iDola: Bridge Modeling to Verification and Implementation of Interrupt-driven Systems

Han Liu*, Hehua Zhang*, Yu Jiang†, Xiaoyu Song‡, Ming Gu* and Jiaguang Sun*
*School of Software Tsinghua University, TNLIST, KLISS, Beijing, China
†Department of Computer Science and Technology, Tsinghua University, TNLIST, KLISS, Beijing, China
‡Department of ECE, Portland State University, Oregon, USA

***Abstract*—In real-time embedded applications, interrupt-driven systems are widely adopted due to strict timing requirements. However, development of interrupt-driven systems is time-consuming and error-prone. To conveniently ensure a trustworthy system design and implementation is a challenging problem, especially in complex applications. In this paper, we present a novel domain-specific language called iDola to model interrupt-driven systems declaratively and concisely. A major strength of iDola is the feasibility to capture complex interrupt handling mechanism in real-time operating systems and target platforms, such as delayed service and buffered processing. We also propose the formal operational semantics and code generation algorithm of iDola, so that iDola models can be transformed to timed automata for verification and loaded to generate platform-specific codes. We apply iDola on the modeling of an industrial interrupt-driven system, multifunction vehicle bus controller which runs in an embedded environment with eCos operating system. Based on iDola, the system is modeled with a dispatcher which embodies advanced interrupt handling in eCos, including buffered interrupt service routine and deferred service routine. Through transformation, the system design is verified and design bugs are detected. Code generation is also executed using the proposed algorithm. Generated codes display comparatively equal performance in the real system. We believe iDola can facilitate building a trustworthy interrupt-driven system.**

Keywords*—*Interrupt-driven system; Domain-specific language; Interrupt handling

I. Introduction

It is no exaggeration to say that timing is the air of embedded world. Like we human need fresh air, timing constraints are the first priority in many embedded systems, especially safety-critical ones. This common knowledge explains why interrupt-driven mechanism is widely applied in industry. An interrupt-driven system(IDS) processes specific input from the environment as interrupts so that urgent tasks can be serviced as soon as possible. As a result, the difficulty of achieving the expected timing is largely reduced.

However, due to the asynchronous nature of interrupts which may lead to various interleaving executions, IDS are error-prone, especially in complicated applications. Hazard in an IDS comes from three aspects. The first is the unpredictable source, which may generate any interrupt at any time during the execution. Another is the complex interrupt handling mechanism of the real-time operating system(RTOS) and target platform. Unexpected execution paths are likely to be produced to cause failures under such mechanism. The third is the poorly implemented interrupt handler, which is incapable of providing the correct service for interrupts. Thus, it is both challenging and essential to control the three risks. From the practical view, the characteristics of interrupts post requirements on expressiveness and semantics of system modeling language. The expected situation is that IDS can be modeled in a straightforward manner with complex interrupt mechanism captured. Yet, existing solutions are insufficient to achieve these two goals at the same time. Strict formal methods with accurate interrupt semantics are inconvenient to build the entire system. On the other side, expressive modeling approaches, which guarantee rapid system design, are incapable to maintain full mechanism of interrupts. Thus, we present our work to narrow the gap between these two conflicting requirements so that modeling of IDS is better supported.

In the global picture of model-driven engineering(MDE), verification and code-generation are conducted based on models so that trustworthy codes are produced in the target application. However, for embedded IDS, verification is difficult due to the shortage in modeling. Moreover, the state space may be huge due to complex interrupt scenarios. On top of that, code-generation is also strictly required because of the limited computation resource. In another word, for current solutions, it is infeasible to acquire acceptable performance on both verification and code-generation of IDS.

To conquer the limitations above and enhance the engineering of IDS, we propose a novel approach in this paper, which smoothly bridges modeling to both verification and implementation. Contributions of our work are:

1) A declarative domain-specific language(DSL) along with formal semantics, called iDola, to assist rapid and complex design of IDS. With iDola, schedulers, sources and interrupt handlers can be easily specified. Apart from classical interrupt mechanism, iDola is capable of modeling complex handling of RTOS and runtime platform, including delayed process and buffered service routines. Besides, system configuration is well supported for different application platforms.
2) A complete model transformation scheme from iDola to timed-automata(TA). The transformation is defined on pure iDola models without extra adjustment so that verification is feasible based on the same models as in modeling phase.
3) A non-intrusive code-generation method to produce executable application codes. Majority of the generation is automated with interfaces provided for developers to add platform specific information. The

generated codes can be directly used in the target application without any modification.

The rest of this paper is organized as follows: related work is introduced in section II. Section III presents the modeling, verification and code-generation of IDS with iDola in detail. In section IV, case studies on multiple systems with different scales are exhibited to show how iDola works for real-life applications. Section V gives a conclusion of our work.

