OS-in-the-loop CEGAR for Multitasking Embedded Control Software

Dongwoo Kim, Yunja Choi

kdw9242@gmail.com

소프트웨어재난연구센터

KNU KYUNGPOOK NATIONAL UNIVERSITY

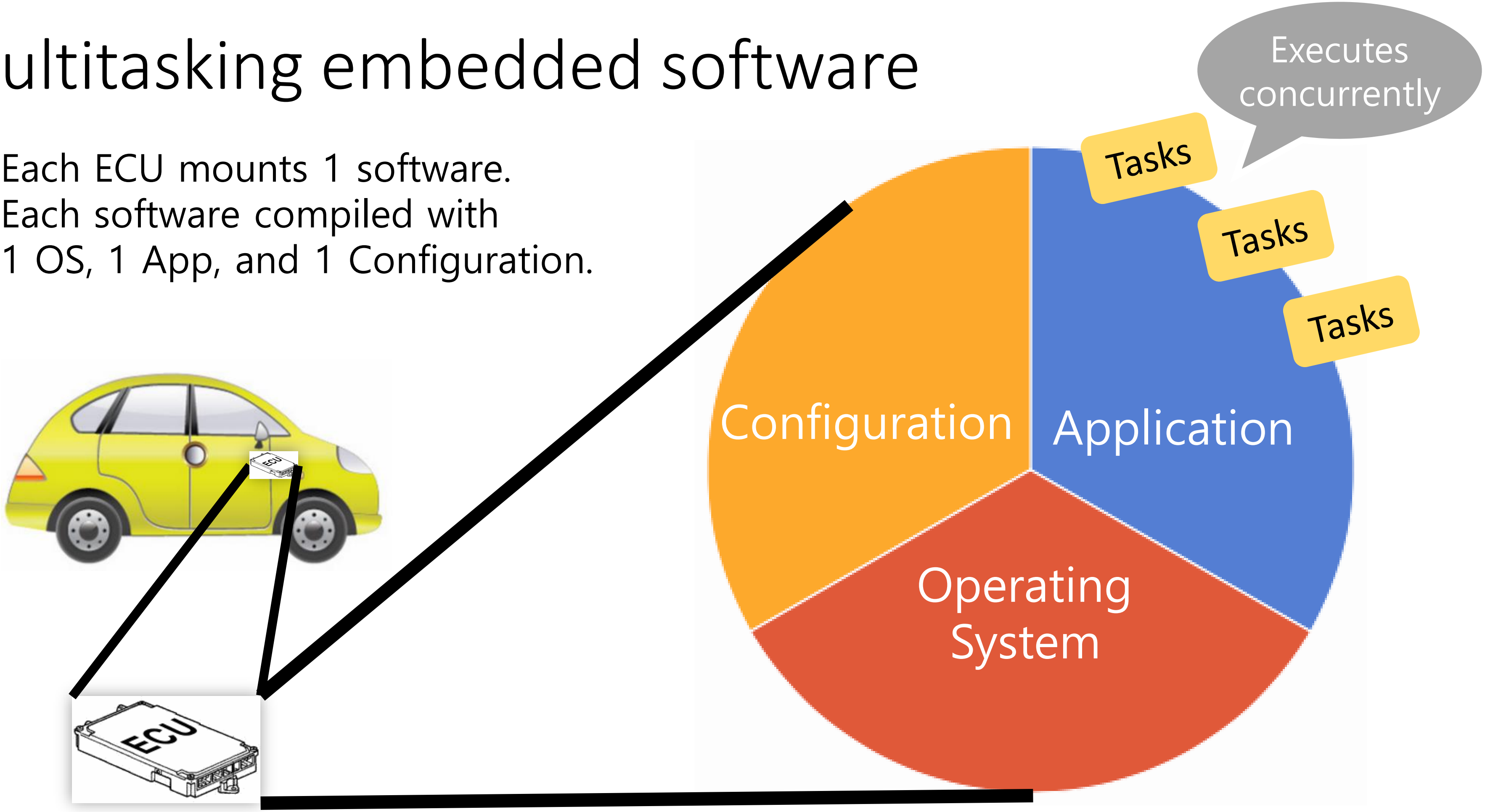
Software Safety Engineering LAB

Outline:

- Background
  - Multitasking embedded software
  - Model checking
- Limitations of existing methods
- The proposed verification method: OiL-CEGAR
  - Formal OS model and
  - OiL-CEGAR process
- Experiments
- Conclusion & Future work

Multitasking embedded software

- Each ECU mounts 1 software.
- Each software compiled with
  1 OS, 1 App, and 1 Configuration.

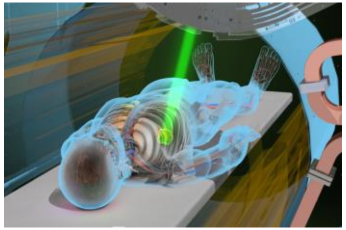


A car has hundreds of ECUs

Verification of multitasking software

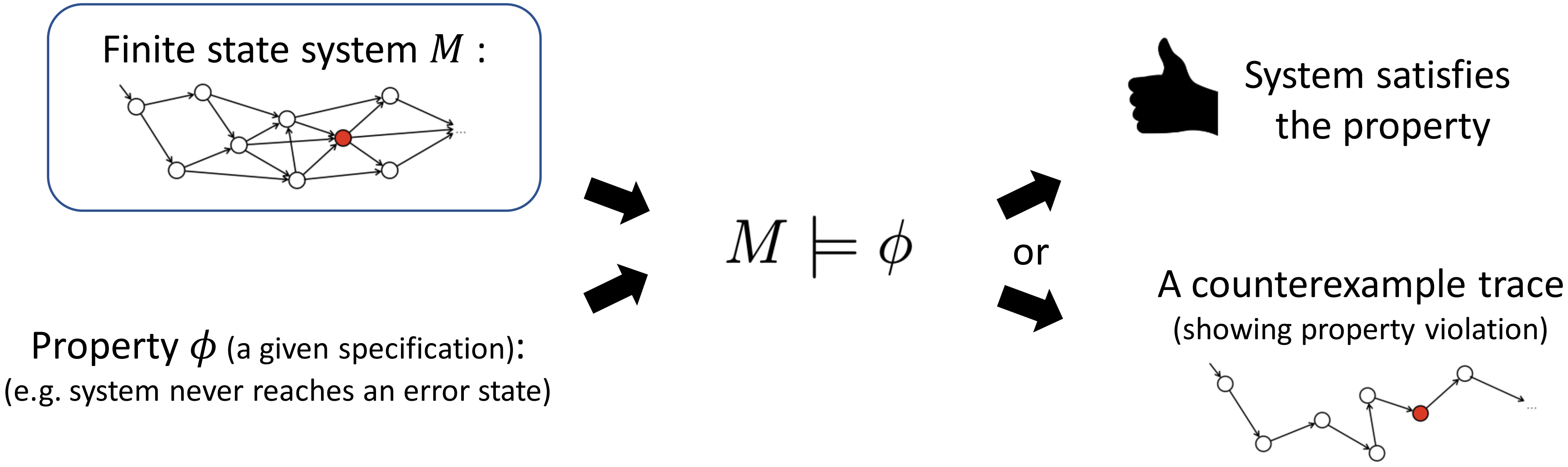
- Multitasking is used in most embedded software
  - ✓ usually written in C language
  - ✓ <u>uses multiple tasks</u>
  - ✓ e.g., brake pedals, engines, sensors, actuators, etc.
  - ✓ safety-critical
  - ✓ require comprehensive verification

- Model checking is suitable for comprehensive verification
  - ✓ rigorously verify software systems



Model checking

- Method for checking whether model M meets a given specification ϕ.



Finite state system M :

Property ϕ (a given specification):
(e.g. system never reaches an error state)

$$M \models \phi$$

or

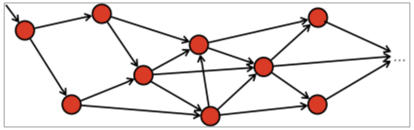
System satisfies the property

A counterexample trace
(showing property violation)

- Model checking can be applied to a model or a **program code** `(C, Java, etc)`
- However, model checking on multitasking embedded software is **very challenging**.

