LETTER

Model Checking an OSEK/VDX-Based Operating System for Automobile Safety Analysis*

Yunja CHOI[†a)], ***Member***

SUMMARY An automotive operating system is a typical safety-critical software and therefore requires extensive analysis w.r.t its effect on system safety. Our earlier work [1] reported a systematic model checking approach for checking the safety properties of the OSEK/VDX-based operating system Trampoline. This article reports further performance improvement using embeddedC constructs for efficient verification of the Trampoline model developed in the earlier work. Experiments show that the use of embeddedC constructs greatly reduces verification costs.

key words: *OSEK/VDX, Trampoline, model checking, safety analysis*

1. Introduction

An operating system is a representative safety-critical system, since a fault in an automotive operating system can result in catastrophic failures of the automobile. A comprehensive safety analysis is a must, not only at the overall system level, but also at the software level. Model checking [3], one of the most commonly used automated formal verification techniques, has been applied for the verification of operating systems [4]–[6]. Nevertheless, existing works are either limited to small-scale operating systems such as TinyOS or focus on specific aspects of a system, such as timing and scheduling analysis.

Our earlier work [1] reported a case study for model checking the safety properties of Trampoline [7], an operating system based on the OSEK/VDX international standard [2]. By adopting a faithful model translation from the kernel code and thus avoiding aggressive abstractions, the approach made the comprehension of counterexamples straightforward, reducing the time needed for manual analysis. The faithful translation approach, however, naturally suffers from high verification cost. This paper presents a method for further improving the verification performance by utilizing the embeddedC constructs in Promela. Experiments show that the use of embeddedC constructs reduces the number of states and transitions to be traversed during the model checking process, resulting in greatly reduced verification costs.

This paper briefly reviews the model checking approach presented in [1] and then explains how the former approach is extended for better performance together with the experimental evidence showing its positive impact.

Manuscript received May 31, 2012.
Manuscript revised October 17, 2012.
[†]The author is with School of CSE, Kyungpook National University, South Korea.

a) E-mail: yuchoi76@knu.ac.kr
DOI: 10.1587/transinf.E96.D.735

2. Model Checking Safety Properties

The safety properties are identified by means of a Software Fault Tree Analysis (SFTA), starting from system-level safety requirements, such as "an automobile control system shall change the direction of the wheels in time". Each safety requirement is analyzed in a top-down manner to identify detailed potential software faults such as "delay in task sequencing". Each such software fault is further analyzed to identify operating system level safety properties by combining the top-down analysis with the identification of safety requirements from the international standards OSEK/VDX and AUTOSAR to reduce the depth of the software fault tree; in our study, a total of 56 safety properties were identified from three software fault trees with an average depth of six. The two safety properties identified from the SFTA are listed in the following:

SR1. Tasks shall not be in the waiting state indefinitely while allocating resources.
SR2. Tasks shall not wait for events indefinitely.

Figure 1 illustrates our model checking process in three steps: (1) light-weight model checking, (2) incremental verification, and (3) performance improvement using embeddedC, which is an extension from the earlier work.

2.1 Light-Weight Model Checking

In the first step, a formal model (the Promela model in this case study) is constructed from the Trampoline kernel code, which is validated using the SPIN simulator and then verified with respect to each safety property using the SPIN model checker. Our model construction process eliminates

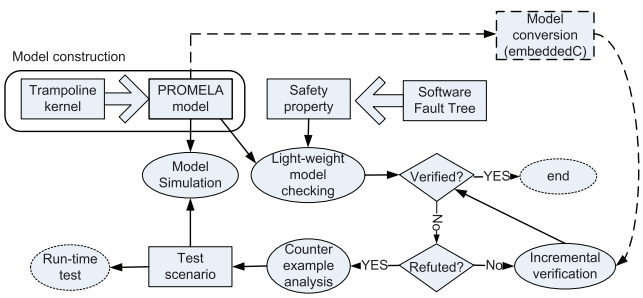


Fig. 1 Verification process for Trampoline.

wrapper functions, tracing functions, and emulating functions from the code. As a result, the original Trampoline operating system comprising a total of 4,530 lines of code is converted into a PROMELA model comprising a total of 1,500 lines of code. The model checker generates a counterexample for each refuted safety property. We analyze this using counterexample replay in the SPIN simulator and use it to generate a test scenario so that we can confirm through runtime testing that the counterexample trace actually leads to a safety violation. In this step, light-weight model checking is applied using limited resources, e.g., 4 Gbytes of system memory. The process ends if all the properties are verified in this step.

2.2 Incremental Verification

However, it is possible that the model checker neither verifies nor refutes the property due to a lack of resources. The second step tries to address this issue by using incremental verification, which imposes constraints on the external model (i.e., the task model) of the operating system and performs incremental verification as the constraints are gradually lifted. This provides a method for quantitatively measuring the verification coverage even when comprehensiveness cannot be achieved.

