutex==0)                            if(mutex==0)
          ChainTask(Read);                        ChainTask(Write);
      else                                    else
          TerminateTask();                        TerminateTask();
    }                                       }
  end                                     end
```

Figure 3.2: The behaviors of *Write* task and *Read* task

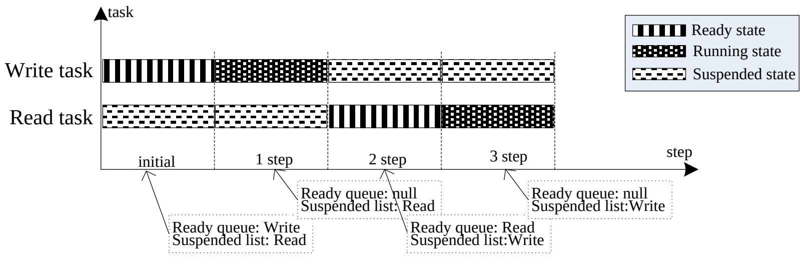


Figure 3.3: Tasks' execution behaviors which are dispatched by FPS

Example 5 The system consists of two tasks: *Write* task and *Read* task. The behaviors of *Write* task and *Read* task are illustrated in figure 3.2. In initial state, we assume *Write* task is in the *ready queue* but *Read* task is in the *suspended list*. Furthermore, we set the priorities of *Write* task and *Read* task to be 3 and 2 respectively. According to the above setting, we can know *Write* task is dispatched to run by FPS in the first step. Since the initial value of variable *mutex* is equal to 0, the service command "ChainTask(Read)" will be sent to FPS. Once FPS has received this service command and responded it, then in the second step, we can see *Write* task is dispatched into *suspended list*, *Read* task is dispatched into *ready queue*. At this moment the *running unit* is idle, hence, we can find that the *Read* task is dispatched to run in third step. All of the tasks' execution behaviors which are dispatched by FPS are illustrated in figure 3.3.

Chapter 4

FPS model

In the chapter 3, we have already introduced the history and operating system standard of OSEK/VDX OS. In this chapter, we will firstly talk about the analysis process of FPS based on OSEK/VDX OS. According to the analysis process of FPS, in the second part, we will establish a model for FPS with extended finite state machine. Since tasks and scheduler can communicate with each other using service commands in our research, in the last part, we will show the method about how to process a service command appearing in the task's transition relation.

4.1 Analysis FPS based on OSEK/VDX

In OSEK/VDX OS, a task whose type is either basic or extended has a statically defined priority. In our research, we establish FPS model based on the basic task and use some key attributes to indicate a basic task's configure file. These key attributes are abstracted by a conformance class which is named as *TaskConfigure*. The structure of configure file for basic task is as follows:

```
TaskConfigure {
  TaskName::string;
  TaskPriority::int;
  Autostart::bool;
};
```

The particular meanings of the above parameters in *TaskConfigure* are as follow:

TaskName: the name of task, and each task has a unique name;
TaskPriority: the priority of the task;
Autostart: set to be either *true* or *false*. *Autostart* defines whether the task can be moved into ready state automatically.

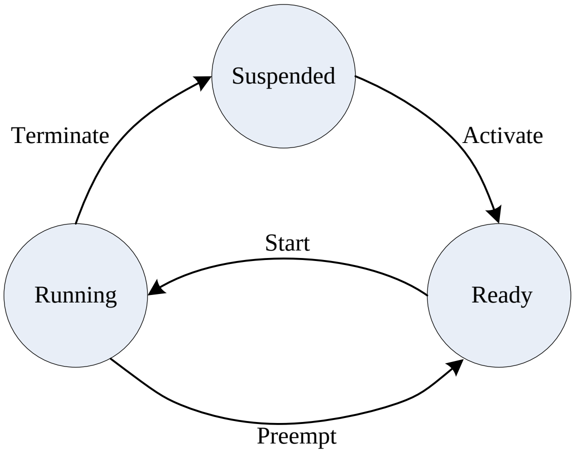


Figure 4.1: State transition diagram of a basic task

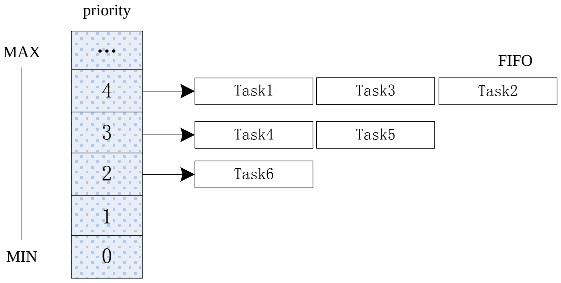


Figure 4.2: The store structure of ready queue

In OSEK/VDX OS, a basic task usually exists in one of three states [30] *Ready*, *Running*, and *Suspended*. Transitions between states occur within four possible events: *Activate*, *Start*, *Preempt* and *Terminate*. The state transition diagram of a basic task is shown in figure 4.1.

Task switching within OSEK/VDX OS [31] is performed by a scheduler using one of three possible policies: non-preemptive, full preemptive or mixed preempt. In our research we focus on the full-preemptive policy. In order to present the policy more clearly, we firstly create three types of store structures for *ready* state, *running* state and *suspended* state.

OSEK/VDX OS permits several basic tasks to share the same priority by storing them in a special queue structure which is named as *readyQueue* in our article. The structure of ready queue is illustrated in the figure 4.2.

We use a class (C++ language) to realize the *readyQueue* and its functions. The definition of the class is as follows:

```
ReadyQueue{
public:
   void Initial();
   bool Empty();
```

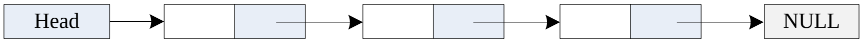


Figure 4.3: The store structure of suspended list

```
    bool Full(int priority);
    TaskConfigure Dequeue();
    bool Enqueue(TaskConfigure enTask);
    int RetuFirPrio();
private:
    TaskConfigure element[PriorityRegion][TaskAmount];
};
```

In order to store the task in running state, we define a variable *runTask* as its store structure, that is:

```
TaskConfigure runTask;
```

Similarly, in order to store the tasks in suspended state we define a list structure, the list structure is shown in figure 4.3. We also use a class to realize this structure and its functions. The class of suspended list is described as follows:

```
SuspendList{
public:
    void Initial();
    TaskConfigure DeleteList(string TaskName);
    bool InsertList(TaskConfigure InTask);
    bool SearchList(string TaskName);
private:
    TaskConfigure element;
    SuspendList *next;
};
```

In full-preemptive scheduler policy of the OSEK/VDX OS, the scheduler is only executed when one of following five events occurs.

- The *runTask* is empty;
- The priority of *runTask* is lower than the first element of the *readyQueue*;
- A task in *runTask* sends service command "*TerminateTask*()" to scheduler for terminating itself;
- The task in runTask sends service command "*ActivateTask*(*callTask*)" to scheduler for calling another task (if the *callTask* in the *suspendList*, it will be moved into *readyQueue* from *suspendList* by scheduler);

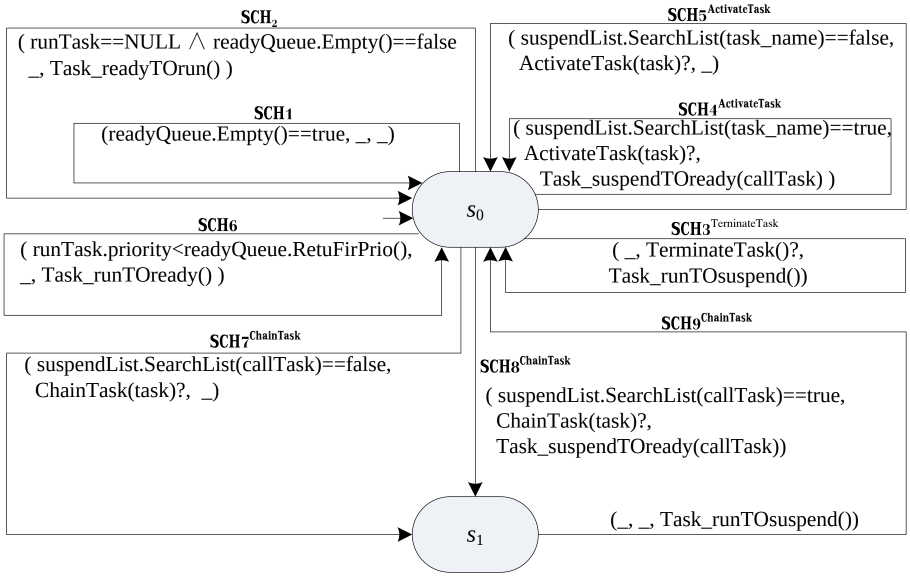

SCH2
(runTask==NULL ∧ readyQueue.Empty()==false
_, Task_readyTOrun())
SCH1
(readyQueue.Empty()==true, _, _)
SCH5ActivateTask
(suspendList.SearchList(task_name)==false,
ActivateTask(task)?, _)
SCH4ActivateTask
(suspendList.SearchList(task_name)==true,
ActivateTask(task)?,
Task_suspendTOready(callTask))
s0
SCH6
(runTask.priority<readyQueue.RetuFirPrio(),
_, Task_runTOready())
SCH3TerminateTask
(_, TerminateTask()?,
Task_runTOsuspend())
SCH9ChainTask
SCH7ChainTask
(suspendList.SearchList(callTask)==false,
ChainTask(task)?, _)
SCH8ChainTask
(suspendList.SearchList(callTask)==true,
ChainTask(task)?,
Task_suspendTOready(callTask))
s1
(_, _, Task_runTOsuspend())

Figure 4.4: Fixed priority scheduler model

- A task in *runTask* sends service command "*ChainTask*(*callTask*)" to scheduler for terminating itself and calling another task;

4.2 Model for FPS

Based on the above analysis, we define an *extended FSM* to establish a model for FPS. The definition of *extended FSM* can be expressed as follows:

Definition 4 (*Extended FSM for FPS*) An *extended FSM* for FPS is: $M_{FPS} = (S, s_0, E, T, V)$, where the definitions of S, s_0, E and transition relation set T are same as *definition 2*. In our FPS model, the V consists of *readyQueue*, *suspendList* and *runTask* store structure and we use four functions: *Task_readyTOrun*(), *Task_runTOready*(), *Task_suspendTOready*(*TaskConfigure callTask*) and *Task_runTOsuspend*() to represent the scheduler's actions which can eventually realize the state transition of task. The behaviors of each function is shown below:

```
Task_readyTOrun() {
    runTask=readyQueue.Dequeue();
}
Task_runTOready(){
    readyQueue.Enqueue(runTask);
    runTask=NULL;
}
Task_suspendTOready(callTask){
    string TaskName=callTask.TaskName;
    TaskConfigure enTask=suspendList.DeleteList(TaskName);
    readyQueue.Enqueue(enTask);
  }
  Task_runTOsuspend(){
    suspendList.InsertList(runTask);
    runTask=NULL;
  }
```

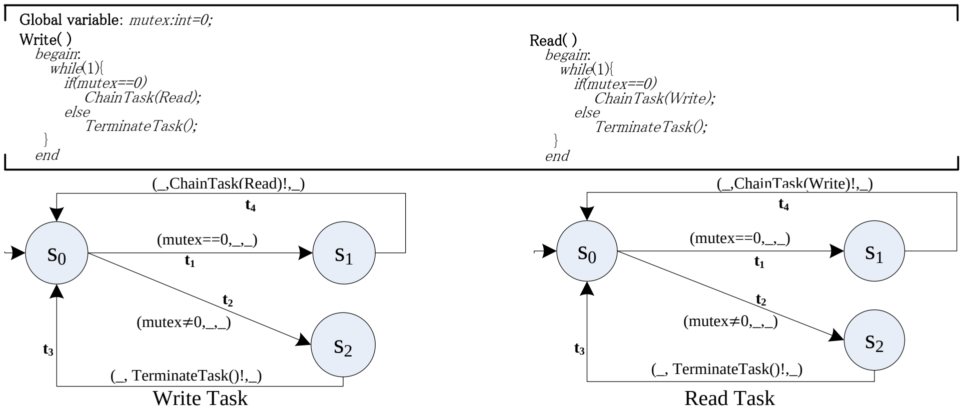
In addition, there are three types of events in our FPS model, including *TerminateTask!*, *ChainTask!* and *ActivateTask!*. These events belong to interactive service commands between tasks and FPS. The FPS model is shown in figure 4.4.

4.3 Definition for describing task behaviors

In our research, each task's behaviors are described by *E-FSM* which has been defined in *definition 2*. Since our research basing on OSEK/VDX OS, the synchronization events set E of a task consist of three types of service commands, such as *TerminateTask*?, *ChainTask*? and *ActivateTask!*?. Especially, since tasks use service commands to request FPS for achieving terminating itself, activating a task or chaining a task, we combine the scheduler and task's *E-FSM* basing on service commands. In order to ensure tasks and FPS can be synchronously executed, we use *definition 5* to define the process of a service command that appears in the task's transition $t \in T$.

Definition 5 (*how to process a service command appearing in transition t*) Let t $=(s_i, c, e, a, s_j)$, where s_j denotes the target state of s_i. If e is not *null* and $e \in \{$ *TerminateTask*?, *ActivateTask*?, *ChainTask*?$\}$, the transition relation t must follow this structure:

$$t = (s_i, \_, e, \_, s_j) \tag{4.1}$$



Global variable: *mutex:int=0;*

```
Write( )
  begain:
    while(1){
      if(mutex==0)
         ChainTask(Read);
      else
         TerminateTask();
    }
  end
```

```
Read( )
  begain:
    while(1){
      if(mutex==0)
         ChainTask(Write);
      else
         TerminateTask();
    }
  end
```

Figure 4.5: How to process a service command that appears in transition relation t

Example 6 In this example, we will show the process about how to describe a service command that appears in transition t. We can see in the figure 4.5, the service commands

ChainTask and *TerminateTask* are represented as *t*(*s*, _, *ChainTask*(*Read*/*Write*)!, _, *s'*) and *t*(*s*, _, *TerminateTask*()!, _, *s'*) respectively.

Chapter 5

Bounded model checking in the presence of Scheduler

In previous chapter, we have established a model for fixed priority scheduler with *E-FSM*. In this chapter, we will firstly talk about how to use a execution tree to model all of the tasks' execution paths under FPS's dispatching. In the second part, we will illustrate two strategies of extracting execution paths based on execution tree in which BMC is employed to generate the VCs for the transition system *M*.

5.1 Execution tree

5.1.1 Concept of execution tree

In our approach, tasks can send three types of service commands to FPS in any branch and each service command can be used by FPS to change the data of FPS's *readyQueue*, *supendList* or *runTask*. If we use *definition 3* to simply combine tasks and FPS behaviors, we will not obtain the correct combinatorial model for the verification system. In order to obtain all possible execution paths of a verification system, we establish an execution tree to represent tasks' executing process which are conducted by FPS. In the execution tree, each possible execution path is from root node to one of leaf-nodes and the total amount of execution paths is equal to the total amount of leaf-nodes. Especially, each node of execution tree consists of node index *NodeIndex*, a transition relation $t \in T^i$ where i means index of task, a FPS model *copyFPSModel* and a pointer array *childPoint*$[\cdot]$ which is used to point to its children nodes. The structure of node is as follows:

$$Node : \{NodeIndex, t, copyFPSModel, childPoint[\cdot]\}$$

Since each node of execution tree includes a FPS model in our approach, the node of execution tree can call its FPS model to response to the service commands which exists in its transition relation t. Here we use an example to show the process about how to establish a execution tree.

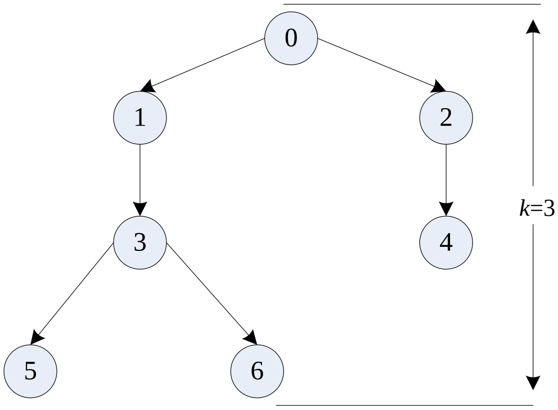


Figure 5.1: An execution tree

Example 7 In this example, we use *Write* task and *Read* task that appear in *example 6* as our verification system. In addition, we set the configure files of *Write* task and *Read* task as follows:

Configure files of $Write$ task and *Read* task
$Wrtie : \{Write, 4, true\};$
$Read : \{Read, 2, false\};$

In order to establish execution tree, firstly, we create an *initial FPS*. In the *initial FPS*, each task's configure file is inserted into *readyQueue* or *suspendList* according to the parameter *autostart* of task's *configure file* (if a task whose *autostart* is true, the task is inserted into *readyQueue*, otherwise, task is inserted into *suspendList*). Especially, in the *initial FPS*, *runTask* is equal to NULL. Hence, we can get the following *initial FPS* according to the configure files of *Write* task and *Read* task.

Initial FPS:
readyQueue:$< Write >$
suspendList:$\{Read\}$
$runTask$: NULL

Secondly, we create a *root-node* for execution tree. We set the *NodeIndex* of *root-node* as 0, transition t as *null*, and *copyFPSModel* is set to be equal to *initial FPS* and each of *childPoint*$[\cdot]$ to point to NULL. Since *runTask* is NULL, FPS of *root-node* dispatches *Write* task from *readyQueue* to *runTask* for running. If the *Write* task occupies the *runTask*, the transition relations t_1 and t_2 will be unfolded and saved in *child-node 1* and *child-node 2* of execution tree respectively, as we can see in the figure 5.1. Furthermore, the *childPoint*$[\cdot]$ of *root-node* point, *child-node 1* and *child-node 2*, and the *copyFPSModel* of *child-node 1* and *child-node 2* are the same as his father's *copyFPSModel*.

In *child-node 1* and *child-node 2*, since the service command of transition t_1 and t_2 are *null*, FPS of *child-node 1* and *child-node 2* do not need to be synchronously executed. In the next step, the transition t_4 and t_3 are unfolded from *child-node 1* and *child-node 2*, and saved in *child-node 3* and *child-node 4* respectively. The *copyFPSModel* of the new *child-node 3* and *child-node 4* are equal to their father respectively.

Particularly, in *child-node 3*, since the service command of transition t_4 is *ChainTask*(*Read*)!, FPS of *child-node 3* will be synchronously executed, the execution behaviors of FPS are as follow:

1. FPS of *child-node 3* dispatches *Read* task from *suspendList* to *readyQueue*;

2. FPS of *child-node 3* dispatches *Write* task from *runTask* to *suspendList*.

We can see, in *child-node 3*, since the *runTask* of *copyFPSModel* is NULL when the FPS of *child-node 3* finishes its operations, FPS of *child-node 3* will dispatch *Read* task from *readyQueue* to *runTask* for running. The *Read* task's transitions t_1 and t_2 will be unfolded and saved in *child-node 5* and *child-node 6* respectively. However, in *child-node 4*, since the transition t_3 whose service command is *TerminateTask!*, FPS of *child-node 4* will dispatch *Write* task from *runTask* to *suspendList*. In this time, we can find the *readyQueue* of *child-node 4* is NULL. Hence, the FPS of *child-node 4* cannot dispatch a task form *readyQueue* to *runTask* for running, we do not need to create new *child-nodes* for *child-node 4*. In this example, we have obtained an execution tree which represents the *Write* and *Read* task executed three steps under the FPS's dispatching, the execution tree is illustrated in figure 5.1.

5.1.2 The algorithm for establishing a *k*-step execution tree

Transitions store structure for tasks

As we know, a well-designed storage structure can reduce the searching complexity. In our research, we use Hash method [32] to store transition relations set T of a task to reduce the time of establishing execution tree, the Hash store structure is illustrated in figure 5.2, where the state index means index of source state of $t_i \in T$. Since t_1 and t_2 have the same source state (where, t_1 and t_2 represent the two branches that start from state s_0), t_1 and t_2 are stored in the same array (the structure of element of store array is: $(s, condition, event, action, s)$). In order to store and access the transition relations set T for a task, we define two types of operations on it. The operations are as follow:

Operation 1 (*Create a Hash store structure and save transition relations T of a task*):

$$Create(T) \longmapsto [\text{ Hash store }]^{[task\_index]}$$

Operation 2 (*Select and return t iff the source state s of t is equal to input state s*):

$$Select(s) \text{ from } [\text{ Hash store }]^{[task\_index]}$$

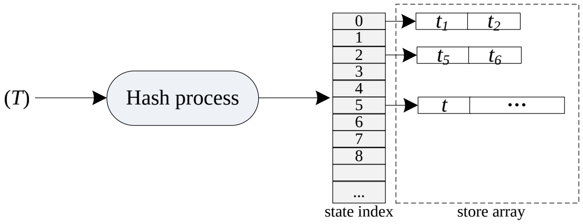


Figure 5.2: The structure about how to store the transition relations T of a task

Establishment of k-step execution tree

In order to show the process about how to establish a k-step execution tree, we use *definition 6* and *definition 7* to describe the processes of creating *root-node* and *child-node* respectively, the *definition 6* and *definition 7* are as follow:

Definition 6 (*Creating process of root-node*): The execution tree has only one root-node. The initial attributes and operation function of root-node are as follows:

- **Initial attributes of root-node:**
  NodeIndex: The root-node's *NodeIndex* is 0;
  t: The root-node's transition relation t is *null*;
  copyFPSModel: In the root-node's *copyFPSModel*, *readyQueue*, *suspendList* and *runTask* are equal to the *readyQueue*, *suspendList* and *runTask* of initial FPS model respectively.
  childPoint[·]: Each pointer of the *childPoint*[·] points to NULL in the initial state;

- **Operation function of root-node:**

  1. Root-node calls its FPS model *copyFPSModel* to dispatch a task from *readyQueue* to *runTask*. Iff the *runTask* is not equal to NULL, goto 2, otherwise, return;
  2. Extract the index i of task which is in the *runTask*;
  3. Use the Operation 2 of Hash store of $task^i$ to select and return all of the transition relations with s_c^i and save each transition relation t into a set ET, where s_c^i is current state of extended FSM of $task^i$. The initial value of s_c^i is equal to the initial state of extend FSM of $task^i$ in initial state;
  4. Create n new child-nodes, where n is equal to the total amount of elements of ET;
  5. Let n pointers of the *childPoint*[·] of root-node point to these new child-nodes respectively;

Definition 7 (*Creating of child-node*): When a task which is in the *runTask* executes one step, one transition relation t of the task is saved into a child-node. In child-node, the source state s of transition relation t is equal to target state s' of his father's transition relation t. The initial attributes and operation function of child-node are as follow:

- **Initial attributes of child-node:**

  NodeIndex: In execution tree, the child-node's *NodeIndex* is indicated according to the order which is from top to bottom and from left to right, the order is shown in figure 5.3;

  t: Transition relation *t* of child-node is equal to one of element of *ET* which is founded by his father;

  copyFPSModel: In the child-node's *copyFPSMode*, *readyQueue*, *suspendList* and *runTask* are equal to *readyQueue*, *suspendList* and *runTask* of its father's *copyFPSModel* respectively in the initial state.

  childPoint[·]: Each pointer of the *childPoint*[·] of the child-node points to NULL in the initial state;

- **Operation function of child-node:**

  1. If $t.e$ is not equal to the symbol "_" , goto 2, otherwise goto 4;

  2. Child-node calls its FPS model *copyFPSModel* to response to the service command e:

```
case service command="TerminateTask!":
    ↘ child-node.copyFPSModel.SCH$_3$; goto 3;
case service command="ChainTask!":
    ↘ child-node.copyFPSModel.(SCH$_8$ | SCH$_7$); ↘ child-node.copyFPSModel.SCH$_9$; goto 3;
case service command="ActivateTask!":
    ↘ child-node. copyFPSMode.(SCH$_4$ | SCH$_5$);
    if (child-node.copyFPSModel.runTask.priority<callTask.priority)
        extracting the index $i$ of task which in the runTask; $s_c^i$=t.s';
        ↘ child-node.copyFPSModel.SCH$_6$; ↘child-node.copyFPSModel.(SCH$_2$ | SCH$_1$); goto 3;
```

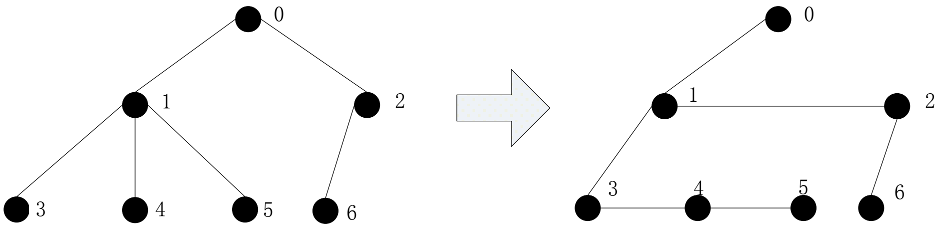
  3. If the *runTask* in the child-node's *copyFPSModel* is equal to NULL, ↘ child-node.copyFPSModel.(SCH$_2$ | SCH$_1$); Otherwise, nothing will be done;

  4. If the *runTask* in the child-node's *copyFPSModel* is not equal to NULL, extract the index i of task which is in the *runTask* and goto 5. Otherwise, it means there is no task can be dispatched from *readyQueue* to *runTask* by FPS, so we do not need to create new child-nodes for this node;

  5. Use the Operation 2 of Hash store of $task^i$ to select and return all of the transition relations with s_c^i and save each transition relation t into set ET;

  6. Create n new child-nodes, where n is equal to the total amount of elements of ET;

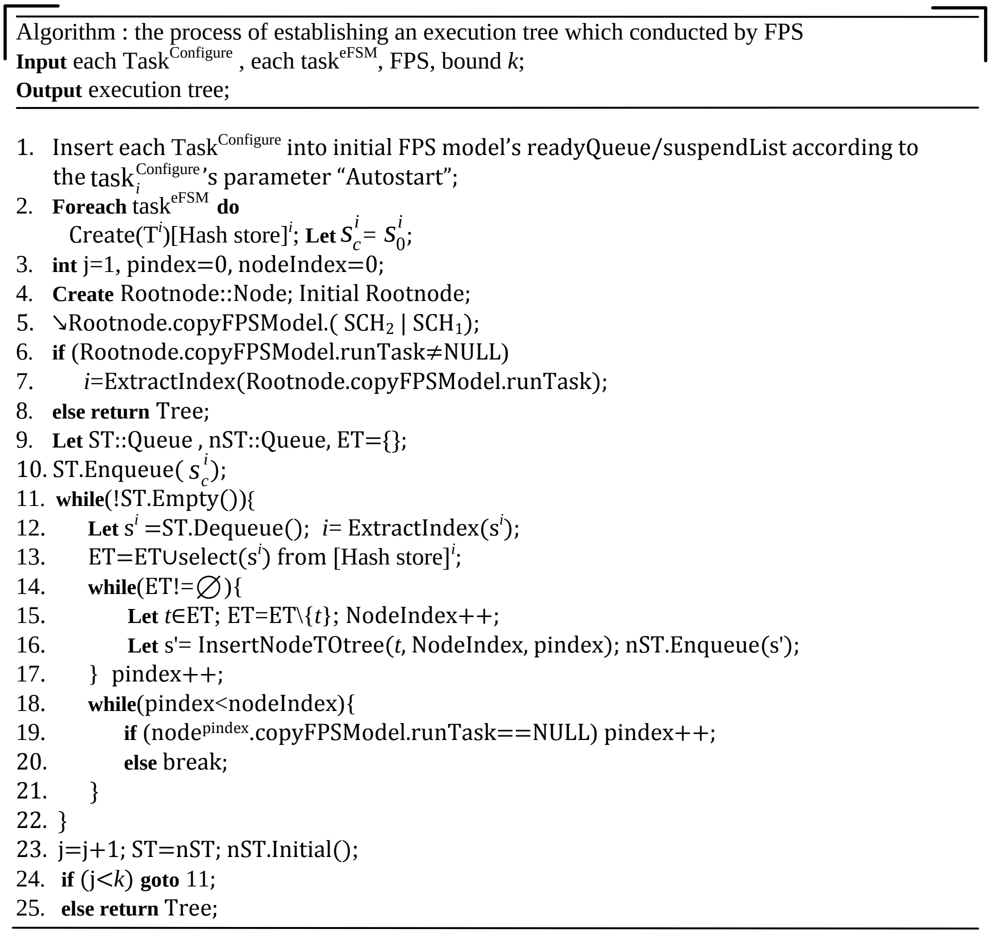
  7. Let n pointers of the *childPoint*[·] point to these new child-nodes respectively;



Common structure Tree

Child-brother structure Tree

Figure 5.3: The transition between the common structure tree and child-brother tree



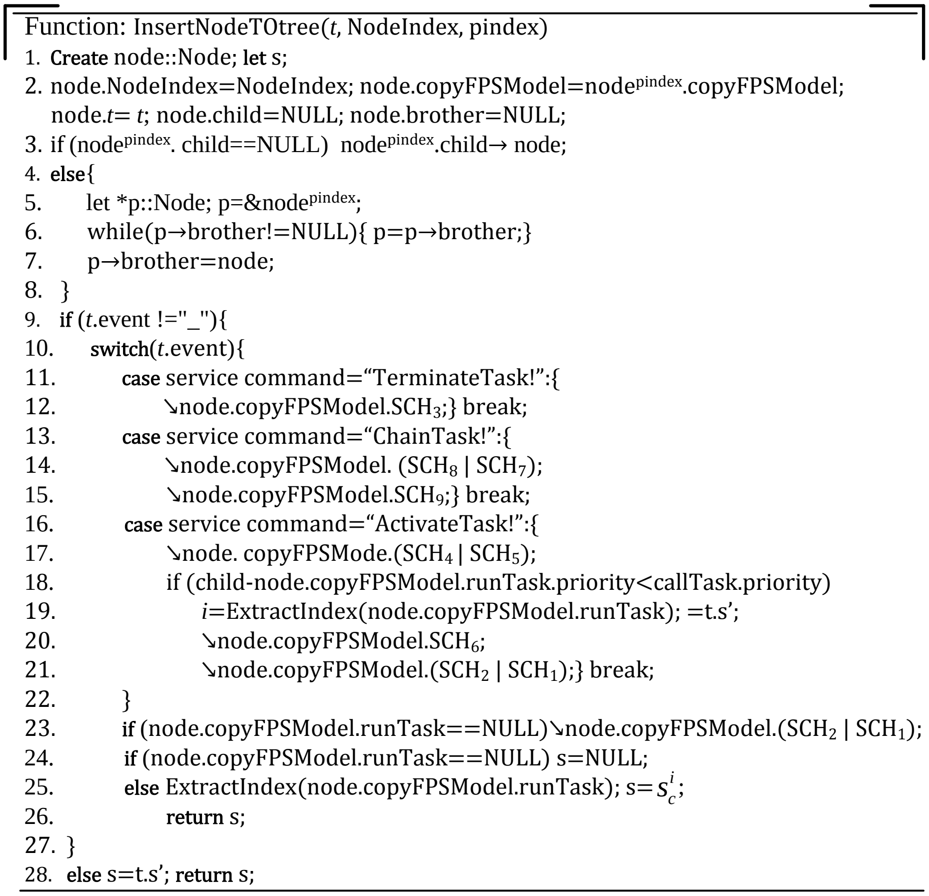
```
Algorithm : the process of establishing an execution tree which conducted by FPS
Input each Task^Configure , each task^eFSM, FPS, bound k;
Output execution tree;