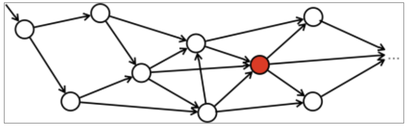
Properties

- Boolean property (invariants)
  (it should be satisfied in all states)



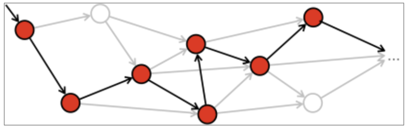
e.g., The running state should never be reached

- Assertion property
  (it should be satisfied in a state)



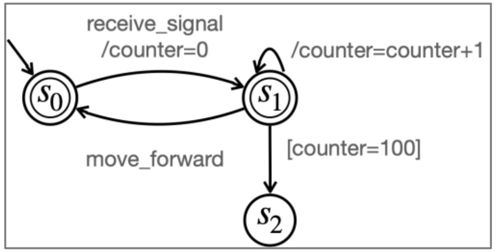
e.g., Variable v cannot have value after statement 32

- Temporal property (specifies dynamic behavior)
  (it should be satisfied in every path)



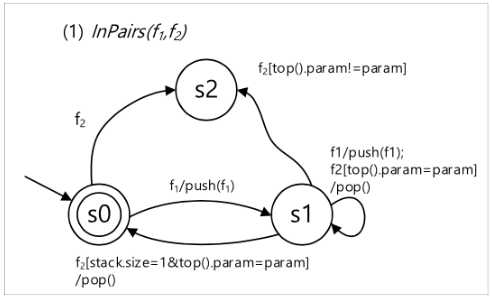
e.g., All task must be ready eventually

- Monitoring automata
  (it should not remain in error states, infinitely)



e.g., When it receives a forward signal, it must move forward in 100 ticks.

- API-call constraint
  (a type of monitoring automata having API-call events)

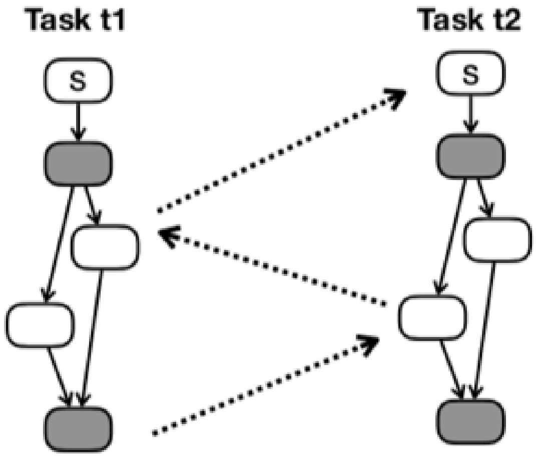


e.g., API-calls f1 and f2 shall be called in pairs.

Limitation: Model checking multitask program code with OS

- An OS implementation and application program code are can be directly verified.

Operating System Code + Application Code = Embedded System Code



Problems:

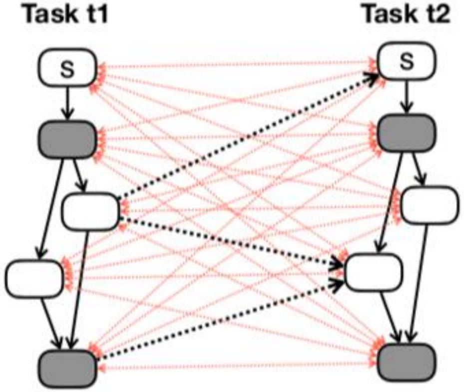
1. An enormous load of verification cost is required as it consumes time and memory exponential to the size of the program.
2. Usually, performs bounded model checking (cannot verifies the whole system)

- X. Zhu, M. Zhang, J. Guo, X. Li, H. Zhu, and J. He, "Toward a unified executable formal automobile OS kernel and its applications," IEEE Transactions on Reliability, 2018.

Limitation: Model checking multitask program code w/o OS

- Complexity can be reduced by using highly abstracted OS. (allow all possible context switch)
- Most of reported traces are **false alarms** having incorrect task execution order

Highly abstracted operating system + Application Code = Embedded System Code



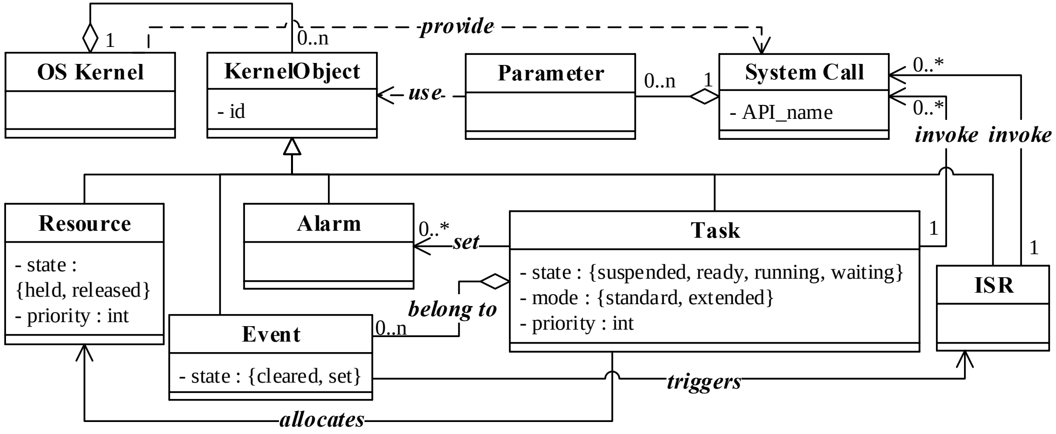
Problems:

1. False-alarm will be reported as there are no scheduling policy considered
2. Also, performs bounded model checking (cannot verifies the whole system)

- E. Clarke, et al., "Behavioral consistency of c and verilog programs using bounded model checking," in Proceedings of the 40th Annual Design Automation Conference, 2003
- L. Yin, et al., "Scheduling constraint based abstraction refinement for multi-threaded program verification," CoRR, vol. abs/1708.08323, 2017.
- O. Inverso, et al., "Bounded model checking of multi-threaded c programs via lazy sequentialization," in International Conference on Computer Aided Verification, 2014
- A. Gupta, et al., "Predicate abstraction and refinement for verifying multi-threaded programs," in Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2011
- T. A. Henzinger, et al., "Lazy abstraction," in Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2002

Necessity of operating system

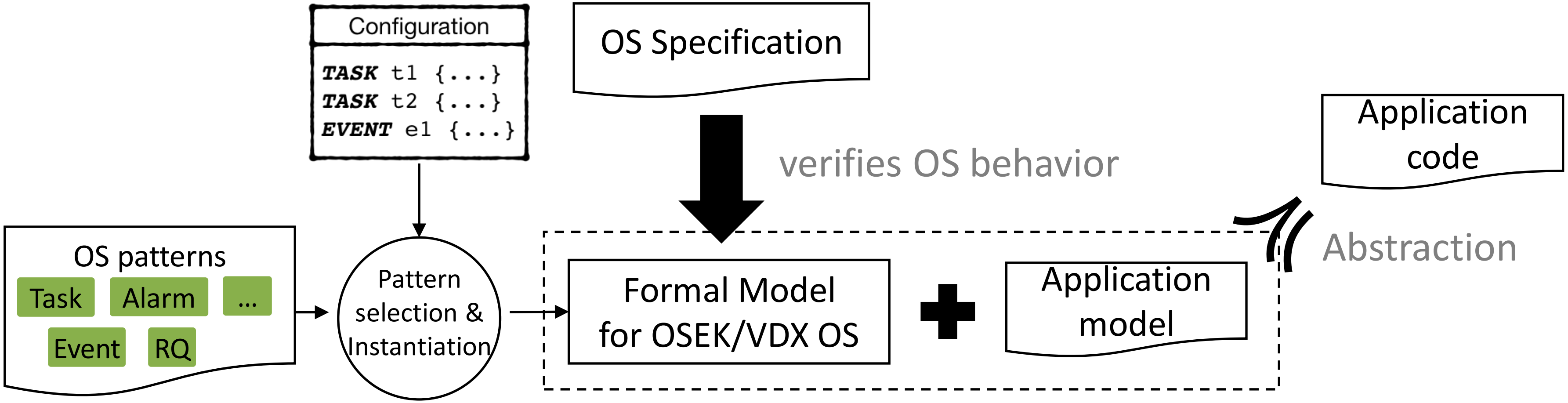
- Task scheduling involves multiple objects of the OS kernel, including
  - Tasks, API functions, resources, events, alarms, and ISRs, etc.
- A sound OS is necessary to improve the verification accuracy



Structure of an embedded OS (OSEK/VDX OS)

Insight: use of a sound OS model

- **OS model** correctly schedule an application and remove false alarms
- Model-level verification is efficient as it exclude all the details of programming language.
- Modeling language supports for concurrency, atomicity, and blocking.



