Taming Interrupts for Verifying Industrial Multifunction Vehicle Bus Controllers

Han Liu[1,2,3(✉)], Yu Jiang[1,2,3], Huafeng Zhang[1,2,3], Ming Gu[1,2,3,4], and Jiaguang Sun[1,2,3]

[1] Key Laboratory for Information System Security, Ministry of Education, Beijing, China
[2] Tsinghua National Laboratory for Information Science and Technology, Beijing, China
[3] School of Software, Tsinghua University, Beijing, China
liuhan0518@gmail.com
[4] China Railway Rolling Stock Corporation (CRRC), Beijing, China

Abstract. Multifunction Vehicle Bus controllers (MVBC) are safety-critical sub-systems in the industrial train communication network. As an interrupt-driven system, MVBC is practically hard to verify. The reasons are twofold. First, MVBC introduces the concurrency semantics of deferred interrupt handlers and communication via hardware registers, making existing formalism infeasible. Second, verifying MVBC requires considering the environmental features (*i.e.*, interrupt ordering), which is hard to model and reason. To overcome these limitations, we proposed a novel framework for formal verification on MVBC. First, we formalized the concurrency semantics of MVBC and described a *sequentialization* technique so that well-designed sequential analyses can be performed. Moreover, we introduced the *happen-before interrupt graph* to model interrupt dependency and further eliminate false alarms. The framework scaled well on an industrial MVBC product from `CRRC Inc.` and found 3 severe software bugs, which were all confirmed by engineers.

1 Introduction

Multifunction Vehicle Bus controllers (MVBC) are an essential sub-system in the industrial train communication network (TCN). Unfortunately, as an interrupt-driven system with software and hardware, MVBC is highly error-prone. Even worse, employing formal verification on MVBC is practically challenging. The reasons are twofold. First, MVBC incurs concurrency from random arrival of interrupts, asynchronous handlers and software-hardware communication via registers. Such concurrency is little clearly investigated and can fail existing analyses. Second, MVBC is reactive to environmental inputs (*i.e.*, interrupts), but their dependency, *i.e.*, in what order interrupts occur, is hard to reason.

Our Solution. We proposed a novel framework to verify MVBC in practice. We first formalized its concurrency semantics and described a *sequentialization* technique, considering asynchronous deferral and hardware register communication.

J. Fitzgerald et al. (Eds.): FM 2016, LNCS 9995, pp. 764–771, 2016.
DOI: 10.1007/978-3-319-48989-6_48

The sequentialized programs can be then verified using existing sequential verifiers. Second, we introduced the *happen-before interrupt graph* to model interrupt dependency and further prune false alarms.

Contribution. Main contributions are summarized below.

- We formalized the interrupt-driven concurrency model of MVBC-like systems.
- We proposed a sequentialization based framework to practically verify MVBC.
- We have applied the framework on a real-world industrial MVBC product and found 3 severe previously unknown bugs, which were all confirmed.

2 Multifunction Vehicle Bus Controller

MVBC is used to control the communication between the train bus and devices [5,6]. As an interrupt-driven concurrent system, MVBC consists of both software and hardware. While the classical concurrency semantics are widely discussed [3, 7,8], MVBC-like systems are relatively little studied. We first introduce two highly-relevant concurrency features.

Asynchronous Deferral. To service an interrupt request, an Interrupt Service Routine (ISR) will be invoked. ISR is prioritized and preemptive. It can asynchronously post a *deferral* into a global FIFO queue for delayed execution. Deferrals cannot preempt each other but can be preempted by other ISRs.

Hardware Registers. The communication between software and hardware of MVBC is realized via shared hardware registers. The code below defines 2 macros for register writing and reading. Particularly, strict memory consistency may be violated in this type of communication, *e.g.*, a `HAL_IO_INPUT` after `HAL_IO_OUTPUT` cannot guarantee to access the same value.

```
/* Write to hardware registers */
HAL_IO_OUTPUT(IO_RESET | __content);
```

```
/* Load from hardware registers */
HAL_IO_INPUT(__content);
```

Table 1. $\mathbb{T}$: types. $\mathbb{N}$: priority. Var: variables. i: input. r: return value. e: expression. b: basic statement. c: predicate. λ: empty rule. a: address for a register.