II. RELATED WORK

Modeling and analysis of IDS are the top priority in the engineering of real-time embedded applications(RTEA). In practice, interrupt handling is dependent on the source of interrupts, scheduling and dispatching mechanism, and the handler program. Thus, a major task to model IDS is to fully capture dynamic features of the three aspects. With the system model, analysis is responsible for uncovering design defects in the handler programs due to incorrect implementations and incomaptibility with the execution platform.

Formal methods and techniques, which can facilitate complete analysis and verificatioin, are leveraged to model interrupt mechanism. Hoare has defined a Communicating Sequential Processes(CSP) representation for interrupt behavior in [8], while algebra of communicating processes(ACP) is used in [12] to capture prioried and non-deterministic nature of interrupts. Bérard introduced Interrupt Timed Automata(ITA) [2], which is suited to the description of multi-task systems with interruptions in a single processor environment. Reachability problem and real-time properites are verified in [3] and [1]. In [11], a formal model of interrupt-driven programs with operational semantics is proposed from a timing perspective, so that analysis over time properities is feasible during development of such programs. The model is small but efficient in depicting crucial timing features of the system. In contrast to iDola, interrupt nesting is not supported in the proposed language and model. Based on [11], the author presents denotational semantics of interrupt-driven programs in [10]. The denotational semantics can be linked with the operational senmantics so that timing analysis can be done to ensure reliability of the system. Furthermore, a formal model as an extenstion of Dijkstra's language of guarded commands is proposed in [24]. The model has probabilistic operational semantics to handle randomicity and nonderminism. In order to model nested interrupts together with an interrupt environment, [17] presents controller automata. Occurrence of interrupts are realized as a sequence of transitions with time. A major advantage of controller automa is the ability to run simulation at model level. In [13], a framework is provided to model and evaluate preemptive scheduling of interrupts in terms of efficiency. Both synchronous and asynchronous events can be captured in high-level model. Moreover, interrupts is modeled as a form of interaction in [23] so that interrupt features including preemption and nesting are constructed as early as possible. Special operations such as enabing and disabling of interrupts, are modeled with a markov state transition diagram in [15] for performance analysis. Compared to iDola, works mentioned above focus on the interaction between interrupt handlers and a simple scheduler. They do not concern complex interrupt handling mechanism.

As for the analysis of IDS, stack overflow, interrupt overload and real-time properties are listed as problems in interrupts [18]. Timing analysis is performed using bounds on the number of context switches [14]. [4] presents a static checker for Z86-based software with hard-time requirements. Interrupt latencies, stack size, as well as fundamental safety and liveness properites are verified by the checker. Furthermore, they presents a tool to analyze deadlines of interrupt handlers [5]. Main contribution of the tool is to significantly reduce efforts of testing without decrease on precision. In [9], an abstract formal model is proposed to represent AUTOSAR OS programs with timing protection for deadline analysis. In addition, as a common problem caused by interrupts, data race is also widely analyzed in [7][22][16][21][6]. [7] proposes to generate interrupts to test access point. [22] exhibits a tool to analyze conflicting shared memory access with a lableing scheme. [16] converts interrupt handlers to multi-thread programs for detection while [6] analyzes the data flow to search races. Besides, Regehr compares the difference between interrupt and thread, then proposes a verification through transformation [19]. [20] presents an abstraction technique to reduce the state space, so that large-scale verification is possible. These methods share the idea that they are applied on source codes while iDola advances analysis to the design phase and is available for complex real-time environment with operating system and target platform.

III. BUILD A SYSTEM WITH IDOLA

For embedded IDS, manual coding is time-consuming and error-prone. On top of that, programs have to be compiled and transplanted to the target platform for execution. To simplify this process and ensure the trustworthiness of the system, common practice is MDE. However, to support rapid system design and implementation in MDE is a challenging issue. To address this problem, we present iDola, a domain-specific modeling language which provides higher abstraction for system constructs, to build an IDS. The development process is shown in Figure 1.

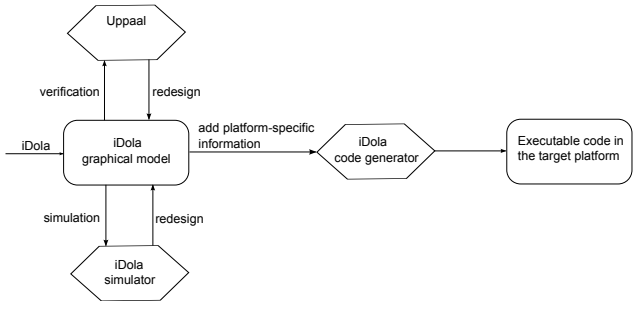


Fig. 1: Development process with iDola

With iDola, a system is modeled graphically. The iDola model can be loaded for simulation and transformed to timed automata so that verification can be done by Uppaal. Once functional correctness is ensured in model-level, developers can manually attach platform specific information to the model and automatically generate executable codes in the target platform. We implement an iDola development tool which includes design module, simulator, model transformer and code generator.