2.3 Performance Improvement Using EmbeddedC

Even incremental verification might not scale in case the number of application tasks is over the threshold. The third step utilizes the embeddedC constructs in Promela to reduce the number of states and transitions to be traversed during the model checking process. This step is applied only to those properties that are not verified or refuted in the first and the second step, due to the issue of usability; SPIN does not provide simulation capabilities for models using embeddedC constructs, and thus it is extremely difficult to validate models with embeddedC constructs in the initial step.

3. A Safety Bug Found

Our case study revealed that Trampoline violates $SR2$. A counterexample trace that leads to an abnormal system halt is illustrated in Fig. 2: Task t_1 activates task t_2, which preempts t_1 as soon as it is activated. t_2 activates task t_3 and then waits for event 2. While t_2 is in the waiting state, t_3 goes to the running state, since it has higher priority than t_1, and activates task t_4. t_3 is preempted by t_4, but is soon resumed after t_4 goes to the waiting state for event 0. t_3 sets event 0 for t_4 and terminates afterwards. Setting event 0 is supposed to resume t_4, but t_1 is resumed instead. Both t_2 and t_4 remain in the waiting state indefinitely.

The counterexample trace is applied directly to the Trampoline kernel code for analysis since the mapping between the kernel and the Promela model is one-to-one, making the analysis straightforward. It turns out that the problem is due to the encoding and checking mechanism for

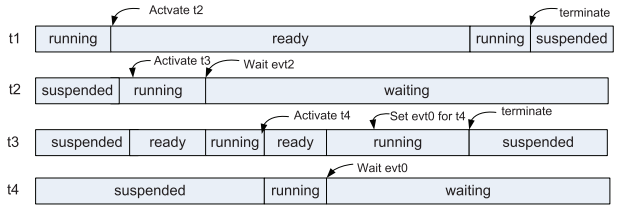


Fig. 2 A counterexample of SR_2.

events in the Trampoline kernel code:

```
1:  tpl_status
      tpl_set_event(tpl_task_id task_id,
                    tpl_event_mask in_event){
2:      ....
3:    if((events->evt_wait & in_event)!=0){
4:        ....
5:      // wake up and put the waiting process
        //in the ready queue
6:          ...
7:    }  ...
8:  }
```

As stated in line 3 of the code, *tpl_set_event* performs a bitwise-and operation of the event mask and the event number to check that the event number is indeed on the waiting list. However, this encoding and checking mechanism only works correctly when the event number is greater than 0; $tpl\_set\_event(t_4, 0)$ does not have any effect on the event mask since the bitwise-and operation of the event mask and the event number are always equal to 0. In this case, the lines between 4 and 6 are not executed, and thus cannot wake up the task waiting for event 0. Trampoline does not provide any means for checking or warning that the value of an event is not supposed to be equal to zero. We anticipate that this is a typical case of a safety gap; bad things may happen where we take things for granted.

4. Incremental Verification

Though the initial verification was successful in finding a safety bug for $SR2$, it failed to verify or refute $SR1$. SPIN quickly ran out of memory on a SUN workstation with 30 Gbytes of memory. Even when using the bit-state hashing option provided by SPIN for better scalability, the measured coverage was around 20%, which is far from being comprehensive. We anticipate that the inefficiency of the initial verification comes from two factors: (1) The Trampoline kernel itself is too large in the statespace, and (2) the task model is too generous in that it allows an arbitrary number of system calls per task, which is not realistic in practice.

We tackle the second issue by adopting an incremental verification approach that limits and increases the number of system calls per task as verification succeeds. In this way, we still allow arbitrary behavior of user tasks up to a certain point while providing comprehensive verification under the given constraints. Table 1 shows the performance of model checking $SR1$ as the number of system calls per task increases from 3 to 11. The columns from left to right

Table 1 Performance of incremental verification.

APIs	Depth	States	Transitions	Memory	Time
3	35,159	8.00e+06	1.26e+07	1019.498	289
5	1,041,301	4.20e+07	6.57e+07	3318.760	1.37e+03
7	2,180,863	1.12e+08	1.75e+08	8555.065	3.65e+03
9	3,396,568	1.92e+08	2.97e+08	13,418.443	6.20e+03
11	4,934,305	3.02e+08	4.67e+08	28,229.922	9.90e+03

represent the number of API calls, the depth of the verification search, the number of states explored, the number of transitions, the amount of memory used in Megabytes, and the time required to finish verification in seconds.