Configuration

```
TASK t1 {...}
TASK t2 {...}
EVENT e1 {...}
```

OS Specification

verifies OS behavior

Application code

Abstraction

OS patterns

Task | Alarm | ... | Event | RQ

Pattern selection & Instantiation

Formal Model for OSEK/VDX OS

Application model

<Application verification>

* Y. Choi, "A configurable V&V framework using formal behavioral patterns for OSEK/VDX operating systems," Journal of Systems and Software, 2018.

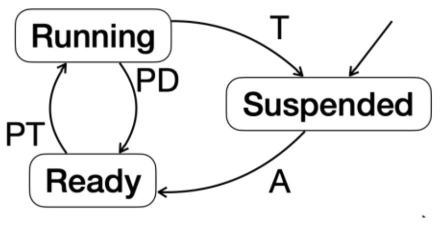
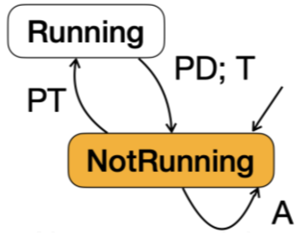
List of sound OS models

- G. Klein et al., “seL4: Formal verification of an OS kernel,” in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp.207–220.
- H. Zhang, G. Li, Z. Cheng, and J. Xue, “Verifying OSEK/VDX automotive applications: A spin-based model checking approach,” STVR, 2018.
- Y. Huang, et. al, “Modeling and verifying the code-level OSEK/VDX operating system with CSP,” in 2011 Fifth International Conference on Theoretical Aspects of Software Engineering, 2011, pp. 142–149.
- **Y. Choi, “A configurable V&V framework using formal behavioral patterns for OSEK/VDX operating systems,” Journal of Systems and Software, vol. 137, pp. 563–579, 2018.**
- J. Bengtsson, et al. "UPPAAL—a tool suite for automatic verification of real-time systems." International Hybrid Systems Workshop, 1995.
- X. Zhu, M. Zhang, J. Guo, X. Li, H. Zhu, and J. He, “Toward a unified executable formal automobile OS kernel and its applications,” IEEE Transactions on Reliability, 2018.

- The correctness of generated OS model is verified based on the OSEK/VDX specification.

Insight: use of a sound OS model (cont.)

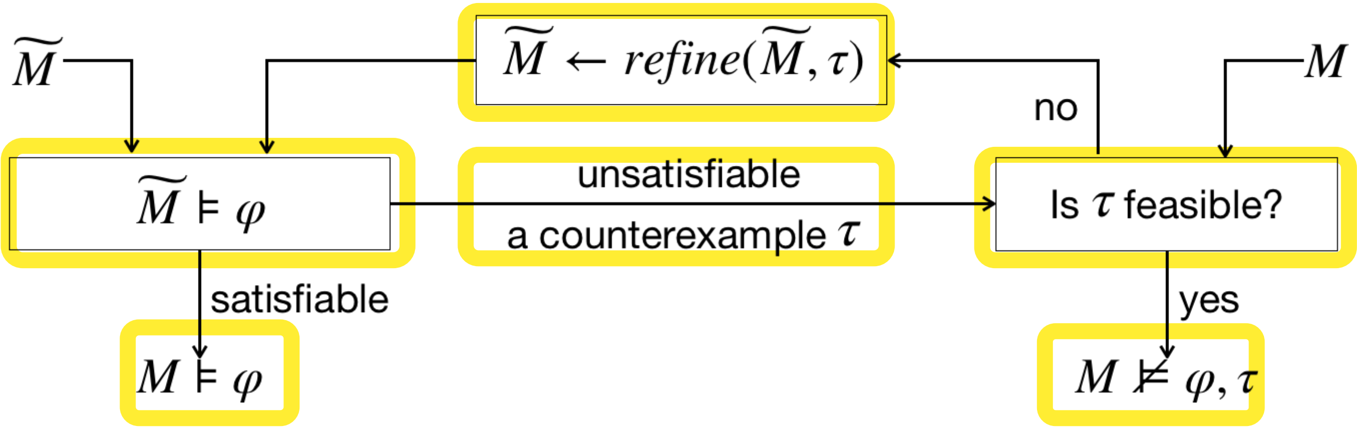
- The application code has to be translated into an application model.
- Informally, abstraction groups a set of states into a state.
- **Abstraction** is necessary due to the heterogeneity (between languages used for a model and program code)
    - \+ Reduce verification complexity
    - – Results in **high false alarm rate**

Concrete model	Abstract model
	
Traces reaching Running: • $SUS \xrightarrow{A} RDY \xrightarrow{PT} RUN$	Traces reaching Running: • $NOT \xrightarrow{PT} RUN$ • $NOT \xrightarrow{A} NOT \xrightarrow{PT} RUN$ • $NOT \xrightarrow{PT} RUN \xrightarrow{PD} NOT \xrightarrow{PT} RUN$ • …

- False alarms shall be automatically identified and removed.

Counterexample-Guided Abstraction Refinement (CEGAR)

Edmund Clarke, et al., "Counterexample-guided abstraction refinement", CAV 2000.