A. iDola Model

M ::= $(Var^*, is^*, Source^+, Dispatcher, ISR^+, DH^*, App^*)$
$Source$ ::= source$\{is^*, pattern^*\}$
$pattern$ ::=not? start is | not? end is | is not? follow is | is not? interrupts is
$Dispatcher$ ::=dispatcher$\{Var^*, Pri, B, Q, Pre, Post, D\}$
Pri ::= priority$\{is = (NUM, NUM)\}^*$
B ::=buffer{capacity = NUM}
Q ::=queue{capacity = NUM}
$Pre, Post$ ::= (pre, post)$\{CFG\}$
D ::= $(is \rightarrow \{name$ [buffer=NUM][, $name$]$\})^+$
ISR ::=isr $name\{Var^*$,mask?, CFG,unmask?—call $name\}$
DH ::=dh $name\{Var^*, CFG$,unmask?$\}$
App ::=app $name\{Var^*, CFG\}$
CFG ::= $Action^*$
$Action$ ::= $stmt|stmt; stmt|$if $cond$ then $Action$ else $Action|$ while $cond$ do $Action|$invoke $name$

Fig. 2: Syntax of iDola

The syntax of iDola is partly listed in Figure 2. An iDola model is consisted of seven categories of constructs: shared data(Var), interrupts defined in the system(is^*), interrupt source($Source$), $Dispatcher$, interrupt service routine(ISR), delayed handler(DH), application routine(App). A $Source$ contains a sequence of interrupt signals(is) and a list of $patterns$ which specify constraints on interrupt signals. Four types of constraints are supported in iDola for depicting the start, end, follow and interruptive relationship of the signal sequence. A $Dispatcher$ is responsible for registering interrupts and dispatching them to corresponding handlers. It includes a name, local variables, priority(Pri) assignments, a buffer(B) to assemble ISRs, a queue($Queue$) to store DHs, initialization and completion actions when registering interrupts(Pre and $Post$), and the dispatching behavior(D). In D, we can attach handlers including buffered ISR and prioritized DH to interrupts. As a major strength of iDola, the capability of modeling a complex $Dispatcher$ makes it feasible to capture extensive interrupt processing features in different platforms. For instance, the configuration of buffer supports the design of ISR assembling, which posts a big DH for multiple ISRs to reduce invoking costs. Furthermore, specification on capacity of queue and priorities of DHs can be used to control the delayed processing. An analogy is the deferred service routine(DSR) and marcro of DSR table size in the real-time operating system eCos. Moreover, ISR, DH and App are similar constructs with local variables and a control flow graph(CFG), except that ISR and DH can execute mask or unmask actions for interrupts. CFG is defined as $Action$ sequence, which is a combination of basic statement($stmt$), sequential statement($stmt; stmt$), branch structure, loop structure and invoking statement.

In order to simulate and analyze the iDola model, we present its operational semantics. The configuration is represented as a tuple $(M, intr, \theta, \Sigma)$, where

- M is an iDola model.
- $intr \in \{is^*\}$ specifies interrupt signals captured by the system.
- $\theta \in \Theta$, where $\Theta = N \cup \{Start, Exit, \perp\}$ represents locations in the control flow graph. N denotes graph nodes. $Start$ and $Exit$ are entrance and exit of the graph respectively. $\perp$ is placeholder if no graph is attached to the current component.
- $\Sigma \triangleq Var^* \rightarrow R$ is a set of states, each of which is defined by the collection of values of all system variables, including a ISR stack and a FIFO DH queue.

For the descriptive convenience, we introduce several auxiliary formulas. **out** is used to specify interrupts generated by $Source$. Calls to **out** get the arrived interrupt. If no interrupts occur, it returns nothing. **cfg** is used to acuire the CFG of an ISR, DH or a given graph location. **isr** and **dh** return corresponding ISRs and DHs for specific interrupts. **loc** reports the current CFG location while **next** steps to a next node in CFG. **pri** gets the priority of an interrupt. **head** is used to get the first DH in $Queue$. **nextIntr** returns a following interrupt to be serviced, which is implemented as a combinational calculation over the **out** function, ISR stack and DH queue. With the model configuration and proposed formulas, operational rules of iDola are listed in Figure 3.