$$
\begin{array}{l}
\mathit{Prog} \coloneqq \mathit{Var}\ (\mathit{ISR}, \mathit{DF})^* \quad \mathit{ISR} \coloneqq \texttt{task}\ p\ (i : \mathbb{T}, r : \mathbb{T}, n : \mathbb{N})\ (\mathit{Var}\ \mathit{stmt}^*) \\
\mathit{DF} \coloneqq \texttt{deferral}\ d\ (i : \mathbb{T}, r : \mathbb{T})\ (\mathit{Var}\ \mathit{stmt}^*) \\
\mathit{stmt} \coloneqq \lambda \mid b \mid \mathit{stmt}; \mathit{stmt} \mid \texttt{while}\ c\ \texttt{do}\ \mathit{stmt} \mid \texttt{if}\ c\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt} \\
\qquad \texttt{return}\ e \mid \texttt{post}\ d \mid \texttt{mask}\ p^* \mid \texttt{unmask}\ p^* \mid \texttt{write}\ a \mid \texttt{read}\ a
\end{array}
$$

Formulation. First, we present an abstract language for MVBC as in Table 1. We consider a collection of ISRs and deferral with shared variables. Supported operations include basic control flow and **`post`** a deferral, **`(un)mask`** certain ISR, **`write`** to and **`read`** from hardware registers. Then, we formalize the

concurrency semantics as transitions on configurations. Each configuration is $\langle P, M, R, S, Q\rangle$ where $P = ISR \cup DF$. $M = P \mapsto \{Idle, Run, Pend\}$ denotes the handler state. $R = ISR \mapsto \{\texttt{true}, \texttt{false}\} \times \{\texttt{true}, \texttt{false}\}$ identifies the arrival and masking of interrupts. S is a stack for ISR with operations `push`, `pop` and get the `top`. Q is a queue for deferral with operations `enqueue`, `dequeue` and get the `head`.

The formal concurrency semantics is shown in Table 2. The DISPATCH rule models a preemption behavior of a ISR. MASK rule disables specific ISR. Without arrival of interrupts, we EXECUTE the `top` of the stack, or `head` of the queue. RETURN semantics differs in ISR and deferral. The former leads to a `pop` while the latter causes a `dequeue`. The POST of a deferral amounts to an `enqueue`.

Table 2. Semantics of MVBC. $i, m, n \in ISR$ $df \in DF$ $p \in P$. $\top$ is a wildcard.

$$\textsc{Dispatch}\ \frac{R_m = (\texttt{true}, \texttt{false}) \wedge M_n = Run \wedge priority : m > n}{i_m : Idle \mapsto Run \wedge p_n : Run \mapsto Pend \wedge S\,[\texttt{push}\ i_m] \wedge R_m = (\texttt{false}, \texttt{false})}$$

$$\textsc{Mask}\ \frac{\texttt{mask}\ \pi \wedge \pi \subseteq ISR}{\forall m \in \pi,\ R_m : (\top, \texttt{false}) \mapsto (\top, \texttt{true})} \qquad \textsc{Execute}\ \frac{p = \texttt{top} \vee (p = \texttt{head} \wedge \texttt{top} = \phi)}{p : \top \mapsto Run}$$

$$\textsc{Post}\ \frac{\texttt{post}\ df}{Q\,[\texttt{enqueue}\ df]} \qquad \textsc{Return-isr}\ \frac{i\texttt{:}\ \texttt{return}\ e}{i : Run \mapsto Idle \wedge S\,[\texttt{pop}]} \qquad \textsc{Return-df}\ \frac{df\texttt{:}\ \texttt{return}\ e}{df : Run \mapsto Idle \wedge Q[\texttt{dequeue}]}$$

3 Approach

In this section, we describe a general framework for verifying MVBC. The workflow of the framework is shown in Fig. 1. Given the MVBC programs, we first employ a sequentialization via inserting *schedule functions* (Sect. 3.1). Then based on the `IEC 61375` and specifications, we model the interrupt dependency using the *happen-before interrupt graph* (Sect. 3.2). The graph is leveraged to reduce the sequential programs. Next, we use an existing verifier (*e.g.*, CBMC [2]) to verify the reduced sequential programs on safety-critical properties of MVBC.