5. Performance Improvement Using EmbeddedC

Though the experimental result is promising, it was performed with limited environmental parameters; the number of tasks was limited to four, with the maximum number of API calls per task being limited to 11. We suggest utilizing embeddedC constructs to achieve better performance.

5.1 EmbeddedC Constructs

EmbeddedC constructs were introduced to facilitate model-driven verification of software systems, making it possible to directly embed implementation code into Promela models [8]. In this way, high-level system design can be modeled independent of implementation details, which can be embedded later and replaced whenever necessary [9].

The nature of the constructs and the way they are handled in the model checking process have an interesting side effect: The execution of the embedded code is invoked during the model checking process, which is treated as a single transition no matter how many lines of code are embedded inside. In contrast, a typical model checking process treats each statement as a source of a new transition, producing a large number of intermediate states. For example, Fig. 3 illustrates two Promela models for the same C program code. The model in the first column is a model that uses no embeddedC constructs and the model in the second column is a model that uses embeddedC constructs.

Table 2 shows the performance comparison when model checking the two models in Fig. 3, as the number of global variables increases from two to five in lines 1–2 and their corresponding assignment statements are added in lines 9–12. From columns two to six, Table 2 shows the number of global variables, the search depth, the number of states traversed, the number of transitions traversed, and the amount of memory in Mega bytes consumed during the model checking process. The first model is a Promela model that uses atomic sequences (PWA), and the second model is a Promela model that uses embedded C constructs (PWE). Note that the model using embedded C requires far fewer resources than the one using the original Promela model and that the verification cost does not increase as the number of variables increases in the models using embedded C constructs.

Pure Promela model	Using embedded C

1: int q=1000;
2: int p[1];
3: chan c=[1] of {bit};
4: active proctype X(){
5: start:
6: atomic{
7: p[0]=0;
8: do
9: :: d_step{p[0]<q -> p[0]++; }
10: :: else -> break;
11: od;
12: }
13: assert(p[0] <= q);
14: c!1;
15: end: goto start;
16: }
17: active proctype Y(){
18: start:
19: c??1;
20: atomic{
21: do
22: :: d_step{ p[0] >0 -> p[0] --; }
23: :: else -> break;
24: od;
25: }
26: assert(p[0] <=q);
27: end: goto start;
28:}
	c_code{ int q=1000;
	int p[1];
	};
	c_track "p" "sizeof(int)"
	c_track "&q" "sizeof(int)"
	chan c=[1] of {bit};
	active proctype X(){
	start:
	c_code{
	p[0]=0;
	while(p[0]<q){
	p[0]++;
	}
	};
	assert(c_expr(p[0] <= q));
	c!1;
	end: goto start;
	}
	active proctype Y(){
	start:
	c??1;
	c_code{
	while(p[0] >0){
	p[0] --;
	}
	};
	assert(c_expr(p[0] <=q));
	end: goto start;
	}

Fig. 3 Sample models.

Table 2 Performance comparison of the sample models.

model	vars	depth	states	transitions	memory
PWA	2	9,057	50	82	2.64
	3	9,062	65	108	2.73
	4	9,067	81	133	2.73
	5	9,072	96	159	2.83
PWE	2,3,4,5	25	38	58	2.54

5.2 Application of EmbeddedC to the Trampoline Model

We anticipate that embeddedC is applicable not only for model-driven verification, but also for improving model checking scalability, especially for synchronized event processing systems with a large number of global variables, which is a common characteristic of embedded software.

As illustrated in Fig. 1, the Promela model constructed from the Trampoline kernel is partly converted into a model with embeddedC constructs in the third step, which is used for incremental verification afterwards. This partial model conversion is performed according to the following steps:

1. Convert the atomic sequence of statements into *c_code* blocks for synchronized event-driven processes.
2. Embed all global variables referenced or used from the converted *c_code* blocks into *c_code* blocks.
3. Embed all user-defined data types used in the *c_code* blocks into *c_decl* declarations.
4. Track each global variable declared in *c_code* blocks using the *c_track* construct.

The conversion process converts only the atomic sequence of statements and related variables into embededC. This preserves the semantics of the original model since an

atomic sequence is executed without interrupts, and thus there is no need to handle them as individual transitions.

Figure 4 shows a fragment of the Trampoline model with embeddedC code, converted from the original Promela model. The atomic sequence in the inline function is converted into a *c_code* block, the global variables *tpl_fifo_rw* and *tpl_h_prio* accessed from the *c_code* block are declared in a *c_code* block, and then the user-defined data type *TPL_FIFO_STATE* is declared in a *c_decl* block. Finally, the global variables are traced by using the *c_track* construct. We note that the multiple transitions required by the inline function *tpl_get_proc* are reduced to one by collapsing them into one *c_code* construct.