$$\text{(Dispatch)}\quad \frac{\mathbf{out}(Source^*)=intr}{(M,\phi,\perp,\Sigma)\rightarrow(M,intr,\theta',\Sigma)} \quad \text{where } \theta' = \begin{cases} \perp, \mathbf{cfg}(\mathbf{isr}(intr)) = \perp \\ Start, \mathbf{cfg}(\mathbf{isr}(intr)) \neq \perp \end{cases}$$

$$\text{(Mask)}\quad \frac{\mathbf{out}(Source^*)=intr}{(M,intr,\theta,\Sigma)\rightarrow(M,intr,\theta,\Sigma)}$$

$$\text{(Execute)}\quad \frac{\mathbf{out}(Source^*)=\phi}{(M,intr,\theta,\Sigma)\rightarrow(M,intr,\mathbf{next}(\theta),\Sigma')}$$

$$\text{(Preemption)}\quad \frac{\mathbf{out}(Source^*)=intr' \wedge \mathbf{pri}(intr') > \mathbf{pri}(intr)}{(M,intr,\theta,\Sigma)\rightarrow(M,intr',\theta',\Sigma')} \quad \text{where } \theta' = \begin{cases} \perp, \mathbf{cfg}(\mathbf{isr}(intr')) = \perp \\ Start, \mathbf{cfg}(\mathbf{isr}(intr')) \neq \perp \end{cases}$$

$$\text{(Suspension)}\quad \frac{\mathbf{out}(Source^*)=intr' \wedge \mathbf{pri}(intr') \leq \mathbf{pri}(intr)}{(M,intr,\theta,\Sigma)\rightarrow(M,intr,\theta,\Sigma')}$$

$$\text{(Delayed post)}\quad \frac{\theta=Exit \wedge \mathbf{dh}(intr)\neq\phi}{(M,intr,\theta,\Sigma)\rightarrow(M,intr,\theta,\Sigma')}$$

$$\text{(Delayed process)}\quad \frac{\mathbf{out}(Source^*)=\phi \wedge \forall\beta(\mathbf{cfg}(\beta)\in\mathbf{cfg}(\mathbf{isr}(is^*)) \wedge \beta\neq N)}{(M,intr,\theta,\Sigma)\rightarrow(M,\mathbf{head}(),\theta',\Sigma)} \quad \text{where } \theta' = \begin{cases} \perp, \mathbf{cfg}(\mathbf{dh}(\mathbf{head}())) = \perp \\ Start, \mathbf{cfg}(\mathbf{dh}(\mathbf{head}())) \neq \perp \end{cases}$$

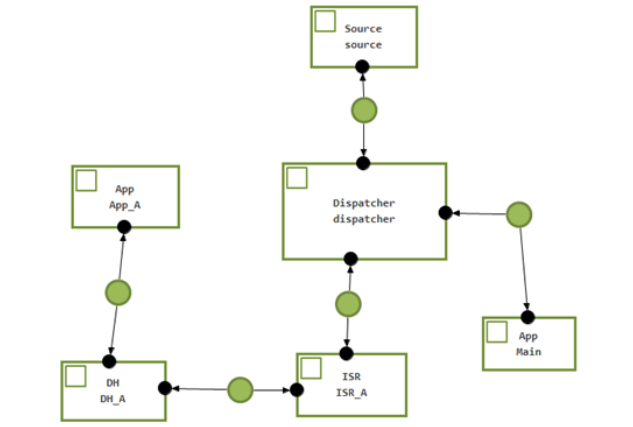
$$\text{(Completion)}\quad \frac{\mathbf{out}(Source^*)=\phi \wedge \theta=Exit}{(M,intr,Exit,\Sigma)\rightarrow(M,\mathbf{nextIntr}(),\theta',\Sigma')} \quad \text{where } \theta' = \mathbf{loc}(\mathbf{cfg}(\mathbf{nextIntr}()))$$

Fig. 3: Operational semantics of iDola

Explanations on the formally-defined operational semantics are as follows:

1) **Dispatch**. Once an interrupt signal is produced and no other signals are present, the $Dispatcher$ starts to service that interrupt and turns to the corresponding ISR. If the ISR is not empty, its $Start$ node is activated. Otherwise, it remains idle.
2) **Mask**. During the processing of an interrupt, same-type interrupts generated by $Source$ are ignored. Location in CFG and system state are not affected.
3) **Execute**. When no interrupts occur, the running component executes a step in its CFG.
4) **Preemption**. When $Source$ generates an interrupt with a higher priority than the current one, $Dispatcher$ saves the context and switches to the high-priority interrupt. Its CFG is then activated.

5) **Suspension**. In contrast to **Preemption**, newly generated interrupt with a lower priority is stored and suspended in $Dispatcher$.
6) **Delayed post**. When an ISR completes and it is associated with a DH, the DH is posted to a queue for delayed handlers.
7) **Delayed process**. When no interrupts occur and executions in all ISRs are completed, $Dispatcher$ begins to launch delayed process from the first handler in $Queue$ and activates the corresponding CFG.
8) **Completion**. An execution is completed if CFG reaches the $Exit$ node. Without new interrupts, $Dispatcher$ starts to service next stored interrupt. The CFG resumes to a historical location if interrupted previously, or it starts from the beginning.




```
int flag=0;
source{
   A;
   start A;
}
dispatcher{
   int count = 0;
   buffer{capacity=3};
   queue{capacity=5};
   task=Main;
   interrupt=A invoke ISR_A{
      count=count+1;
   }
}
app Main{
   /*codes for main task*/
}
```