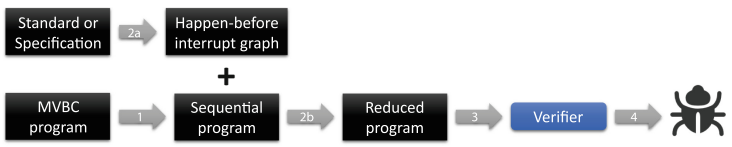


Fig. 1. The general framework for MVBC verification.

3.1 Sequentialization

The sequentialization of MVBC is realized via inserting the *schedule functions*, which can simulate the concurrency through non-deterministic function calls. For ISR, we adopt the similar sequentialization as [14] in left of Fig. 2. For deferral, we propose the `schedule_df` in right of Fig. 2 to run the FIFO queue. Furthermore, we use the `schedule_reg` below to sequentialize the communication via hardware registers. It non-deterministically modifies the register values.

```
void schedule_isr():
  int p = Prio;
  for(int i=1;i<=N;i++):
    if(prio[i]>=Prio && nondet()):
      Prio = prio[i]; ISR[i].entry();
  Prio = p;
```

```
/* Request the execution for deferral.
0 means fetching the head to run. */
void scheduler_df(int id):
  if(id != Queue->head.ID && id != 0)
    return;
  else Queue->head.entry();
```

Fig. 2. Schedule an ISR (**Left**) and a deferral (**Right**)

```
void scheduler_reg(int addr) { if(nondet()) update_reg(addr); }
```

```
1  /* A global variable x.
2  Interrupt 2 has higher
3  priority than 1. The
4  return value DF denotes
5  a post of deferral. */
6  int isr_1():
7    x = 0;
8    HAL_IO_OUTPUT(IO_RESET|x);
9    HAL_IO_INPUT(x);
10   return DONE;
11 int isr_2(){return DF;}
12 void df_2(){x=2;}
```

```
1  int isr_1():
2    x = 0; schedule_isr();
3    HAL_IO_OUTPUT(IO_RESET|x);
4    schedule_reg(ADDR);
5    HAL_IO_INPUT(x);
6    return DONE;
7  int isr_2():
8    enqueue(2); schedule_df(2);
9    return DF;
10 void df_2():
11   x=2; schedule_isr();
12   dequeue(); schedule_df(0);
```

Fig. 3. An example code before (**Left**) and after (**Right**) the sequentialization

Idea of Sequentialization. The insertion of `schedule_isr()` is described in [14]. `schedule_df()` is inserted when deferral is posted. `schedule_reg()` is inserted *after* a write and *before* a read of hardware registers. An example is shown in Fig. 3. In the right, `schedule_isr()` is inserted after an write to global variable in `isr_1()` (line 2) and `df_2()` (line 11). Because `isr_2()` has higher priority than `isr_1()`, no insertions are in `isr_2()`. `schedule_df()` is inserted at line 8 and 12 to run the deferral queue. `schedule_reg()` is inserted at line 4 to capture possible communication between line 8 and 9 in the left.

3.2 Happen-Before Interrupt Graph

Verification on MVBC requires reasoning the interrupt dependency. However, such dependency lacks formulation. For example, `IEC 61375` specifies that:

"After processing a main frame, a slave device will send a slave frame."

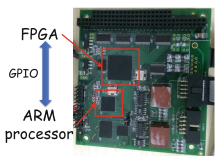
We can conclude that no preemption occurs between interrupt handlers of main and slave frames. However, transferring this kind of knowledge into a practical use case is commonly time-consuming and error-prone. To mitigate this complexity, we proposed the *happen-before interrupt graph* (HBIG) to capture domain knowledge and automatically integrate with the verification.