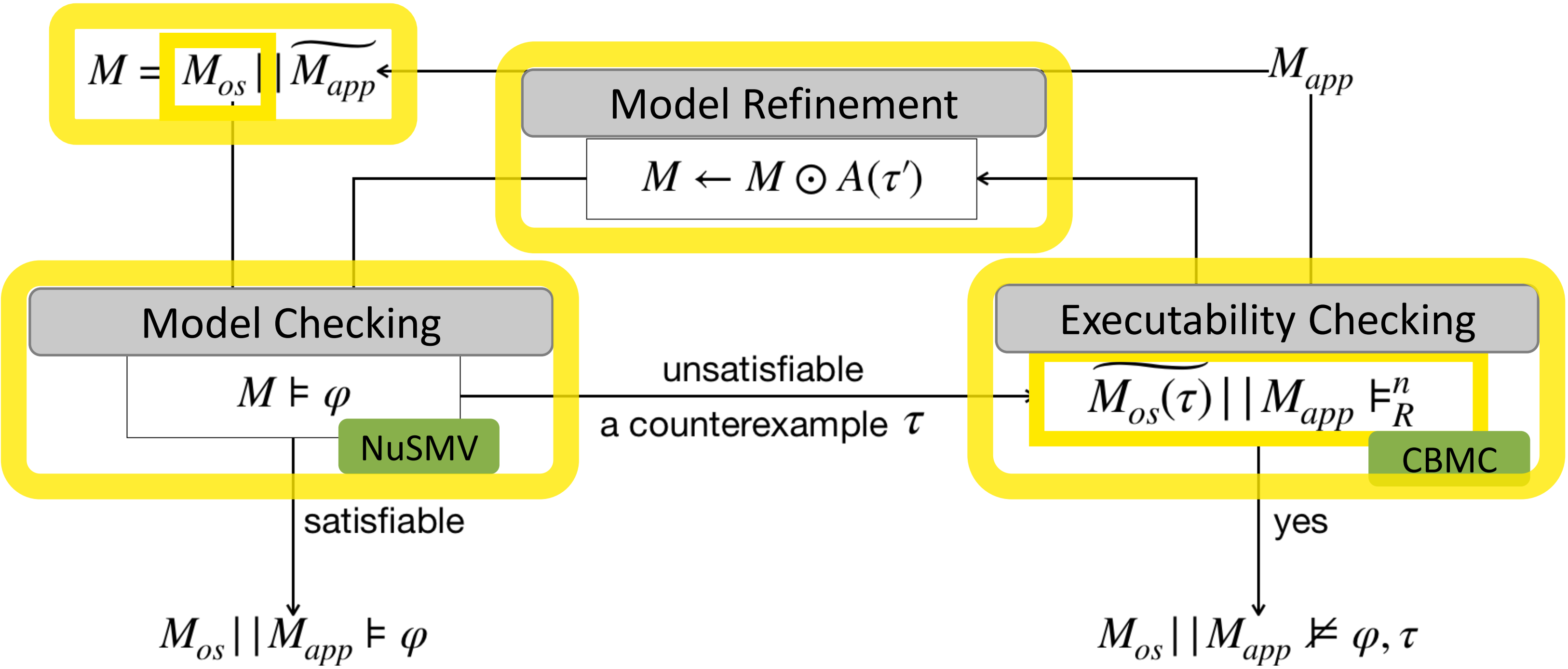


- Benefits
  - Scalable
  - Automated false alarm reduction

- Problems
  - Certain types of false alarms may remain
  - Difficulty of feasibility checking for embedded software

Our approach: OS-in-the-loop CEGAR

OS-in-the-Loop CEGAR (OiL CEGAR)



Model Refinement

$M \leftarrow M \odot A(\tau')$

Model Checking

$M \vDash \varphi$

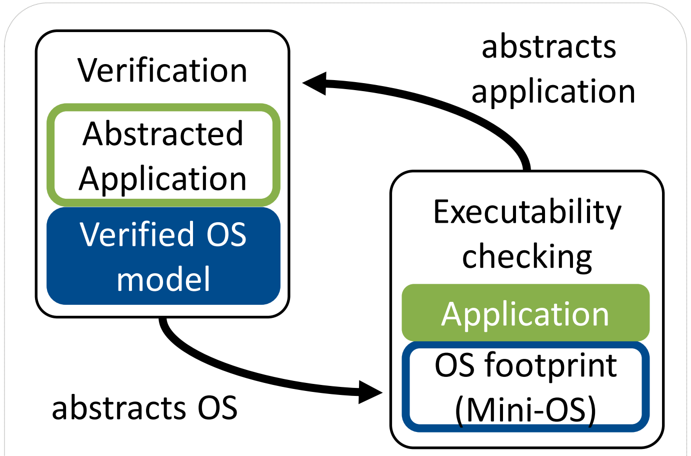
NuSMV

Executability Checking

$\widetilde{M_{os}(\tau)} \,||\, M_{app} \vDash^{n}_{R}$

CBMC

OS-in-the-loop CEGAR



Verification

Abstracted Application

Verified OS model

abstracts application

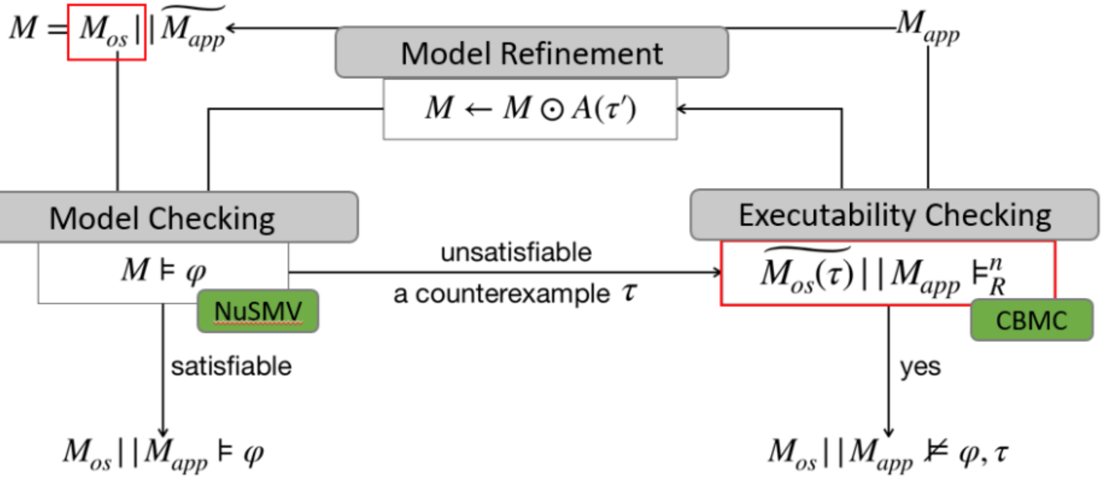
Executability checking

Application

OS footprint (Mini-OS)

abstracts OS

Reciprocal abstraction scheme



$M = M_{os} || \widetilde{M_{app}}$

Model Refinement

$M \leftarrow M \odot A(\tau')$

M_{app}

Model Checking

$M \vDash \varphi$

NuSMV

unsatisfiable

a counterexample τ

Executability Checking

$M_{os}(\tau) || M_{app} \vDash_R^n$

CBMC

satisfiable

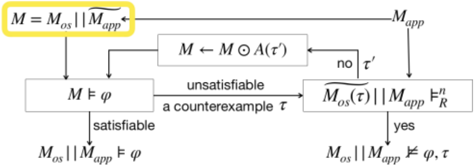
yes

$M_{os} || M_{app} \vDash \varphi$

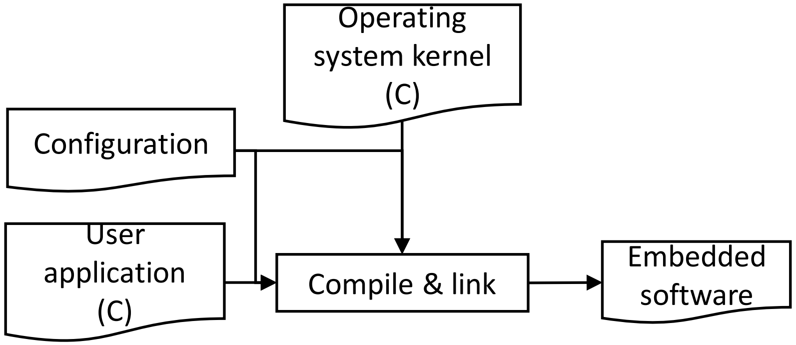
$M_{os} || M_{app} \nvDash \varphi, \tau$

- **A verified OS model** is used to enhance the accuracy of the property checking
- **A mini-OS** is constructed from the counterexample trace for executability checking with improved accuracy
- Model refinements are performed through **trace composition**
- Utilized two different model checkers, NuSMV and CBMC, suitable for the two different purposes