1.  Insert each Task^Configure into initial FPS model's readyQueue/suspendList according to
    the $task_i^{Configure}$'s parameter "Autostart";
2.  Foreach task^eFSM do
      Create(T^i)[Hash store]^i; Let $S_c^i = S_0^i$;
3.  int j=1, pindex=0, nodeIndex=0;
4.  Create Rootnode::Node; Initial Rootnode;
5.  ↘Rootnode.copyFPSModel.( $SCH_2$ | $SCH_1$);
6.  if (Rootnode.copyFPSModel.runTask≠NULL)
7.     i=ExtractIndex(Rootnode.copyFPSModel.runTask);
8.  else return Tree;
9.  Let ST::Queue , nST::Queue, ET={};
10. ST.Enqueue( $s_c^i$);
11. while(!ST.Empty()){
12.     Let $s^i$ =ST.Dequeue();  i= ExtractIndex($s^i$);
13.     ET=ET∪select($s^i$) from [Hash store]^i;
14.     while(ET!=∅){
15.         Let t∈ET; ET=ET\{t}; NodeIndex++;
16.         Let s'= InsertNodeTOtree(t, NodeIndex, pindex); nST.Enqueue(s');
17.     }  pindex++;
18.     while(pindex<nodeIndex){
19.         if (node^pindex.copyFPSModel.runTask==NULL) pindex++;
20.         else break;
21.     }
22. }
23. j=j+1; ST=nST; nST.Initial();
24. if (j<k) goto 11;
25. else return Tree;
```

Figure 5.4: The algorithm for establishing an execution tree which is conducted by FPS

Note that the symbol "↘" in above operation means executing a transition on FPS model, the initial value s_c^i is equal to initial state s_i^0 of extended FSM of $task^i$.

In the above operations of *root-node* and *child-node*, the total amount of elements of set ET means the amount of branches that start from a state s which exists in extended FSM of a task. Usually, since the branch amount is uncertain, we use *child-brother* structure



```
Function: InsertNodeTOtree($t$, NodeIndex, pindex)
Create node::Node; let s;
node.NodeIndex=NodeIndex; node.copyFPSModel=node$^{pindex}$.copyFPSModel;
node.$t$= $t$; node.child=NULL; node.brother=NULL;
if (node$^{pindex}$. child==NULL)  node$^{pindex}$.child→ node;
else{
    let *p::Node; p=&node$^{pindex}$;
    while(p→brother!=NULL){ p=p→brother;}
    p→brother=node;
}
if ($t$.event !="_"){
    switch($t$.event){
        case service command="TerminateTask!":{
            ↘node.copyFPSModel.SCH$_3$;} break;
        case service command="ChainTask!":{
            ↘node.copyFPSModel. (SCH$_8$ | SCH$_7$);
            ↘node.copyFPSModel.SCH$_9$;} break;
        case service command="ActivateTask!":{
            ↘node. copyFPSMode.(SCH$_4$ | SCH$_5$);
            if (child-node.copyFPSModel.runTask.priority<callTask.priority)
                $i$=ExtractIndex(node.copyFPSModel.runTask); =t.s';
                ↘node.copyFPSModel.SCH$_6$;
                ↘node.copyFPSModel.(SCH$_2$ | SCH$_1$);} break;
        }
        if (node.copyFPSModel.runTask==NULL)↘node.copyFPSModel.(SCH$_2$ | SCH$_1$);
        if (node.copyFPSModel.runTask==NULL) s=NULL;
        else ExtractIndex(node.copyFPSModel.runTask); s=$s_c^i$;
            return s;
}
else s=t.s'; return s;
```

Figure 5.5: The algorithm for inserting a new node to execution tree

to construct the nodes relationships of the execution tree. The transition between the common structure tree and *child-brother* structure tree is shown in figure 5.3.

We use an algorithm to show the process of establishing an execution tree in which the *child-brother* structure is used to construct the relationships of nodes. The algorithm is illustrated in figure 5.4, where ST and nST are "Queue^{FIFO}" structure [33], function "InsertNodeTOtree(t, $NodeIndex$, $pindex$)" means insert a new *child-node* into tree. The process of function "InsertNodeTOtree(t, $NodeIndex$, $pindex$)" is shown in figure 5.5. We can get the execution tree according to above approach, the time complexity of establishing a execution tree under k-step is $(k * w)$, where w=MAX (λ^l) , λ^l is equal to the total amount of nodes of l^{th}-level tree. For an execution tree, if the *leaf-node*'s amount is m, then there exist m execution paths and each path is from the *root-node* to a *leaf-node*.

5.2 Two strategies for extracting execution paths

Based on execution tree, we propose two strategies of extracting execution paths. These two strategies not only can achieve the execution paths which reflect the execution transition relations of tasks conducted by FPS but also gain the VCs from execution paths. The difference between our two strategies is that one strategy translates all of the execution paths which exist in execution tree into VCs with BMC method while the other computes the variables' values by itself to trim the execution tree. We name the first approach as "general approach of extracting execution paths (GAE^{2}P)" and the other as "trim-tree approach of extracting execution paths (TAE^{2}P)". In the following part, we will show these two strategies in detail.

5.2.1 General strategy of extracting execution paths (GAE^{2}P)

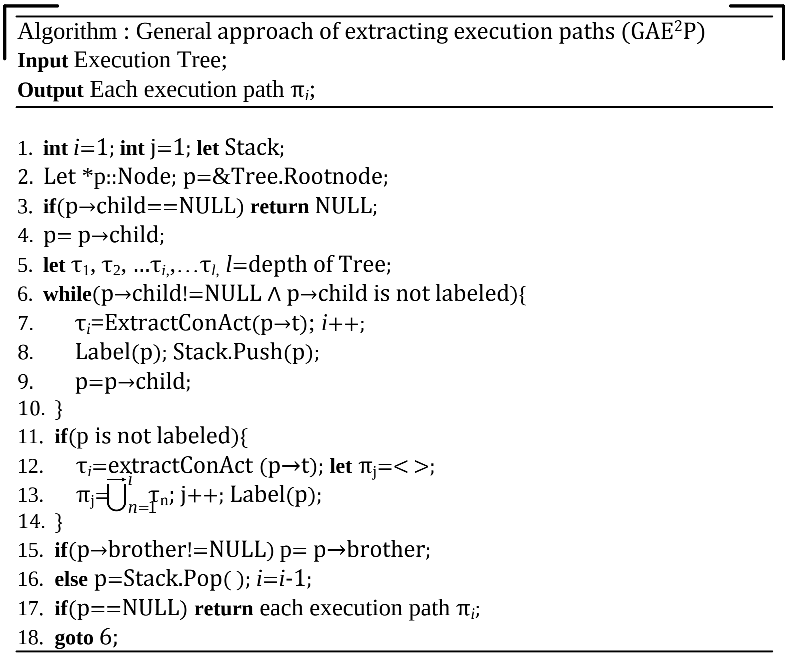
In the GAE^{2}P, all of the possible execution paths will be obtained in which BMC method is employed to generate the VCs of k-step transition system. In the execution tree, each execution path is a direct track which is from *root-node* to a *leaf-node* and the total amount of execution paths is equal to the *leaf-nodes* of execution tree. For instance, in the figure 5.1, we can obtain three execution paths under bound k=3. The three execution paths are as follow:

π_1=τ_0, τ_1, τ_3, τ_5;
π_2=τ_0, τ_1, τ_3, τ_6;
π_3=τ_0, τ_2, τ_4;

Note that in the above execution paths π_1, π_2 and π_3, each τ_i represents a node of execution tree and i is index of node. Since each node τ_i possesses a transition relation t, we can use transition relations which exist in a execution path π_j to represent a execution path. Actually, only the guard function $c \in C$ and action calculation express $a \in A$ which belong to a transition relation of execution path π_j are used to represent a VCs of τ_i according to BMC in our article. Each of execution paths which are represents by VCs is as follows:

π_1= τ_1:$(mutex_0==0)\wedge(mutex_1=mutex_0)\wedge$
τ_3:$(mutex_2=mutex_1)\wedge$
τ_5:$(mutex_2=0)\wedge(mutex_3=mutex_2)$
π_2= τ_1:$(mutex_0==0)\wedge(mutex_1=mutex_0)\wedge$
τ_3:$(mutex_2=mutex_1)\wedge$
τ_6:$(mutex_2 \neq 0)\wedge(mutex_3=mutex_2)$
π_3= τ_2:$(mutex_0 \neq 0)\wedge(mutex_1=mutex_0)\wedge$
τ_4:$(mutex_2=mutex_1)$

Therefore, we can gain the transition system M with execution paths π_1, π_2 and π_3 according to formula (2.8), the transition system M is as follow:



```
Algorithm : General approach of extracting execution paths (GAE$^2$P)
Input Execution Tree;
Output Each execution path $\pi_i$;