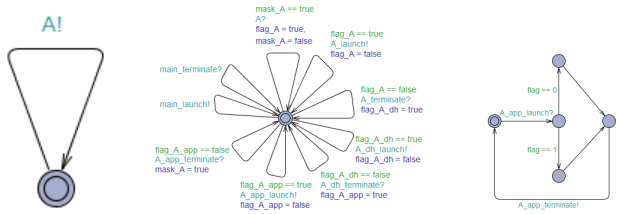
```
isr ISR_A{
   mask A;
   flag=1;
   call DH_A;
}
dh DH_A{
   /*process code*/
   invoke App_A;
   unmask A;
}
app App_A{
   int MSG1=ON_MSG;
   int MSG2=OFF_MSG;

   if flag==1 then
      display MSG1;
   else
      display MSG2;
}
```

Fig. 4: A simple iDola model

A simple iDola model is presented in Figure 4, with graphical representation and textual code. In the example, a main thread $Main$ is running when no interrupts occur. Only one interrupt A is defined. The ISR_A services A by writing a global shared data $flag$. Then ISR_A passes service to a delayed handler DH_A, which does a part of jobs and invokes App_A for further process.

B. Verification support

Due to possible interleaving execution paths, IDS is error-prone, especially in RTEA with complex interrupt handling mechanism. Common faults such as data race, memory leak, may result from inappropriate priorities, unprotected access control, poor flow design of interrupt service and so on. In order to uncover design defects and violations on specific properties, we propose a transformation from iDola to timed automata, so that verification techniques can be applied. We use the example in Figure 4 to illustrate the transformation.



(a) Source (b) Dispatcher (c) App

Fig. 5: Transformation to timed automata

As in Figure 5, the proposed transformation is feasible to all the modeling constructs in iDola. In (a), $Source$ is represented as an automata with self loops. An interrupt is generated through a transition. $Dispatcher$ is translated as a single-state template in (b). In the template, we use one transition to capture interrupts from $Source$. The dispatching of handlers and normal task is modeled as a pair of transitions, referring to launch and termination. A ISR stack and a DH queue are implemented in the template. Operations of stack and queue are translated as transition actions. When an ISR is invoked, it pushes the corresponding interrupt identifier in the stack. At the completion of an ISR, it pops identifier out of the stack and insert its DH into the queue if it is associated with one. Operations of interrupt mask and unmask are denoted as assignments to local variables. Moreover, (c) exhibits an App in timed automata, it is a representation of its CFG which synchronizes with $Dispatcher$ at the beginning and end transitions. As App, ISR and DH share the same transformation, where CFG is translated into timed automata. Through the proposed transformation, systems specified in iDola can be verified to expose poor designs.

C. Code Generation

Code generation is another strength of iDola, which not only improves development efficiency, but also narrows the gap between system model and the target platform.

In design and simulation of iDola models, system behavior is depicted mostly by simple assignment statements, such as $\mathbf{x = 1}$ or $\mathbf{y = x + 3}$. However, in real complex applications, specific functions are often implemented by calling platform-specific APIs like $\mathbf{arm\_send\_frame(data)}$. To absorb these APIs in the generated codes, we support statements binding between model statements and corresponding implementations. Developers manually do the binding so that code generation can be launched. The algorithm of code generation is exhibited in Algorithm 1, 2 and 3.

Algorithm 1 displays the procedure to generate the whole IDS. Except $Source$ component which is an abstraction of environment, other components including shared data(Var), $Dispatcher$, ISR, DH and App are generated in turn.

Algorithm 1 Generate an IDS

function SYSTEMGENERATE(S)
  VarGenerate($S.Var$);
  DispatcherGenerate($S.dispatcher$);
  for each $isr \in S.ISR$ **do**
    generate_ISR(isr);
  for each $dh \in S.DH$ **do**
    generate_DH(dh);
  for each $app \in S.App$ **do**
    generate_App(app);

Code generation of $Dispatcher$ is displayed in Algorithm 2. The first step is to generate local variables. Then initialization operations specified in Pre are generated to codes based on the CFG. Registration is generated over every interrupt. Commonly, registration includes creation, configuration and attaching. However, implementation of registration differs for different platforms. Thus, manual binding information is used here to generate platform-specific codes. Completion operations in $Post$ are also generated.