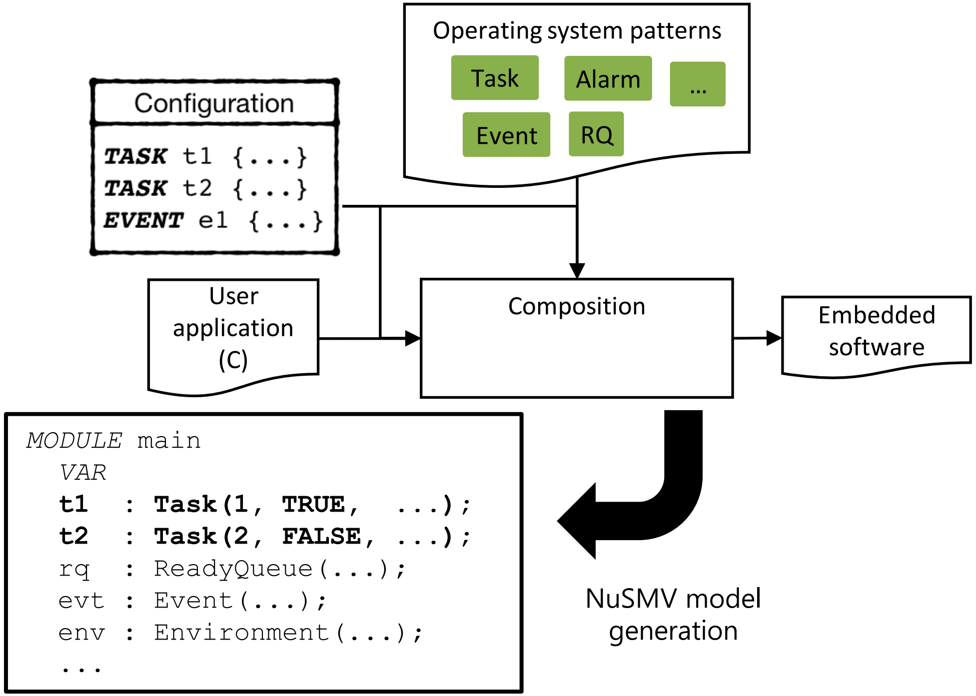
Formal OS models used in this work



- A pattern-based OS model generation framework* is reused



Typical construction of embedded software



```
TASK t1 {...}
TASK t2 {...}
EVENT e1 {...}
```

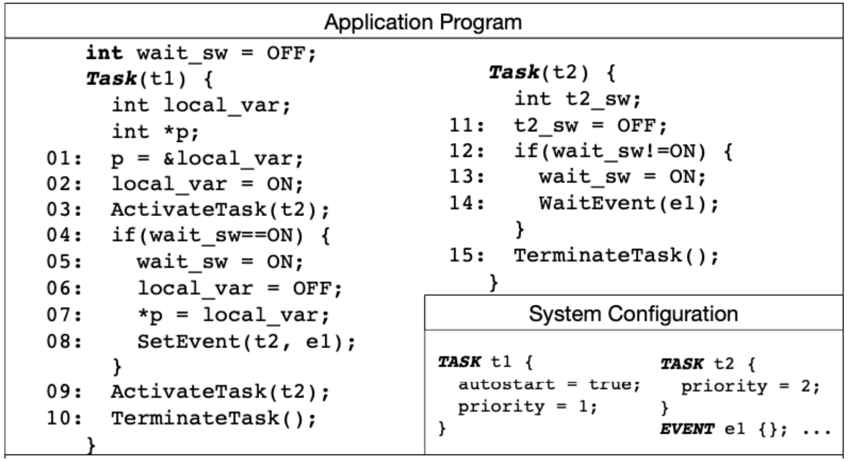
```
MODULE main
  VAR
  t1  : Task(1, TRUE,  ...);
  t2  : Task(2, FALSE, ...);
  rq  : ReadyQueue(...);
  evt : Event(...);
  env : Environment(...);
  ...
```

---

* Y. Choi, “A configurable V&V framework using formal behavioral patterns for OSEK/VDX operating systems,” Journal of Systems and Software, 2018.

Application model construction

1. CFG construction for each task
2. Control abstraction
    - **Blocks together a sequence of statements** to be executed without interrupts
3. Data abstraction
    - **Abstracts visible statement with a unique symbol**
    - Major sources of false alarms
    - But greatly helps to reduce the complexity
4. Conversion into a task statemachine
    - Each transition is guarded to check the scheduling status of the task
5. Parallel composition of task statemachines



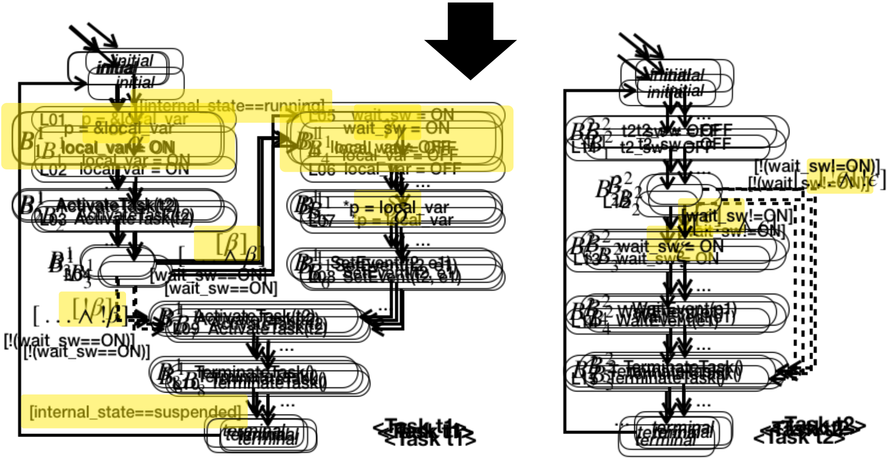
Application Program

```
    int wait_sw = OFF;
    Task(t1) {
      int local_var;
      int *p;
01:   p = &local_var;
02:   local_var = ON;
03:   ActivateTask(t2);
04:   if(wait_sw==ON) {
05:     wait_sw = ON;
06:     local_var = OFF;
07:     *p = local_var;
08:     SetEvent(t2, e1);
      }
09:   ActivateTask(t2);
10:   TerminateTask();
    }
```

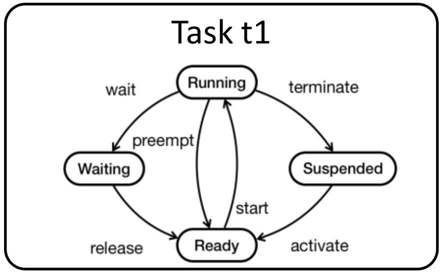
```
    Task(t2) {
      int t2_sw;
11:   t2_sw = OFF;
12:   if(wait_sw!=ON) {
13:     wait_sw = ON;
14:     WaitEvent(e1);
      }
15:   TerminateTask();
    }
```

System Configuration

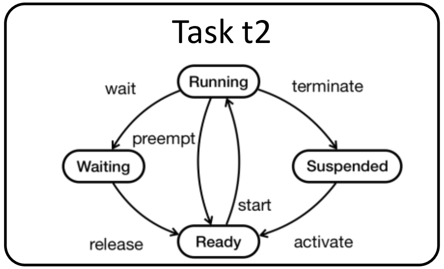
```
TASK t1 {                  TASK t2 {
  autostart = true;          priority = 2;
  priority = 1;            }
}                          EVENT e1 {}; ...
```



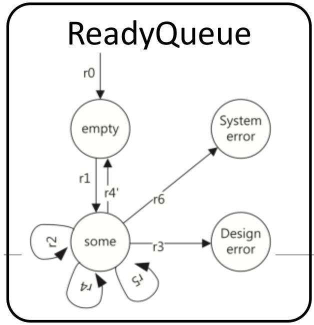
Composition model



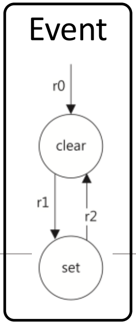
Task t1



Task t2

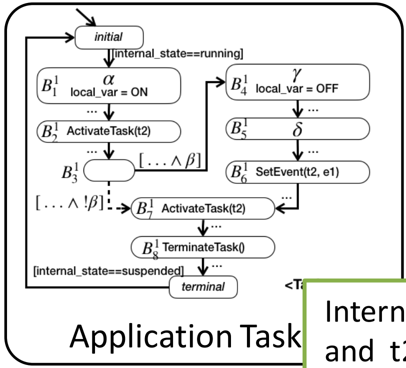


ReadyQueue

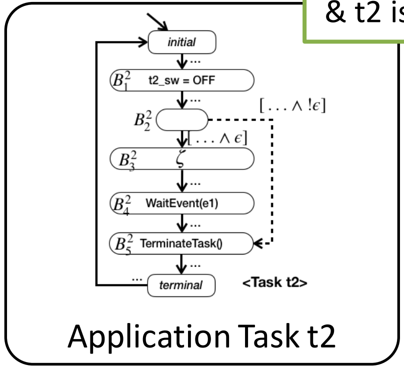


Event

...