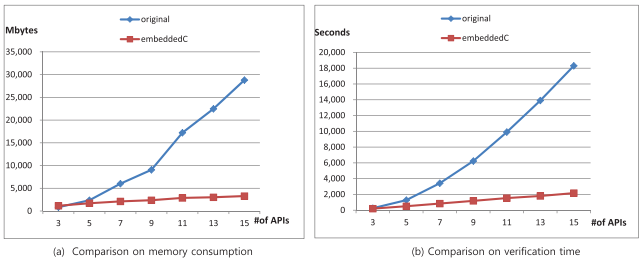
5.3 Performance Improvement

Figure 5 illustrates the performance difference between the original Trampoline kernel model and the model with embeddedC constructs in terms of memory and time consumption as the number of API calls per task increases from 3 to 11. We see that both the absolute value of the verification cost and the rate of the cost increments are greatly reduced as the number of API calls increases.

Pure Promela model	Using embedded C

typedef struct TPL_FIFO_STATE{
  unsigned read;
  unsigned size;
} TPL_FIFO_STATE;
 	
TPL_FIFO_STATE tpl_fifo_rw[n];
short tpl_h_prio = -1;
 	
inline tpl_get_proc(){
  atomic{
  read_idx =
  tpl_fifo_rw[tpl_h_prio].read;
  …
  do
  :: tpl_h_prio>=0 &&
  tpl_fifo_rw[tpl_h_prio].size==0
  -> tpl_h_prio--;
  :: else -> break;
  od;
  }
}
	c_decl{
	typedef struct TPL_FIFO_STATE{
	  unsigned read;
	  unsigned size;
	} TPL_FIFO_STATE;
	}
	
	c_code{
	  TPL_FIFO_STATE tpl_fifo_rw[n];
	  short tpl_h_prio = -1;
	};
	c_track "tpl_fifo_rw"
	  "sizeof(TPL_FIFO_STATE)*n"
	c_track "&tpl_h_prio"
	  "sizeof(short)"
	
	inline tpl_get_proc(){
	  c_code{
	  read_idx =
	  tpl_fifo_rw[tpl_h_prio].read;
	  …
	  while(tpl_h_prio>=0 &&
	  tpl_fifo_rw[tpl_h_prio].size==0)
	  tpl_h_prio--;
	
	  };
	}

Fig. 4 Conversion example from the Trampoline OS.



(a) Comparison on memory consumption (b) Comparison on verification time

Fig. 5 Performance comparison.

6. Conclusion

This case study indicates that model checking can be effective in identifying subtle safety issues even if aggressive abstractions are not employed in model construction. Our approach of faithfully translating the kernel code into a formal model is systematic and allows engineers to easily understand the counterexamples. The scalability issue, which is aggravated by the faithful translation, is handled through incremental verification and the reverse use of embeddedC constructs. The embeddedC constructs were extremely useful for systematically reducing the verification cost. Nevertheless, the approach must be used with care since merging multiple transitions may result in the obfuscation of the important interleaving behavior of asynchronous processes. The application of embeddedC constructs was limited to atomic statements only in our case study for this reason. It was also delayed till the third step because merging transitions makes it difficult to analyze counterexamples and simulation results; the original Promela model is preferred for initial and incremental verification as long as the available resources allow this. EmbeddedC is the last choice for better scalability.

Acknowledgements

This work was partially supported by Kyungpook National University Research Fund, 2012 and the National Research Foundation of Korea Grant funded by Korean Government (2012R1A1A4A01011788).

References

[1] Y. Choi, "Safety analysis of the Trampoline OS using model checking: An experience report," Proc. 22nd IEEE International Symposium on Software Reliability Engineering, 2011.
[2] "OSEK/VDX operating system specification 2.2.3."
[3] G.J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley Publishing Company, 2003.
[4] J. Penix, W. Visser, S. Park, C. Pasareanu, E. Engstrom, A. Larson, and N. Weininger, "Verifying time partitioning in the DEOS scheduling kernel," Formal Methods in Systems Design Journal, vol.26, no.2, pp.103–135, March 2005.
[5] D. Bucur and M. Kwiatkowska, "On software verification for sensor nodes," J. Systems and Software, vol.84, no.10, Oct. 2010.
[6] L. Waszniowski and Z. Hanzálek, "Formal verification on multitasking applications based on timed automata model," Real-Time Systems, vol.38, pp.39–65, 2008.
[7] "Trampoline – opensource RTOS project." http://trampoline.rts-software.org.
[8] G.J. Holzmann and T.C. Ruys, "Effective bug hunting with Spin and Modex," Model Checking Software: The SPIN Workshop, April 2005.
[9] G.J. Holzmann, R. Joshi, and A. Groce, "Model driven code checking," Automated Software Engineering, pp.283–297, 2008.