Algorithm 2 Generate dispatcher component

function DISPATCHERGENERATE($dispatcher$)
  VarGenerate($dispatcher.Var$);
  PreAction($dispatcher.Pre$);
  for each $intr \in dipatcher.interrupt$ **do**
    InterruptRegister($intr$, $dispatcher.Pri$);
  PostAction($dispatcher.Post$);

Algorithm 3 Generate handler component

function HANDLERGENERATE($handler$, $node$)
  if $node = Start$ **then**
    VarGenerate($handler.Var$);
  if $node = Exit$ **then**
    ReturnGenerate($handler$);
  if $node.child.size > 0$ **then**
    for each $cn \in node.child$ **do**
      BranchGenerate($node$, cn);
      if EnterLoop(cn) **then**
        LoopGenerate(cn);
        ISRGenerate($handler$, LoopExit(cn));
      else if isJointNode(cn) **then**
        registerJointNode(cn);
      else
        HandlerGenerate($handler$, cn);
  if isDominatedBy($node$, $jointNode$) **then**
    HandlerGenerate($handler$, $jointNode$);

Handlers as ISR, DH and App are aimed at servicing interrupts. Therefore, main task of code generation on handler components is to traverse a CFG and generate codes. As in Algorithm 3, a recursive strategy is applied which starts from the $Start$ node. For intermediate nodes, the algorithm generates sequential, branch and loop structures recursively. When a joint node in CFG is detected, the algorithm records it and pauses the traverse on the current path. Until all the branch paths assembles at the joint node, the traverse is resumed. Another specialty is the processing of $Exit$ node. For ISR, normal completion is generated if no DH is associated. Otherwise, the algorithm generates invoking codes based on buffer configurations in $Dispatcher$. For DH and App, normal completion is generated.

IV. CASE STUDY

In this section, we apply iDola on the modeling of multi-function vehicle bus controller(MVBC), which is an industrial system in Chinese railway applications provided by THsoft InfoTech(http://www.thit.com.cn/). MVBC is interrupt-driven since majority of its functionalities are realized by service on interrupts. A picture of MVBC and its architecture is shown in Figure 6.

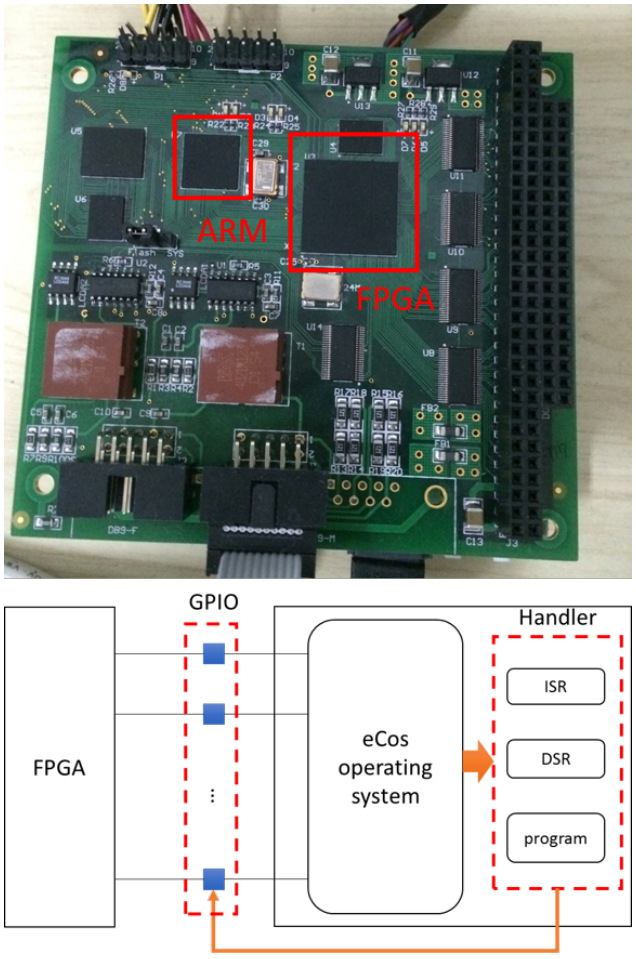


Fig. 6: A picture of MVBC

As marked in Figure 6, MVBC is constituted with FGPA and ARM parts. FPGA is the $Source$ of the system, which procudes interrupts to ARM parts. In the application, seven types of interrupts are defined, including supervisory frame(SPV), main frame(MF), slave frame(SF), error

frame(ERR), timeout(TO), synchronization frame(SYNC) and notime frame(NT). Handler programs are executed in the ARM platform with eCos operating system. ARM is connected to FPGA through GPIO lines. By setting GPIO, FPGA notifies the arrival of interrupts. The eCos then detects the event and invoke corresponding handlers to service the interrupt. Execution of handlers may modifie GPIO and trigger process in FPGA. We develop a modeling and analysis tool of iDola, called Tsmart-Edola. MVBC is modeled graphically in Tsmart-Edola as in Figure 7 with the following specification segment.

```
FPGA{
  MF,SF,SPV,SYNC,NT,ERR,TO;
  start SPV; SF not follow SPV;
  ERR not follow MF; TO not follow MF;
}
dispatcher{
  priority{
    MF=0; SF=0; SPV=1;
    SYNC=(2,2); NT=3; ERR=(4,4); TO=(4,4);
  }
  buffer{capacity = 5}
  queue{capacity = 3}
  post{mvb_device_status_16 = 0;}
  MF->MF_ISR; SF->SF_ISR;
  SPV->SPV_ISR; NT->NT_ISR;
  ERR->(ERR_ISR,ERR_DH);TO->(TO_ISR,TO_DH);
  SYNC->(SYNC_ISR[buffer=3],SYNC_DH);
}
```