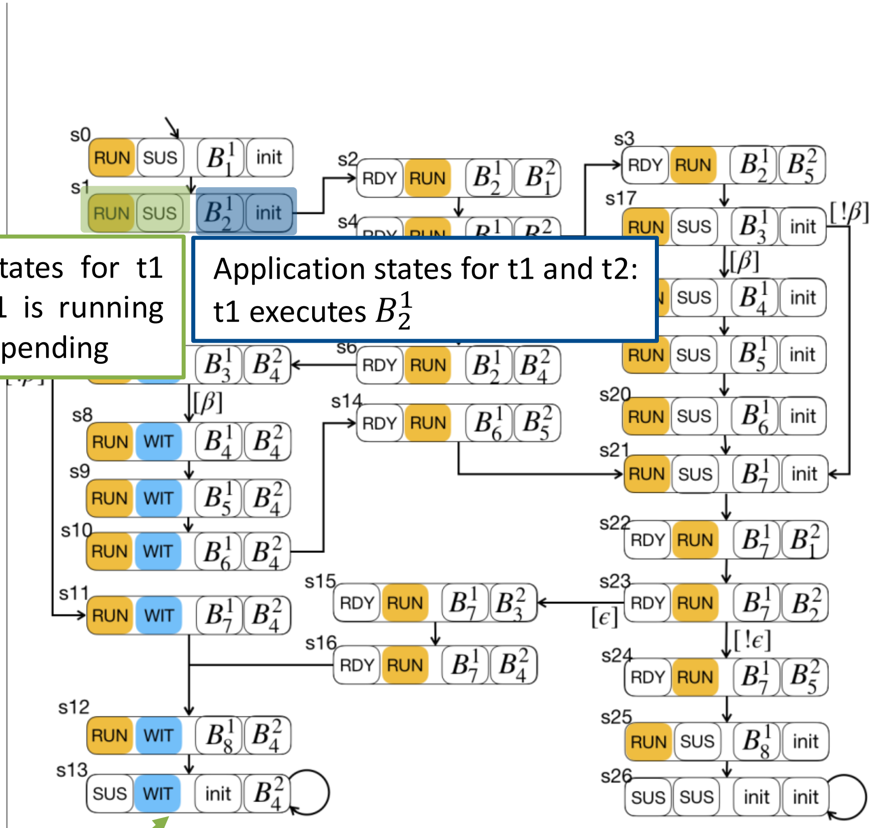
Application Task



Application Task t2

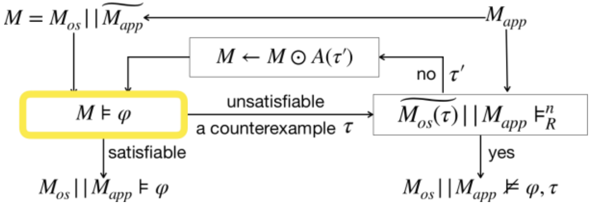
Internal states for t1 and t2: t1 is running & t2 is suspending

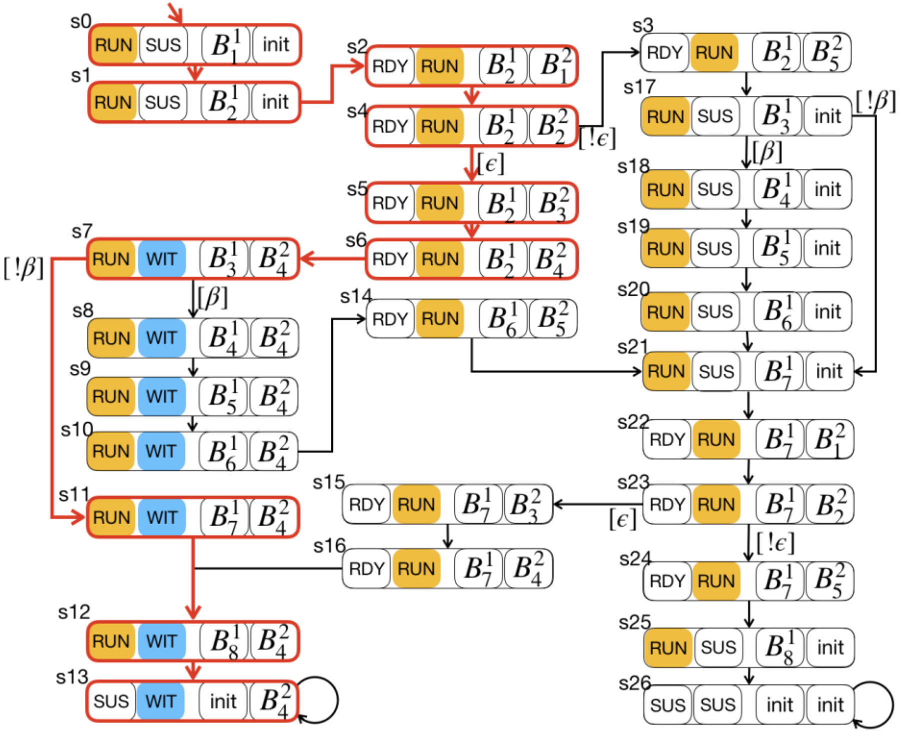
Application states for t1 and t2: t1 executes B_2^1



Infinitely waiting state

Model checking using NuSMV

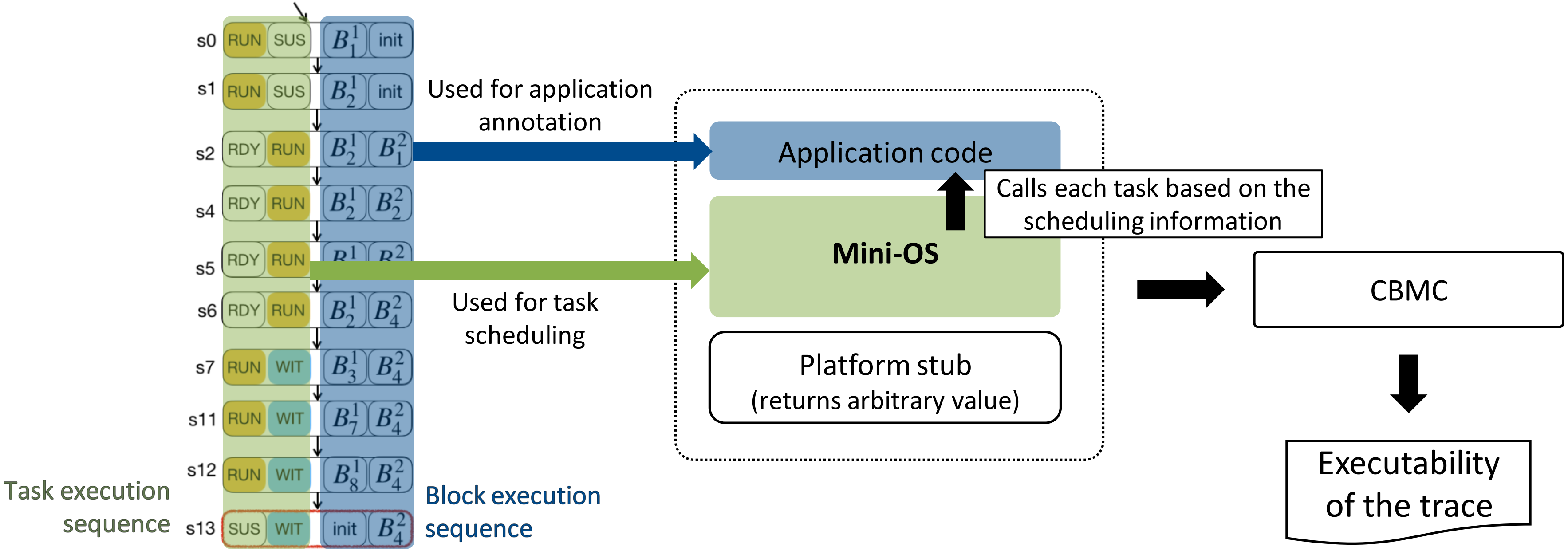




- Boolean properties and temporal logic properties in CTL and LTL can be checked
- Automatically generates a counterexample trace
- We can supply another module for monitoring automata with Boolean or temporal properties.
  - We verify assertion and API-call constraint checking as special types of monitoring automata