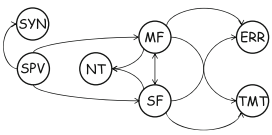
The *happen-before* relation is denoted as $\prec \subseteq ISR \times ISR$. $a \prec b$ implies that `isr_a` is prior to and cannot be preempted by `isr_b`. A HBIG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of interrupts and $\mathcal{E}$ denotes a set of *happen-before* relations. HBIG can be considered as a directed graph. $a \prec b$ indicates a *path* from a to b in the graph. On the contrary, two interrupts are *unordered* if no path connects them. In the verification, a path suggests an infeasible interleaving between two interrupts. Taking the code in left of Fig. 3 as an example, if `isr_1` $\prec$ `isr_2`, the `schedule_isr` at line 2 of right of Fig. 3 can be reduced. To integrate HBIG with the sequentialization, we add the following code before line 4 in `schedule_isr` of Fig. 2 to filter out infeasible interleaving.

```
/* hb(a,b) checks the path existence between a and b. */
if(hb(id, i) || hb(i, id)) continue;
```

4 Evaluation

Target System. We selected TiMVB, an industrial MVBC product from `CRRC Inc.`, as shown in left of Fig. 4. The ARM processor runs C programs on the `eCos`[1] operating system, and communicates with an FPGA via general-purpose input output (GPIO) pins, which are hardware registers as in Sect. 2.





Int	Reg	Deferral	Priority
MF	✓	–	0
SF	✓	–	0
SPV	✓	–	1
SYN	✓	✓	2
NT	–	–	3
ERR	✓	✓	4
TMT	✓	✓	4

Fig. 4. TiMVB; HBIG; **Int**: interrupt, **Reg**: register communications.

TiMVB contains 4923 lines of C code and handles 7 types of interrupts, as in right of Fig. 4 Except NT, other interrupts all communicate with FPGA via

[1] http://ecos.sourceware.org/.

registers. 5 priorities are set (0 is the highest), indicating a large state space. Based on IEC 61375 and discussions with domain experts, we developed an HBIG as in middle of Fig. 4. In specific, SPV *happen-before* all interrupts since it can reach any other nodes. MF and SF *happen-before* other less-prioritized interrupts except SYN, which implies that SYN interrupt can arrive in arbitrary orders.

Verification Results. In the verification, we focused on two kinds of safety-critical properties: (1) Data Validity (DV), Device state data must hold valid values. (2) Frame Consistency (FC), Frame data must match frame types.

The verification results are shown in Fig. 5. We have exposed 1 bug on data validity property (ID=2) and 2 bugs on frame consistency property (ID=6,7). We compared two verification strategies: with and without an HBIG. Based on Fig. 5, HBIG helped improve the time efficiency from 18.84 % to 67.35 %. In particular, it successfully scaled on a complex verification when non-HBIG strategy failed due to memory limitation (ID=4). Moreover, the non-HBIG strategy reported a false positive (ID=3) due to infeasible interleaving. HBIG based strategy, which sequentialized the programs with well-formalized concurrency semantics, generated no false alarms.

ID	Type	Time (Second) NoHB	HB	Improve	Outcome NoHB	HB
1	DV	639.20	488.17	23.63 %	TRUE	TRUE
2	DV	25.42	8.30	67.35 %	FALSE	FALSE
3	DV	37.68	211.75	–	FALSE	TRUE
4	DV	OOM	871.34	$+\infty$	–	TRUE
5	DV	727.18	590.16	18.84 %	TRUE	TRUE
6	FC	23.41	10.06	57.03 %	FALSE	FALSE
7	FC	29.67	15.88	46.48 %	FALSE	FALSE

Fig. 5. Verification results. NoHB: without HBIG. HB: with HBIG. OOM: Out of memory. Framed cell: False alarm.

```
1  void syncprocess_handle():
2    if (sync_checkbit == HAL_IO_ENUM_SYNC_STATUS):
3      mvb_arm_receive_sync(mvb_device_status_16);
4      mvb_device_status = *((MVB_DEVICE_STATUS*) & mvb_device_status_16);
```

```
1  #define mvb_arm_send_main_frame(__content)
2      ba_mf = __content; HAL_IO_OUTPUT(IO_RESET | __content); \
3      HAL_IO_FRAME_WRITE_SIGNAL_PULSE; HAL_IO_SEND_MAIN_FRAME;
```

Two uncovered bugs are shown above (upper ID=2, lower ID=7). The upper bug occurs when the ISR of SYN interrupt is preempted by MF or SF between line 3 and 4. In that case, a *write-write-read* data race is triggered to taint the global variable mvb_device_status_16. As for the lower bug, content of a main frame is set (line 2) and then the frame type is set (line 3). A frame inconsistency manifests when the sending operation of the hardware is performed in between, causing a slave frame sent with the main frame content.