In Figure 7, there are 7 ISR in the system, 3 of which are associated with a DH. In respect of FPGA, we define four patterns to describe constraints on the interrupt sequence, including starting and following relationships. In simulation, these patterns can help accurately reflect the real environment of the system. They can also be used in verification to largely reduce the state space. From the specification of $Dispatcher$, we can see that five priority levels are defined. Since MF and SF are top-priorited, they finish handling in ISR to ensure quality of service. SYNC, ERR and TO are attached to DHs, which inherit the priority from ISR, because their occurrence are comparatively less frequent and have lower impact on the system performance. Moreover, post operations are specified to update the value of a global variable. With a FIFO queue which is declared to store maximum five DHs, process of DHs is delayed until all the executions of ISRs complete. Through this way, we can accurately model the interrupt handling mechanism in eCos with ISR and DSR. In the dispatching configuration, ISR of SYNC interrupt is equipped with a 3-count buffer, which assembles three invoking requests to DH before its real dispatching. This feature is also applied in many RTOS to reduce invoking overhead and handle bursty interrupts.

Table I shows the improvement of simplicity and readability brought by iDola. For interrupt handlers, we compare the iDola model with written C codes. An average control flow graph with 9 nodes is specified in iDola to service an interrupt. Comparatively, 32 lines of C codes are written on average. For the $Dispatcher$, we use 13 lines of specification in iDola to assign priorities, build configuration and attach handlers to interrupts. However, 52 lines of codes are neccessary to finish the corresponding tasks. Because of the declarative and concise nature of iDola as a DSL, the design of an IDS can be efficiently mantained. Especially for the specification of $Dispatcher$ which captures complex interrupt handling mechanism of RTOS and target platform, we can quickly locate inappropriate configurations or modify it to adapt new applications.

TABLE I: Comparison between iDola model and written codes of MVBC

Interrupt	**#Handler**	**#Node**	**LOC**
MF	2	11	46
SF	2	6	28
SPV	1	4	14
SYNC	2	6	21
NT	1	3	11
ERR	2	12	47
TO	2	17	53
Dispatcher	**#Handler**	**#SPEC**	**LOC**
dispatcher	1	13	52

Using the proposed transformation, we verify the MVBC model on data race and reachability properties. Data race is a major threat in IDS with the presence of interrupts and reachability checks whether a specific handler can be invoked when an interrupt occurs. The properties are displayed as:

- A[] SYNC_ISR.mvb_status == SYNC
- flag_sf == true $\rightarrow$ SF_ISR.LAUNCH

For data race properties, we check whether the local copy of the global variable is intruded by other routines. Because the variable is an identifier in its service, conflicting write operations may result in functionality confusion. For reachability properties, we check whether an ISR can always be dispatched if the corresponding interrupt ocurrs. This property is also crucial in RTEA since it is closely related to the deadline analysis of interrupt handling. Through verification, a data race counter-example is reported in Figure 8. The reachability properties are satisfied based on patterns of the interrupt source specified in the dispatcher.

The reported data race is a real design defect when DSR of SYNC is preempted by ISR of MF at the accessing point of the global variable. To fix the defect, we should add access protections to the DSR of SYNC to detect conflicting access.

Based on the iDola model of MVBC, we also use the proposed algorithms to generate platform-specific executable codes. The generation is applied on the major impementation of the MVBC, generated codes along with other written C codes and eCos are compiled and burned into the ARM platform for execution. Due to the compatiability, we bind some ARM APIs to actions in CFG of iDola handlers. For instance, the post operation in $Dispatcher$ is binded to an API mvb_arm_send_status(mvb_device_status_16). After the binding, code generation of MVBC can be done without modification on models.

On the PC with dual core processor and 4GB memory, the code generation executes 376ms and uses 37.98MB memory. Figure 9 shows a comparison between the generated codes and the written codes. An obvious difference is that the generated codes has a slightly smaller size in all the interrupt handlers.