1. int $i$=1; int j=1; let Stack;
2. Let *p::Node; p=&Tree.Rootnode;
3. if(p→child==NULL) return NULL;
4. p= p→child;
5. let $\tau_1$, $\tau_2$, ...$\tau_{i,}$...$\tau_{l,}$ $l$=depth of Tree;
6. while(p→child!=NULL ∧ p→child is not labeled){
7.     $\tau_i$=ExtractConAct(p→t); $i$++;
8.     Label(p); Stack.Push(p);
9.     p=p→child;
10. }
11. if(p is not labeled){
12.     $\tau_i$=extractConAct (p→t); let $\pi_j$=< >;
13.     $\pi_j$=$\overrightarrow{\bigcup}_{n=1}^{l}\tau_n$; j++; Label(p);
14. }
15. if(p→brother!=NULL) p= p→brother;
16. else p=Stack.Pop( ); $i$=$i$-1;
17. if(p==NULL) return each execution path $\pi_i$;
18. goto 6;
```

Figure 5.6: The algorithm of general approach of extracting execution paths (GAE^{2}P)

$$[[M]]_3 := I(s_0) \wedge \bigwedge_{i=1}^{3} \pi_i \tag{5.1}$$

If we gain the transition system M, then we can translate the transition system M and verification property formula f into Yices file, and use Yices to check whether this transition system M satisfies the verification property formula f.

Since the child-brother structure is used to construct the nodes' relationship of execution tree in establishing execution tree, we adopt a revised depth-first search strategy to gain all of the execution paths in GAE^{2}P, the algorithm of GAE^{2}P is shown in figure 5.6. In this algorithm, we use a particular operational character "$\overrightarrow{\cup}$" to save data of each node of a path into an ordered set π_j according to the order of depth-first search strategy [34]. Actually, in order to gain the VCs of i-step from a path, in the algorithm GAE^{2}P only the condition and action which are included in the transition t of each node of a path are saved into the element τ_i of π_j. In order to extract the condition and action form a node, we use a function *ExtractConAct*(t) which appears in algorithm GAE^{2}P to represent the extraction behavior. In addition, a stack [35] is used to accomplish the traversal process in algorithm GAE^{2}P.

According to the GAE^{2}P's algorithm which appears in figure 5.6, we can see the time complexity of gaining all of the execution paths is $O(d * c)$, where d means the depth of execution tree (actually, the depth of execution tree depend on the bound k) and c means the maximum degree of all the nodes in execution tree. When all the path π_j are gained from execution tree, we can use these π_j to establish a transition system M under bound

k. The transition system M is as follows:

$$M = I(s_0) \vee_{j=1}^{n} \pi_j \tag{5.2}$$

$$M = I(s_0) \vee_{j=1}^{n} \wedge_{i}^{k} \tau_i^j \tag{5.3}$$

Here, n means the total amount of the execution paths, $I(s_0)$ is a function which is used to indicate the initial value of variables. Especially, if the total amount r of elements of an execution path π_j is smaller than bound k, we use an additional element τ to fill these non-existent elements from τ_{r+1} to τ_k in order to ensure that the total amount of elements of every execution paths is equal to bound k . The additional element τ whose condition and action are null, we use symbol "_" instead of condition and action respectively.

5.2.2 Trim-tree strategy of extracting execution paths (TAE^{2}P)

In the GAE^{2}P, we can see all of the execution paths are included in transition system M. However, if the initial value of each variable of verification problem has been given out explicitly and there are only several variables not related to the verification properties, we do not need to check all of the execution paths. In order to reduce the total amount of execution paths, all of the variables' values are computed in each node of execution tree in TAE^{2}P. Therefore, we can use variables' values to examine the guard functions and trim the unreachable node to get one reachable execution path l from all of the execution paths. The path l reflects the execution track of tasks which are conducted by FPS.

For instance, in the figure 5.1, we assume initial value of *mutex* is equal to 0. Intuitively, we can find the node 2 is a unreachable node when the initial value of global variable *mutex* is equal to 0, hence, we can trim the relation between node 0 and node 2. Furthermore, since the action $a \in A$ which exist in the translation relation of node 1 is equal to symbol "_", the value of variable *mutex* is equal to 0, the same value of variable *mutex* also can be found in the node 3. Fortunately, since the value of variable *mutex* is equal to 0 in the node 3, we can trim the relation between node 3 and node 6 once again. Therefore, according to the above trimming process, we can only obtain one execution path l, furthermore, each variable's value has been acquired and saved in each node of path l when we trim the execution tree. The reachable path l is as follows:

l= τ_0, τ_1, τ_3, τ_5;

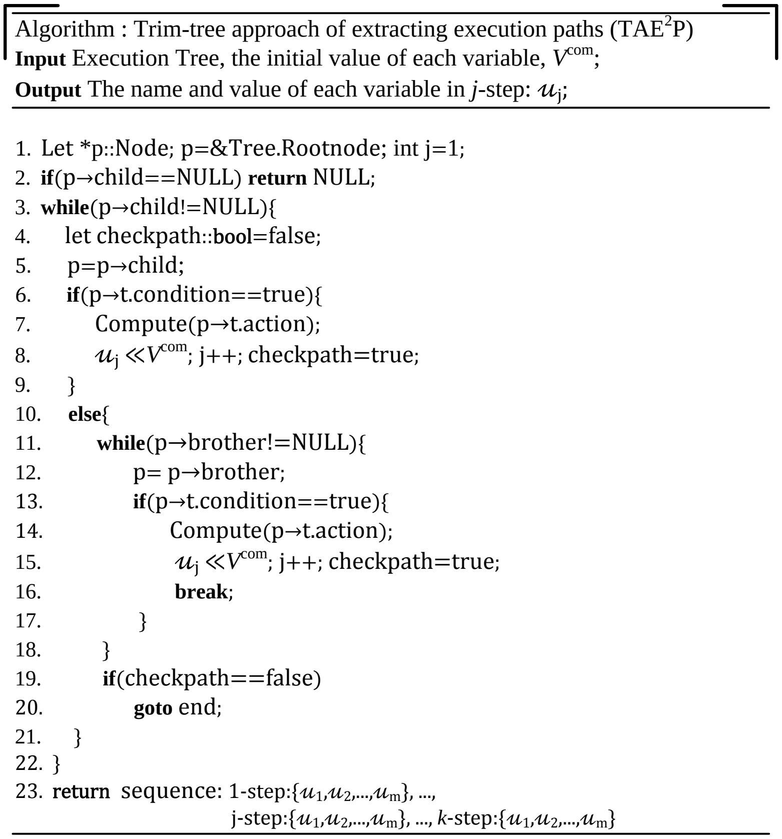
Since each variable's value has been computed and saved in each node τ_i, we can use variables which exist in each τ_i of path l to represent the tasks' execution processes, the path l which is represented by each node τ_i' variables is as follows:

l= $(mutex_0 = 0) \wedge (mutex_1 = 0) \wedge (mutex_2 = 0) \wedge (mutex_3 = 0)$

Therefore, we can gain the transition system M with τ_i according to formula (2.8), the transition system M is as follow:

$$M = I(s_0) \vee_{i=1}^{k} \tau_i \tag{5.4}$$

If we gain the transition system M, then we can translate the transition system M and verification property formula f into Yices file, and use Yices to check whether this transition system M satisfies the verification property formula f.



Algorithm : Trim-tree approach of extracting execution paths (TAE^{2}P)
Input Execution Tree, the initial value of each variable, V^{com};
Output The name and value of each variable in *j*-step: u_j;