No infinitely waiting state

Static executability checking



Task execution sequence

Block execution sequence

Used for application annotation

Used for task scheduling

Application code

Calls each task based on the scheduling information

Mini-OS

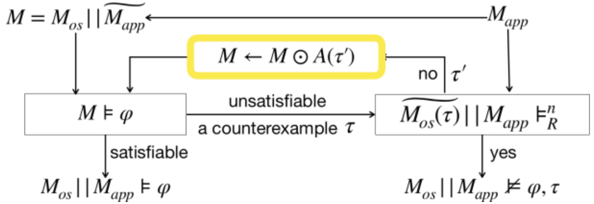
Platform stub
(returns arbitrary value)

CBMC

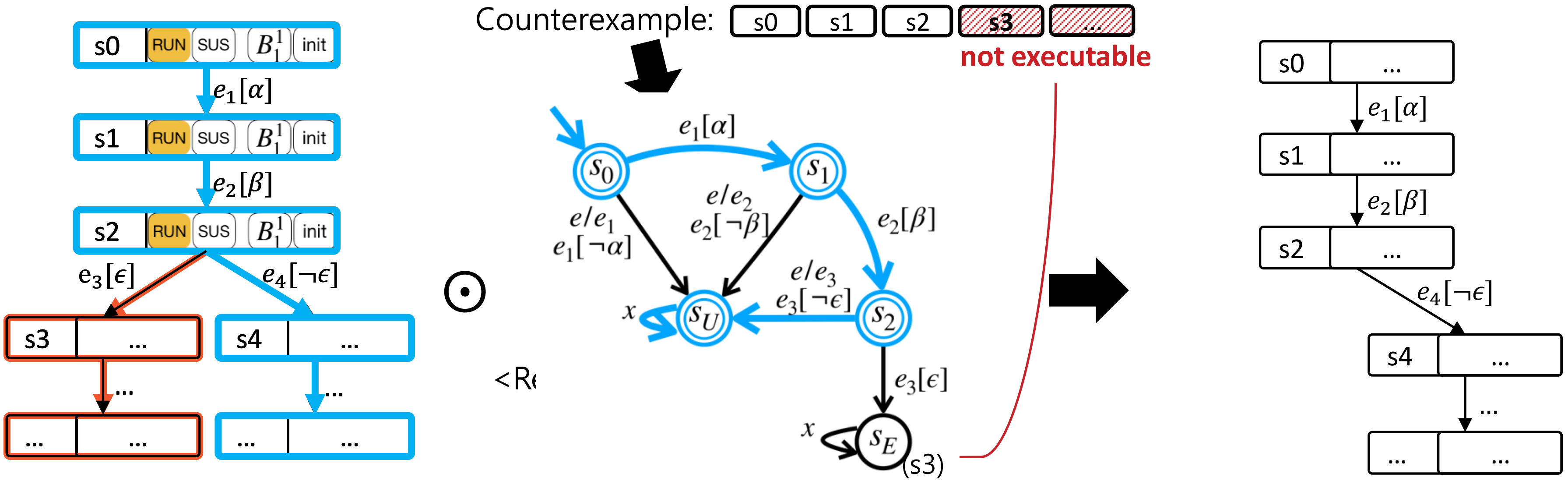
Executability of the trace

- Executability of a counterexample is confirmed by checking the reachability of each application block

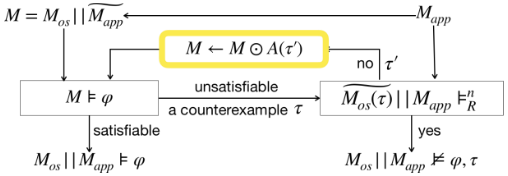
Model refinements



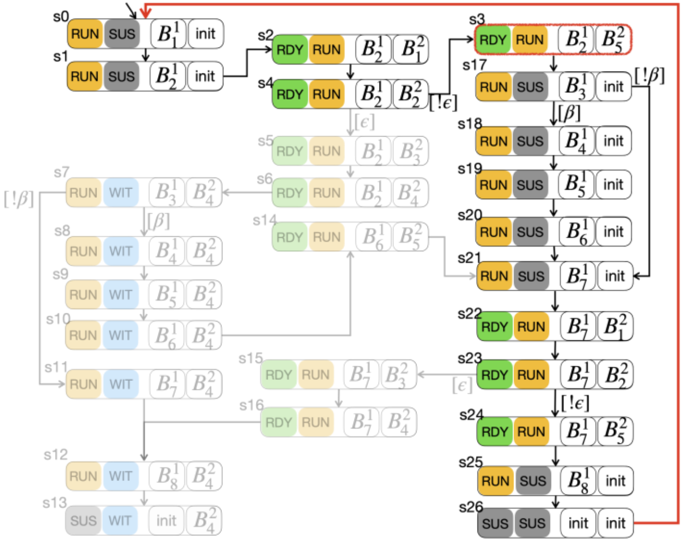
- The model is refined through trace composition with refinement base
- **A refinement base** is a statemachine constructed from a subtrace of the counterexample up to the non-executable statement block.
- **Trace composition** of two statemachines A and B, retains a trace in A only if an equivalent trace without leading to an error state exists in B



False alarm reduction for cycles

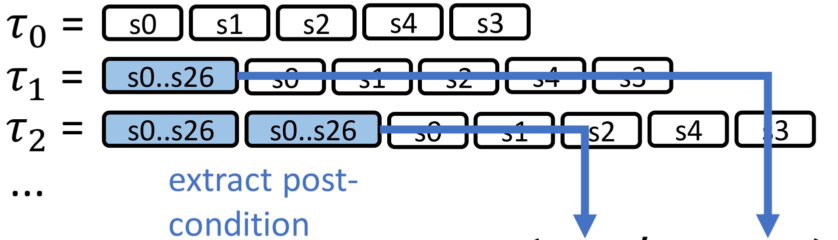


- **Cycles** in the composite model make **infinite traces** reaching error state
- As refinement base remove one trace at a time, these traces can be refined infinitely.
- These traces can be removed if post-condition of the new trace already tested



<Composition model with cycle>

Counterexamples with cycles:

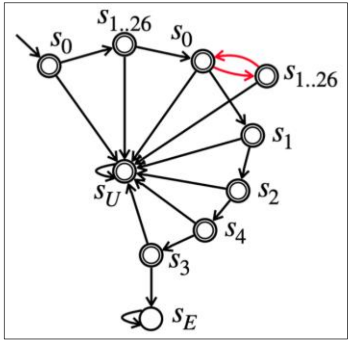


extract post-condition

$$\neg(post' \rightarrow post)$$

SMT Solver

Unsatisfiable



<Refinement base for cyclic traces>

Experiments

Experiments 1 & 2

Objective

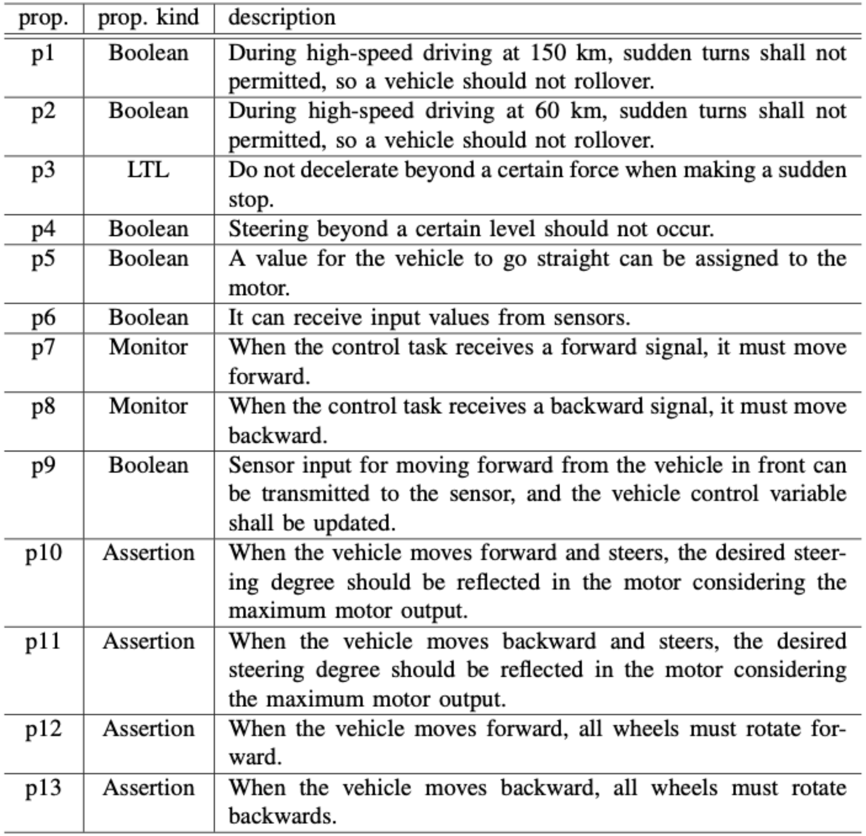
- Experiment 1-1: To evaluate **effectiveness** of property checking
- Experiment 1-2: To evaluate **effectiveness** of API-call constraint checking
- Experiment 2: To **compare the verification accuracy** of OiL-CEGAR

Applications

- TS1. Two example programs running on Erika OS
  (small scale / 3 tasks / tens of LoC)
- TS2. An object-follower and a platoon running on **Lego Mindstorms NXT**
  (realistic / 3~4 tasks / hundreds of LoC)
- TS3. Application programs running on **Lego Mindstorms NXT**
  (small scale / 2~3 tasks / ~87 LoC)
- TS4. Test programs from a **commercial conformance test suite**
  (complex / comes from domain experts / 5+tasks / hundreds of LoC)

Effectiveness of OiL-CEGAR : Property checking (2)

- TS2. An object-follower and a platoon running on **Lego Mindstorms NXT** (realistic / 3~4 tasks / hundreds of LoC)
- Assuming these are real vehicle, we verifies properties that should be satisfied on real vehicles.
- There are 13 properties for
  - vehicle rollovers,
  - sharp turns,
  - sudden stops,
  - or liveness properties.
- Compared with testing method
  which is good at identifying presence of bugs



prop.	prop. kind	description
p1	Boolean	During high-speed driving at 150 km, sudden turns shall not permitted, so a vehicle should not rollover.
p2	Boolean	During high-speed driving at 60 km, sudden turns shall not permitted, so a vehicle should not rollover.
p3	LTL	Do not decelerate beyond a certain force when making a sudden stop.
p4	Boolean	Steering beyond a certain level should not occur.
p5	Boolean	A value for the vehicle to go straight can be assigned to the motor.
p6	Boolean	It can receive input values from sensors.
p7	Monitor	When the control task receives a forward signal, it must move forward.
p8	Monitor	When the control task receives a backward signal, it must move backward.
p9	Boolean	Sensor input for moving forward from the vehicle in front can be transmitted to the sensor, and the vehicle control variable shall be updated.
p10	Assertion	When the vehicle moves forward and steers, the desired steering degree should be reflected in the motor considering the maximum motor output.
p11	Assertion	When the vehicle moves backward and steers, the desired steering degree should be reflected in the motor considering the maximum motor output.
p12	Assertion	When the vehicle moves forward, all wheels must rotate forward.
p13	Assertion	When the vehicle moves backward, all wheels must rotate backwards.

Effectiveness of OiL-CEGAR : Property checking (2)

OiL-CEGAR | Testing

App.	prop.	Expected result	OiL-CEGAR verification				OiL-CEGAR exec. chk.		Total time(s)	result	CROWN (option: -hybrid)					
			Time(s)	Mem(MB)	len(trace)	#R	Time(s)	Mem(MB)			Time_first(s)	Mem_first(s)	Mem_total(s)	found	TA ratio	result
obj_follower	p1	satisfied	5,428	207	106	558	10,976	1201	18,000	timeout	-	-	1,119	0	-	timeout
	p2	violated	6	39	36	4	15	268	37	violated	46	41	1,646	604	10/10	violated
	p3	violated	18	47	67	12	69	631	127	violated	60	39	1,375	79	10/10	violated
	p4	violat							1	violated	18	36	162	201	10/10	violated
	p5	violat							5	violated	926	109	134	44	10/10	violated
	p6	violat							6	violated	1	16	1,777	626	10/10	violated
	p7	violat							2	violated	51	47	147	10	10/10	violated
	p8	violat							2	violated	917	94	142	2	2/2	violated
	p9	violat							6	violated	641	76	113	20	10/10	violated
	p10	violat							8	violated	34	48	164	46	10/10	violated
	p11	violat							5	violated	654	63	122	2	2/2	violated
	p12	violat							5	violated	-	-	146	0	-	not found
	p13	violat							8	violated	-	-	146	0	-	not found
osek_platoon	p1	violat							5	violated	110	22	90	157	10/10	violated
	p2	violat							5	violated	8	16	88	714	10/10	violated
	p3	violat							6	violated	-	-	90	0	-	not found
	p4	violat							5	violated	7	16	100	868	10/10	violated
	p5	violated							7	violated	13	17	101	957	10/10	violated
	p6	violated	1	26	4	0	1	31	7	violated	1	16	54	790	10/10	violated
	p7	violated	604	90	66	64	348	312	1130	violated	25	17	57	48	10/10	violated
	p8	satisfied	13,875	261	106	588	2414	337	18,000	timeout	-	-	57	0	-	timeout
	p9	violated	1	29	6	0	1	42	1	violated	22	16	54	58	10/10	violated
	p10	violated	31	46	32	11	15	114	78	violated	1	16	115	1,329	10/10	violated
	p11	satisfied	11,739	255	116	808	4087	342	18,000	timeout	-	-	124	0	-	timeout
	p12	satisfied	11,291	215	116	909	4250	319	18,000	timeout	-	-	124	0	-	timeout
	p13	satisfied	11,435	214	67	904	4137	298	18,000	timeout	-	-	124	0	-	timeout

OiL-CEGAR
- Founds all the property violations while testing cannot find 3 violations (testing finds 86% of prop. violation)
- But two method cannot verify the 5 properties (expected to be satisfied)
- OiL-CEGAR sometimes finds property violations faster than testing method

Test environment:
- Ubuntu Linux-based machine with a 3.3Ghz Intel Xeon Gold 6234 CPU and 192 GB of memory.
- NuSMV version 2.6.0 with dynamic variable reordering & cone-of-influence reduction.
- CBMC version 5.10 for executability checking

Conclusion

- This thesis proposed a model checking technique for the verification of multitasking embedded applications.
  - Model checking is applied to the **formal OS model** and the abstraction model.
  - Executability of a trace was checked on the application code with a mini-OS.
  - In model refinements, a refinement base is introduced to remove false alarm traces.

- TORCHE is **accurate** and **efficient** for verifying multitasking embedded applications.
  - TORCHE is accurate as it includes not only **formal operating system** but also, **application program code**.
  - TORCHE is also efficient as it **abstracts applications** and **operating systems** in model checking and executability checking, one at a time.

Future work

Improving performance

- Reuse of NuSMV checking data after refinement
- Configuration slicing
- Removing missed true alarm
- Improving executability checking

Reducing false alarms

- Relative timing
- Hardware abstraction
- Use of non-deterministic interrupts
- Use of finite counterexample trace

Infinite refinements

- Verification over fixed-size memory
- Verification of infinite spaced program

Support for more platforms

- Support other OS and platforms
- Support general OS such as Linux or Windows…

Thank you!!