5 Lessons Learned

i. Software correctness is not system reliability. Verification of embedded software should reason their interactions with hardware, including the interaction semantics and how it is implemented. In our case, without considering hardware registers in TiMVB, the frame consistency bugs cannot be uncovered.

ii. "Interfaces" of formal methods are desired. One way to facilitate the practical application of formal methods is a convenient interface to the practice. In our case study, engineers used to hard-code the interrupt dependency, which is imprecise and error-prone. From this point, the HBIG is practically useful by encapsulating the implementation and offering high-level abstractions.

6 Related Work

Program Sequentialization. The original idea of sequentialization was to transform a program within bounded context switches [8,10]. Later attempts have considered balancing efficiency and accuracy [4,13], and handling asynchrony [3,7,14]. We extended the sequentialization by modeling deferral into a FIFO queue and leveraging HBIG for further reduction.

Analyses on Interrupts. Interrupt-driven software has been widely discussed [1,9,14]. Schwarz *et al.* provided static analyses on prioritized tasks under dynamical scheduling [12]. Schlich *et al.* proposed to reduce non-nested interrupts [11]. Our insight is to formalize and analyze more real-world interrupt semantics, including deferral and communication via hardware registers, which have long been ignored.

7 Conclusion

In this paper, we introduced a verification framework for MVBC systems. The framework is based on the formal semantics of MVBC to sequentialize interrupt handlers and model their dependency. On an industrial product TiMVB, the framework helped find two types of previously unknown defects, which were confirmed by engineers. Our future plan includes verifying extensive real-world systems and building industry-friendly tools.

Acknowledgement. This research is sponsored by NSFC Program (No.91218 302, No.61527812), National Science and Technology Major Project (N0.16ZX010 38101), MIIT IT funds (Research and application of TCN key technologies) of China, and National Key Technology R&D Program (No.2015BAG14B01-02).

References

1. Brylow, D., Damgaard, N., Palsberg, J.: Static checking of interrupt-driven software. In: ICSE 2001, pp. 47–56 (2001)
2. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
3. Emmi, M., Lal, A., Qadeer, S.: Asynchronous programs with prioritized task-buffers. In: FSE 2012, pp. 48:1–48:11. ACM, New York (2012)
4. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded model checking of multi-threaded C programs via lazy sequentialization. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 585–602. Springer, Heidelberg (2014)
5. Jiang, Y., et al.: Design and optimization of multiclocked embedded systems using formal techniques. TIE **62**, 1270–1278 (2015)
6. Jiang, Y., et al.: Design of mixed synchronous/asynchronous systems with multiple clocks. TPDS **26**, 2220–2232 (2015)
7. Kidd, N., Jagannathan, S., Vitek, J.: One stack to run them all: reducing concurrent analysis to sequential analysis under priority scheduling. In: Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 245–261. Springer, Heidelberg (2010)
8. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. FMSD **35**(1), 73–97 (2009)
9. Liu, H., et al.: idola: bridge modeling to verification and implementation of interrupt-driven systems. In: TASE, pp. 193–200 (2014)
10. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: PLDI 2004, pp. 14–24 (2004)
11. Schlich, B., Noll, T., Brauer, J., Brutschy, L.: Reduction of interrupt handler executions for model checking embedded software. In: Namjoshi, K., Zeller, A., Ziv, A. (eds.) HVC 2009. LNCS, vol. 6405, pp. 5–20. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19237-1_5
12. Schwarz, D., et al.: Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In: POPL, pp. 93–104 (2011)
13. Tomasco, E., Inverso, O., Fischer, B., Torre, S., Parlato, G.: Verifying concurrent programs by memory unwinding. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 551–565. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46681-0_52
14. Wu, X., Chen, L., Mine, A., Dong, W., Wang, J.: Numerical static analysis of interrupt-driven programs via sequentialization. In: EMSOFT 2015, pp. 55–64 (2015)