```
Let *p::Node; p=&Tree.Rootnode; int j=1;
if(p→child==NULL) return NULL;
while(p→child!=NULL){
    let checkpath::bool=false;
    p=p→child;
    if(p→t.condition==true){
        Compute(p→t.action);
        $u_j \ll V^{com}$; j++; checkpath=true;
    }
    else{
        while(p→brother!=NULL){
            p= p→brother;
            if(p→t.condition==true){
                Compute(p→t.action);
                $u_j \ll V^{com}$; j++; checkpath=true;
                break;
            }
        }
        if(checkpath==false)
            goto end;
    }
}
return  sequence: 1-step:{$u_1$,$u_2$,...,$u_m$}, ...,
                  j-step:{$u_1$,$u_2$,...,$u_m$}, ..., k-step:{$u_1$,$u_2$,...,$u_m$}
```

Figure 5.7: The algorithm of Trim-tree approach of extracting execution paths (TAE^{2}P)

The algorithm of TAE^{2}P is shown in figure 5.7. Especially, in the TAE^{2}P's algorithm, we insert a function *Compute*(*action*) into TAE^{2}P to compute the variables' values in order to trim the unreachable paths based on the revised depth-first search strategy, hence, we can only get one execution path l from all of the execution paths. The path l reflects the execution track of tasks which are conducted by FPS. In addition, each variable's name and value of every node which exists in the path l are saved into an ordered set V according to execution order of path l. Let $\forall \mu \in V_j^{com}$, $V_j^{com} = \bigcup V^i$, where i means the index of task and j represents the j-step. The structure of element of μ is as follows:

$$\mu\text{:\{variable name, variable value\}}$$

Therefore, we can use the sequence V_1^{com}, V_2^{com}, ..., V_k^{com} to establish the transition system M under bound k. The transition system M is as follows:

$$M = I(s_0) \vee_{j=1}^{k} V_j^{com} \tag{5.5}$$

$$M = I(s_0) \vee_{j=1}^{k} \wedge_{l=1}^{m} \mu_l^j \tag{5.6}$$

Here, m means the total amount of variables of the V^{com}, $I(s_0)$ is a function which is used to indicate the initial values of variables.

Especially, in the function *Compute*(*action*) which appears in algorithm of TAE^{2}P, if the parameter action equals symbol "_", it means that we do not need to compute the variables' values, in this case, $\forall v \in V^{com}$, $v' = v$. If the action is not equal to "_", it means there exists a calculation expression, in this case, $\exists v_f \in V^{com}$, the variable v_f not only appear in the calculation expression but also appear on the left side of symbol "=", we can use following formulae to represent the computing process of the function *Compute*(*action*).

$$\exists v_f \in V^{com}, v_f^{'} = a(v_f) \tag{5.7}$$

$$\forall v \in V^{com} \setminus \{v_f\}, v^{'} = v \tag{5.8}$$

The symbol "$\ll$" which appears in algorithm TAE^{2}P means outputting the name and value of each variable $v \in V^{com}$ to μ_l.

5.3 Verification process with SMT tool Yices

In BMC, a verification system or problem consists of a transition system M, a temporal logic verification formula f and a user-supplied bound k. According to GAE^{2}P and TAE^{2}P, the transition system M under bound k can be obtained, the processes of verification formula f have been proposed in [4], therefore, we will not discuss it in our article. In this part we will only talk about how to translate the transition system M which is constituted by VCs into Yices file.

In the GAE^{2}P, since a path $\pi_j = \wedge_i^k \tau_i^j$ and each τ_i^jconsists of a guard function expression $c \in C$ and an action calculation expression $a \in A$, we can use formula (5.2) to represent the VCs of a verification system or problem. In order to achieve the Yices file according to the formula (5.2), we use following translation process to interpret c and a which appears in τ_i^j.

Translation process 1 (*process guard function c*): if c is equal to symbol "_", we do not need to translate it into Yices file, otherwise the translation process is: $c \longrightarrow c_Y$, where c_Y is a Yices expression which is defined by BNF [26] and symbol "$\longrightarrow$" means translating the c into c_Y, the c_Y is as follows:

$c_Y ::=< cur_s >< \theta >< var_m^i >< var_n^i >< cur_e >$;
$cur_s ::=$ "(";
$cur_e =$ ")"
$\theta ::=$ " $> \mid >= \mid < \mid <= \mid = \mid / =$";
var_m^i::= the variable $var_m^i \in V^{com}$ which appears on the left side of operator θ;
var_n^i::= the variable $var_n^i \in V^{com}$ which appears on the right side of operator θ;

Note that θ is a comparison operator which appears in the c, i is index of τ_i^j.

Translation process 2 (*process action a*): if a is not equal to symbol "_", the translation process is: $a \longrightarrow a_Y$, where a_Y is a Yices expression defined by BNF as follows:

$a_Y ::=< cur_s >< eP >< v^i >< exp >< cur_e >;$
$exp ::=< cur_s >< \theta >< expr >< expr >< cur_e >;$
$expr ::= cv^{i-1} \mid exp;$
$cur_s ::=$ "("; $cur_e =$ ")"
$eP ::=$ " =";
$\theta ::=$ "$+ \mid - \mid * \mid / \mid =$";
v^i ::=the variable $v^i \in V^{com}$ which appears on the left side of symbol "=" of a;
cv^{i-1} ::=the variable $cv^{i-1} \in V^{com}$ which appears on the right side of symbol "=" of a;

In addition, $\forall v_{other} \in V^{com} \setminus \{v^i\}$, we use another formula to translate each v_{other} into Yices expression in the same i-step to ensure that other variables v_{other} hold its value as same as last step. The translation process is: $v_{other} \longrightarrow z_Y$, where z_Y is a Yices expression defined by BNF as follows:

$z_Y ::=< cur_s >< eP >< v^i_{other} >< v^{i-1}_{other} >< cur_e >;$
$cur_s ::=$ "("; $cur_e =$ ")";
$eP ::=$ " =";

Note that i is index of τ_i^j.

Eespecially, if a is equal to symbol "_", we can also use formula $v \longrightarrow z_Y$ to translate each variable $\forall v \in V^{com}$ into Yices file in i-step.

In the TAE^{2}P, since $\forall v^l \in V^{com}$ whose name and value are saved into μ_l^j , we only need to translate the variable's name and value into Yices expression according to formula (5.4). The translation process is: $\sigma_l^j \longrightarrow L_Y$, where L_Y is a Yices expression defined by BNF as follows:

$L_Y ::=< cur_s >< eP >< v^l_{name} >< v^l_{value} >< cur_e >;$
$cur_s ::=$ "("; $cur_e =$ ")";
$eP ::=$ " =";
v^l_{name} ::=the name of μ_l^j;
v^l_{value} ::=the value of μ_l^j;

Note that l is index of μ_l^j.

Chapter 6

Verification tool

In the chapter 5, we have already discussed the algorithm about how to establish a execution tree, the two types of strategies for extracting execution paths from execution tree and how to translate the VCs of transition system M into Yices file. In this chapter, we will talk about the architecture of our verification tool and how to use our verification tool.

6.1 Architecture of verification tool

In order to accomplish the evaluation of our approach, we have implemented two types of tools according to GAE^{2}P and TAE^{2}P based on the execution tree with C++ respectively. Our verification tool consists of five modules: input files examination module, establishment execution tree module, extraction execution paths module, translation process module and Yices verification module. The architecture of our verification tool is illustrated in figure 6.1.

Our verification tool reads two types of input files for checking a system. One is called "Tasks file" which consists of tasks' behaviors that are described by extended FSM, combination variables set and configure file of each task. The other is called "Verification property formula file" which includes a verification formula that is described by LTL. Especially, in the input files examination module of our verification tool, these two types of input files can be examined. If the input files include some unexpected inputs, our tool can return the reason of unexpected inputs to user. The function of establishment execution tree module is to establish a k-step execution tree according to the input files and bound k. When the k-step execution tree has been established, the extraction execution paths module is used to extract the execution paths from execution tree. Especially, in the extraction execution paths module, user can choose GAE^{2}P or TAE^{2}P strategy to extract the execution paths from execution tree according to the degree of correlation between target system's variables and verification property formula. In order to check whether verification property formula satisfies the execution paths using Yices, the translation process module is used to translate execution paths and verification property formula into Yices file. The verification results are outputted by Yices which is processed in

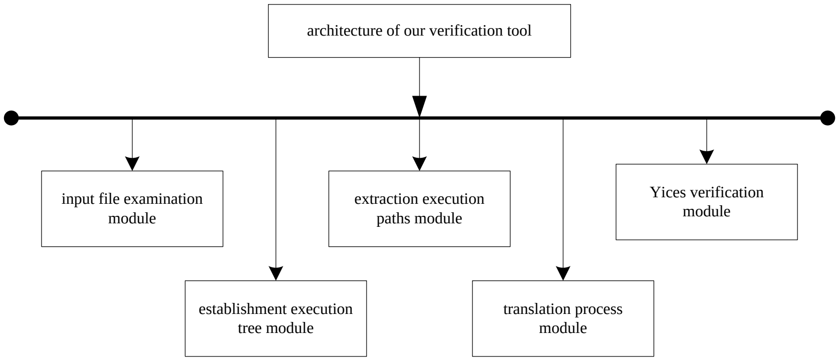

architecture of our verification tool
input file examination module
establishment execution tree module
extraction execution paths module
translation process module
Yices verification module

Figure 6.1: The architecture of our verification tool

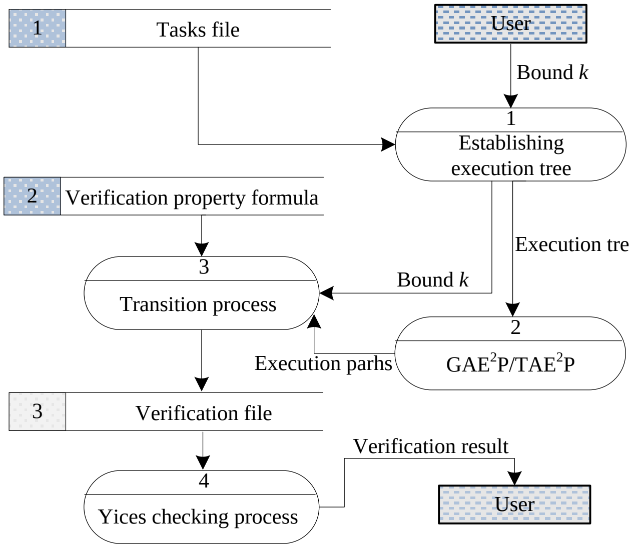
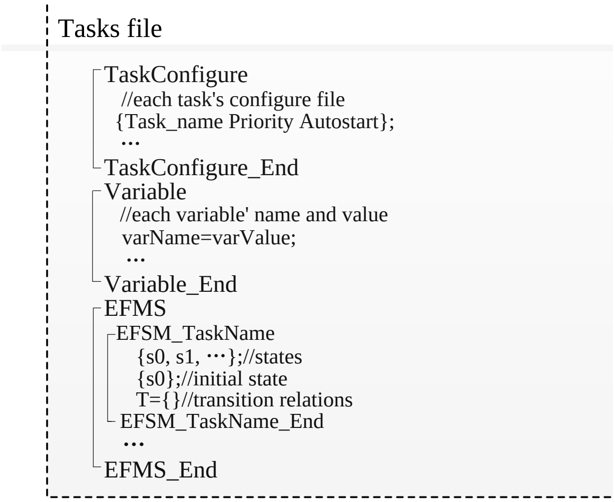


Figure 6.2: The data flow diagram of our tools

Yices verification module. The format of verification results is either “sat” or “unsat”, the symbol “sat” means the verification property formula satisfies the execution paths, the symbol “unsat” means the verification property formula cannot satisfy the execution paths. The data flow diagram of our verification tool is shown in figure 6.2.

6.2 An example for using verification tool

In order to use our verification tool conveniently and easily, the input data types of our verification tool are limited to be three types: *Tasks file*, *Verification property formula file* and bound k. The *Tasks file* is used to store tasks' configure files, the combination variables of each task's global variables and local variables, and each task's behaviors which are described by *E-FSM*. The *Verification property formula file* is used to store the verification property formula which is described by *LTL*. Especially, for the*Tasks file*, we use a format to organize the data of *Tasks file*. The format of *Tasks file* is illustrated in

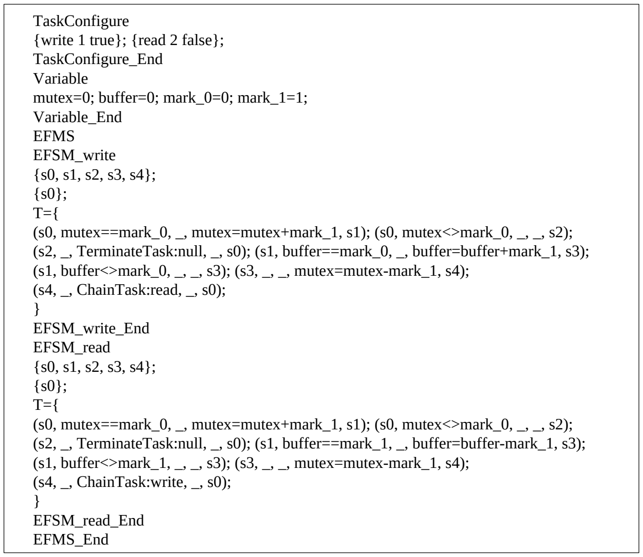


```
Tasks file
  TaskConfigure
    //each task's configure file
    {Task_name Priority Autostart};
    ···
  TaskConfigure_End
  Variable
    //each variable' name and value
    varName=varValue;
     ···
  Variable_End
  EFMS
   EFSM_TaskName
      {s0, s1, ···};//states
      {s0};//initial state
      T={}//transition relations
   EFSM_TaskName_End
    ···
  EFMS_End
```

Figure 6.3: The format of Tasks file

figure 6.3.

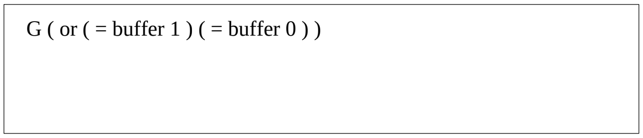
We can see in the figure 6.3, the first part is tasks' configure file, and each task's configure file is indicated in this part. In the second part, all of variables and its value are listed. Besides, each task's behaviors which are described by *E-FSM* are stored in third part. Especially, In the third part of *Tasks file*, the first line is used to save all states of task, the initial state is denoted in the second line, the transition relation T is presented in last line. For instance, we can achieve the *Tasks file* according to figure 7.1, the *Tasks file* is shown in figure 6.4.



```
TaskConfigure
{write 1 true}; {read 2 false};
TaskConfigure_End
Variable
mutex=0; buffer=0; mark_0=0; mark_1=1;
Variable_End
EFMS
EFSM_write
{s0, s1, s2, s3, s4};
{s0};
T={
(s0, mutex==mark_0, _, mutex=mutex+mark_1, s1); (s0, mutex<>mark_0, _, _, s2);
(s2, _, TerminateTask:null, _, s0); (s1, buffer==mark_0, _, buffer=buffer+mark_1, s3);
(s1, buffer<>mark_0, _, _, s3); (s3, _, _, mutex=mutex-mark_1, s4);
(s4, _, ChainTask:read, _, s0);
}
EFSM_write_End
EFSM_read
{s0, s1, s2, s3, s4};
{s0};
T={
(s0, mutex==mark_0, _, mutex=mutex+mark_1, s1); (s0, mutex<>mark_0, _, _, s2);
(s2, _, TerminateTask:null, _, s0); (s1, buffer==mark_1, _, buffer=buffer-mark_1, s3);
(s1, buffer<>mark_1, _, _, s3); (s3, _, _, mutex=mutex-mark_1, s4);
(s4, _, ChainTask:write, _, s0);
}
EFSM_read_End
EFMS_End
```

Figure 6.4: Example for Tasks file

Furthermore, we also can achieve the *Verification property formula file* according to the figure 7.1, the *Verification property formula file* is shown in figure 6.5.

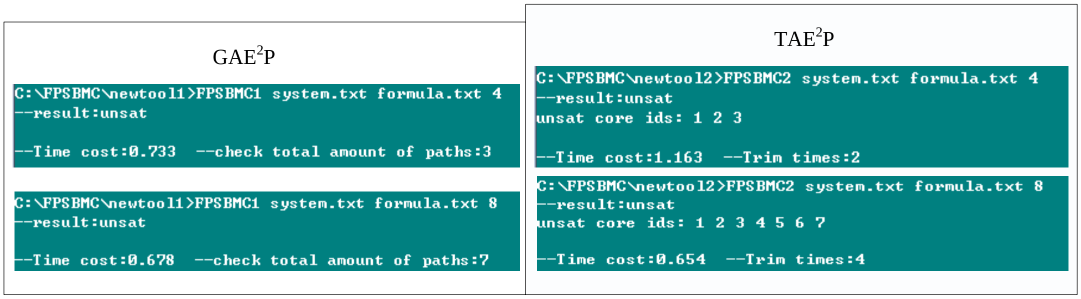


```
G ( or ( = buffer 1 ) ( = buffer 0 ) )
```

Figure 6.5: Example for Verification property formula file

Especially, in the *Verification property formula file*, we use prefix paradigm to input the verification formal *f*, it is similar to the Yices input format.

Once we have already achieved these two files, we can use our verification tool to check the target system M with verification formula f under k-step. The launching command of our verification tool is "FPSBMC system.txt formula.txt bound k". The output of our verification tool is shown in figure 6.6.



GAE^{2}P

```
C:\FPSBMC\newtool1>FPSBMC1 system.txt formula.txt 4
--result:unsat

--Time cost:0.733   --check total amount of paths:3
```

```
C:\FPSBMC\newtool1>FPSBMC1 system.txt formula.txt 8
--result:unsat

--Time cost:0.678   --check total amount of paths:7
```

TAE^{2}P

```
C:\FPSBMC\newtool2>FPSBMC2 system.txt formula.txt 4
--result:unsat
unsat core ids: 1 2 3

--Time cost:1.163   --Trim times:2
```

```
C:\FPSBMC\newtool2>FPSBMC2 system.txt formula.txt 8
--result:unsat
unsat core ids: 1 2 3 4 5 6 7

--Time cost:0.654   --Trim times:4
```

Figure 6.6: The output of our verification tool

Chapter 7

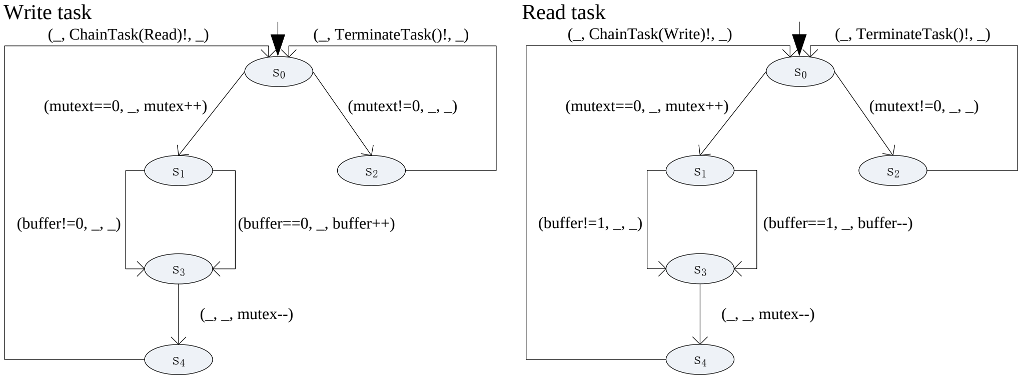
Experiments and evaluation

In this chapter, we firstly carry out some relevant experiments with our tools, then the evaluation and discussion for our verification approach based on experiments' results are demonstrated in the second part.

7.1 Experiments

In order to understand our experiments clearly, we chose a common Read/Write problem as our experiments example. In our Read/Write program, two tasks communicate with each other using a share global *buffer* as well as the variable *mutex* which is used to represent the mutual semaphores. Especially, in our example, tasks use service commands to request FPS for responding to its particular behavior, such as terminating itself and chaining a task. For the verification formula f, we considered whether the global *buffer* could satisfy the design requirement or not, the verification formula f and the Read/Write program were illustrated in figure 7.1.

Since our approach for the first time drew scheduler behaviors into the safety property checking of multi-task software, we only conducted the comparison between our two types of tools in our experiments. We have carried out three experiments to evaluate our tools with respect to different aspects. In the first experiment, we increased the bound k gradually with a fixed amount of tasks, the results of verification time under different bound k were shown in Table 7.1. However, in the most cases of practical checking process of multi-task software, only several important variables (which are relative to the checking process rather than all variables) were considered to be checked. As we can see in Table 7.1, there are only two variables in Read/Write program, hence, based on the first experiment, in the second experiment we inserted 1000 irrelative variables $x_1, x_2, \cdots, x_{1000}$ into V^{com} to express the execution behaviors which was unrelated to the checking property. The results of verification time were shown in Table 7.2. Furthermore, in the real execution process of multi-task software, a task cannot be run until it is dispatched by scheduler to get CPU. Then, simplily increasing the amount of tasks will not affect the verification process according to the characteristics of executing tasks of multi-task software. In the third experiment, we only increased the total amount of tasks



Verification formula: $\boldsymbol{G}$(buffer==0 ∨ buffer==1)

Task configure file:

Task Name	Priority	Autostart
Read	3	True
Write	2	False

Figure 7.1: Read/Write program and verification formula f

based on first experiment. The experiment results were shown in Table 7.3.

In our data tables, $RW(x, y)$ represents the Read/Write problem, where the parameters x and y denote the total number of *Read* and *Write* program separately. The symbol "p" denotes the amount of verification variables, "c" represents the total size of the test program's code, "v" is the total amount of variables of V^{com} and we use "p^c" to indicate the total amount of code associated with verification properties. Furthermore, we use symbol "r" to show the total amount of checking paths of GAE^{2}P under bound k and symbol "tr " to illustrate the total times of trim-tree of TAE^{2}P. The environment of experiments is: windows vista OS, AMD $64 * 2$ Dual Core CPU, 2.00 GB RAM.

Table 7.1: Fixed total amount of tasks, increase bound k

	Test program	Bound k	#p	#p^c / #c	#v	GAE^{2}P		TAE^{2}P	
						#r	Time(s)	#tr	Time(s)
1	RW(1,1)	5	1	4 / 18	2	5	0.101	3	0.082
2	RW(1,1)	10	1	4 / 18	2	15	0.139	6	0.136
3	RW(1,1)	15	1	4 / 18	2	31	0.249	8	0.203
4	RW(1,1)	20	1	4 / 18	2	63	0.511	10	0.421
5	RW(1,1)	25	1	4 / 18	2	191	1.791	13	1.058
6	RW(1,1)	30	1	4 / 18	2	511	6.857	16	4.742
7	RW(1,1)	35	1	4 / 18	2	1023	16.41	18	8.063
8	RW(1,1)	40	1	4 / 18	2	2047	46.09	20	16.63

Table 7.2: Insert 1000 irrelevant variables

	Test program	Bound k	#p	#p^c / #c	#v	GAE^{2}P		TAE^{2}P	
						#r	Time(s)	#tr	Time(s)
1	RW(1,1)	5	1	4 / 1018	1002	5	1.604	3	1.592
2	RW(1,1)	10	1	4 / 1018	1002	15	1.649	6	1.639
3	RW(1,1)	15	1	4 / 1018	1002	31	1.754	8	2.203
4	RW(1,1)	20	1	4 / 1018	1002	63	2.031	10	3.121
5	RW(1,1)	25	1	4 / 1018	1002	191	4.242	13	3.558
6	RW(1,1)	30	1	4 / 1018	1002	511	11.351	16	11.242
7	RW(1,1)	35	1	4 / 1018	1002	1023	21.22	18	22.43
8	RW(1,1)	40	1	4 / 1018	1002	2047	50.57	20	48.13

Table 7.3: Increase total amount of tasks based on experiment 1

	Test program	Bound k	#p	#p^c / #c	#v	GAE^{2}P		TAE^{2}P	
						#r	Time(s)	#tr	Time(s)
1	RW(1,1)	5	1	4 / 18	2	5	0.118	3	0.092
2	RW(2,1)	10	1	4 / 18	2	15	0.141	6	0.138
3	RW(3,2)	15	1	4 / 18	2	31	0.257	8	0.214
4	RW(4,3)	20	1	4 / 18	2	63	0.509	10	0.441
5	RW(5,4)	25	1	4 / 18	2	191	1.792	13	1.097
6	RW(6,5)	30	1	4 / 18	2	511	6.851	16	4.771
7	RW(7,6)	35	1	4 / 18	2	1023	16.49	18	8.611
8	RW(8,7)	40	1	4 / 18	2	2047	46.10	20	16.73

7.2 Evaluation and discussion

According to Table 7.1 in the experiment 1, we can find that the checking time of TAE^{2}P is shorter than GAE^{2}P under the same bound k and the total amount of tasks. Especially, we can notice that when the bound k is increased to 40, GAE^{2}P generated 2047 possible execution paths and TAE^{2}P trimmed 20 unreachable sub-paths over execution tree for getting one execution path. However, in the experiment 2, according to Table 7.1 and Table 7.2, we can notice that TAE^{2}P cannot hold its advantage any longer in verification time compared with GAE^{2}P when the bound k is increased to 30. The reason for this phenomenon is: (i) although GAE^{2}P translates all of execution paths into Yices, the SMT tool Yices not only is an efficient solver to compute the relation between verification property and transition system but also contains a specific processing method tailored for unsatisfiable cores that commonly removes the VCs which were not relevant to a given property from transition system [36]; (ii) TAE^{2}P spends much time to compute all the variables' values in each step trim-tree process. The compared results between experiment 1 and experiment 2 are shown in left side of figure 7.2.

In the experiment 3, we increased the amount of tasks compared with experiment 1, but the verification time only changed slightly according to table 7.1 and 7.3. The reason for this phenomenon is: although a system has many tasks, which task to be run is determined by FPS, furthermore, task's behaviors cannot be unfolded and inserted into execution tree until it occupies the *running unit*. The compared results between experiment 1 and experiment 3 are shown in right side of figure 7.2.

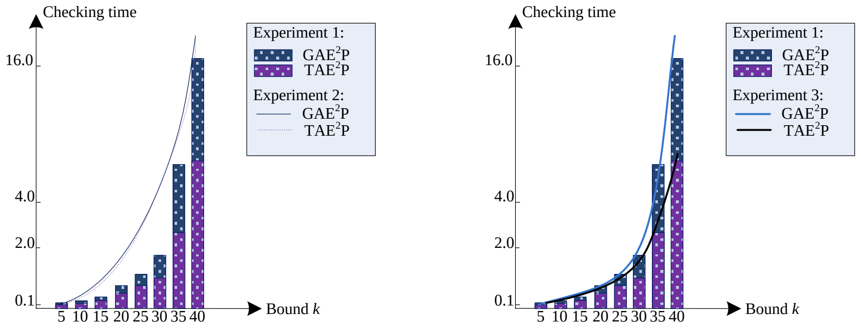


Figure 7.2: The compared results between experiment 1 and experiment 2/experiment 3

According to experiments results, we can know our verification approach has several advantages for checking the safety property of multi-task software based on OSEK/VDX OS, (i) in our approach, BMC is employed to overcome state space explosion; (ii) our verification approach can deal with the service commands that appear in any branches of tasks' behaviors; (iii) all of the k-step execution paths which are conducted by FPS can be obtained based on execution tree. However, there are several disadvantages in our approach, e.g., our approach only focuses on the basic task of OSEK/VDX OS, interruption behavior is not considered in our approach. Furthermore, our approach only can be used to check the safety property. However, the real-time property is a key property for the real-time operating system OSEK/VDX and needs to be checked too.

Chapter 8

Related work

SMT-based BMC is gaining popularity in the formal verification community due to the advent of sophisticated SMT solvers built over efficient SAT solvers [37]. Ganai and Gupta describe a verification framework for BMC which extracts high-level design information from an extended finite state machine (EFSM) and apply several techniques to simplify the BMC problem [5]. Based on the proposed verification framework, Ganai and Gupta develop a lazy method for modelling multi-threaded concurrent systems using shared variables [5]. Although, Ganai and Gupta's method can be used to check the safety property of multi-thread software in which threads concurrently execute in system, this method is restricted to two threads and the scheduler behaviors are not considered in verification process. Therefore, this method cannot be used to check the safety property of multi-task software in which tasks are dispatched by scheduler for executing.

Qadeer and Rehof also present a pragmatic method to discover bugs in concurrent software in which the program analysis is restricted to executions with a bounded number of context switches [38]. However, this method is incomplete since it considers the verification up to a given fixed context bound. This method also dose not draw the scheduler behaviors into the checking process.

Lucas Cordeiro and Bernd Fischer describe and evaluate three approaches to model check multi-threaded software with shared variables and locks using BMC based on SMT [7]. Especially, in this paper, authors use scheduler to reduce the reachability tree, the method of Lucas Cordeiro and Bernd Fischer is as follows:

$$\psi_k = \underbrace{I(s_0) \wedge R(s_0, s_1) \wedge ... \wedge R(s_{k-1}, s_k)}_{transition \quad system \quad M} \wedge \underbrace{\neg\phi_k}_{property} \wedge \underbrace{sch(s_0) \wedge ... \wedge sch(s_k)}_{scheduler} \tag{8.1}$$

However, in Lucas Cordeiro and Bernd Fischer's method, tasks and scheduler cannot communicate with each other using service commands.

To the best of our knowledge, there is no work that considers a SMT-based bounded model checking to check the multi-task software in which tasks are dispatched by fixed priority scheduler and tasks and scheduler can communicate with each other using service commands based on OSEK/VDX OS.

Chapter 9

Conclusion and future work

In our article, we proposed an approach about how to check the safety property of multi-task software in which tasks are dispatched by FPS based on OSEK/VDX OS. Especially, we established an k-step execution tree to represent the execution behaviors of tasks under FPS's dispatching. We also implemented two tools according to our two strategies of extracting execution paths in which BMC is employed to overcome state space explosion and generate the VCs based on execution tree. Furthermore, we evaluated our tools by carrying out some relevant experiments, the experiments results indicated that our tools can check the safety property of multi-task software in which tasks are dispatched by FPS efficiently. However, our research leaves something to be desired, e.g., our research only focus on the basic task of OSEK/VDX OS and our approach only can be used to verify the safety property of multi-task software. In future, we plan to conduct our research on extended task of OSEK/VDX OS based on the current work. Besides, the real-time property of multi-task software is also an aspect to be improved and investigated.

Acknowledgements

My master course at School of Information Science in JAIST will soon come to an end, at the completion of my graduation thesis; I wish to express my sincere appreciation to all those who have offered me invaluable help during the whole period of my study and research.

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Aoki, who led me into the world of Bounded Model Checking and gave me a chance to explore my potential in this research topic at the full capacity. He generously spent much time reading through each draft and walked with me through all the stages of this thesis. 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.

Secondly, I should give my hearty thanks to Professor Kokichi Futatsugi and Assistant Professo Yuki Chiba, for their valuable comments and suggestions. They have put considerable time and effort into their comments on my research. They gave me their help and helped me work out my problems during the difficult course of my research, from whom I benefited a lot. Also, I would like to thank my friends, Lin Wang, Min Zhang, 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 family 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.

Once again, I would like to thank my supervisor Professor Aoki, his rigorous, conscientious and earnest attitude to scholarly studies impressed me most, this would be an invaluable wealth for my future.

Reference

[1] C.Biere and J.-P. Katoen. *Principles of Model Checking*, MIT Press, 2008

[2] Ranjij Jhala. *Software Model Checking*, ACM, 2008

[3] A.Biere. *Bounded Model Checking*, In Handbook of Satisfiability, pp.457-481, 2009

[4] Armin Biere, Alessandro Cimatti, and Edmund M. Clarke. *Bounded Model Checking*, Advances in Computers, vol.58, 2003

[5] M. K. Ganai and A. Gupta. *Efficient Modeling of Concurrent System in BMC*, SPIN, LNCS 5156, pp.114-133, 2008

[6] V. Kahlon. *Semantic Reduction of Thread Interleaving in Concurrent Program*, TACAS, LNCS 5505, pp.124-138, 2009

[7] Lucas Cordeiro and Bernd Fischer. *Verfiying Multi-threaded Software using SMT-based Context-Bounded Model Checking*, ICSE, 2011

[8] L. Cordeiro, and B. Fischer. *SMT-based Bounded Model Checking for Embedded ANSI-C software*, ASE, pp.137-340, 2008

[9] N. Ghafari, A Hu, and Z. Rakamaric. *Context-bounded Transitions for Concurrent Software: An empirical evaluation*, SPIN, LNCS 5156, pp.398-413, 2008

[10] Johnson Lemieux. *Programming in the OSEK/VDX Environment*, CMP Books, 2011

[11] E. Alkassar, N. Schirmer, and A. Starostin. *Formal pervasive verification of a paging mechanism*, In TACAS, pp.109-123, 2008

[12] In Yices, *http://yices.csl.sri.com*

[13] S. Buss. *Intuitionistic validity in T-normal Kripke structure*, Annals of Pure and Applied Logic, pp.159-173, 1993

[14] Jeff magee and Jeff kramer. *Concurrency*, 2006

[15] H. Barringer, M. Fisher, D. Gabbay, and G. Gough. *Proceedings of the Second International Conference on Temporal Logic*, ICTL, 1997

[16] D.M. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*, Oxford Logic Guides, Oxford University Press, 1994

[17] R. Goldblatt. *Logics of Time and Computation*, CSLI Lecture Notes, Center for the Study of Language and Information, Stanford University, second edition, 1987

[18] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. *Chaff: Engineering an Efficient SAT Solver*, In DAC, 2001

[19] M. Prasad, A. Biere, and A. Gupta. *A survey of recent advances in sat-based formal verification*, In STTT, 2005

[20] Raimondi, F. and Lomuscio, A. *Automatic verification of multi-agent systems by model checking via OBDDs*, Journal of Applied Logic, 2005

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

[22] SATLib, *http://www.satlib.org/*

[23] PROVER, *http://www.prover.com/*

[24] SATO, *http://homepage.cs.uiowa.edu/ hzhang/sato/*

[25] GRASP *http://vlsicad.eecs.umich.edu/BK/Slots/cache/sat.inesc.pt/ jpms/grasp/*

[26] BNF, *http://en.wikipedia.org/wiki/BNF*

[27] A. Armando, J. Mantovani and L. Platania. *Bounded Model Checking of Software using SMT Solvers instead of SAT Solvers*, LNCS, pp.146162, 2006

[28] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Ranise and R. Sebastiani. *Efficient satisfiability modulo theories via delayed theory combination*, Computer-Aided Verification (CAV), 2005

[29] SMTlib, *http://www.smtlib.org/*

[30] Yatake, k. and Aoki, T. *Automatic generation of model checking scripts based on environment modeling*, in model checking software, pp.58-75, 2010

[31] Jiang Chen and Toshiaki Aoki, *Conformance testing for OSEK/VDX OS using model checking*, Asia-Pacific Software Engineering Conference, 2011

[32] Hash, *http://en.wikipedia.org/wiki/Hash*

[33] Queue, *http://www.cplusplus.com/reference/stl/queue*

[34] Tarjan, R.E. *Depth-first search and linear graph algorithm*, SIAM journal of computing, pp.146-160, 1972

[35] Stack, *http://en.wikipedia.org/wiki/Stack*

[36] Lucas Cordeiro. *SMT-Based bounded model checking for multi-threaded software in embedded system*, ICSE, 2010

[37] L. M. de Moura and N. Bjrner. *Z3: An efficient SMT solver*. TACAS, LNCS 4963, pp.337340, 2008

[38] S. Qadeer and J. Rehof. *Context-bounded model checking of concurrent software.* TACAS, LNCS 3440, pp.93107, 2005