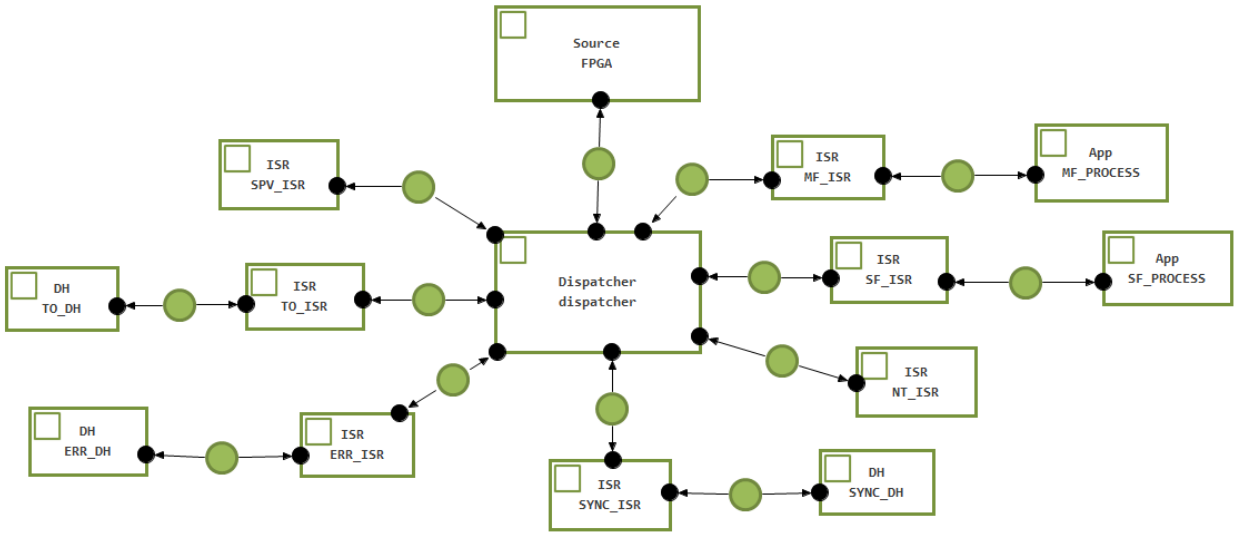


Fig. 7: iDola model of MVBC

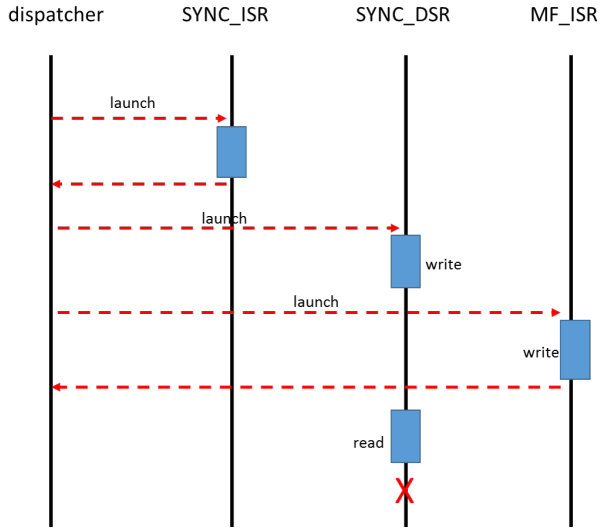


Fig. 8: Counter-example found in verification

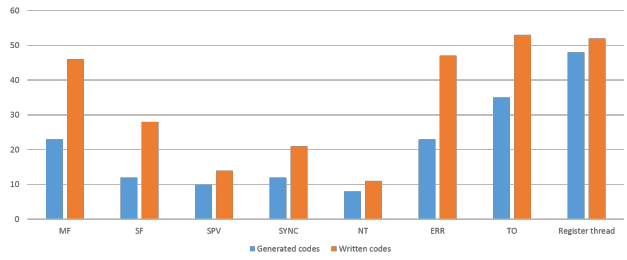


Fig. 9: Comparison between generated and written codes

This is mainly because the written codes contain some debug-related statements, which are ignored during the proposed code generation. For the register thread which is responsible for creating interrupts, registering them in eCos and attaching them to corresponding handlers, generated codes share a equally same size with written codes.

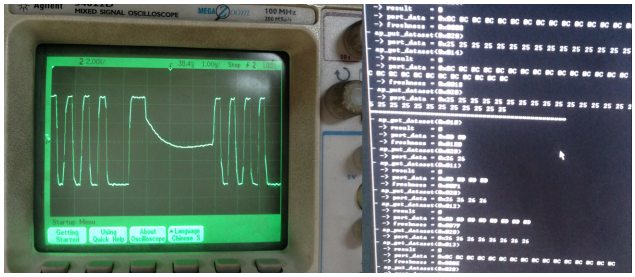


Fig. 10: Communication with generated codes

Equipped with the generated codes, MVBC is accessed into a train communication network to check its functionality and real-time performance. From Figure 10, we can see that the generated codes work well with an average time interval 3.6μs, satisfying the timing requirement in the application domain.

V. CONCLUSION

In this paper, we propose a novel DSL called iDola with formal operational semantics to model IDS in RTEA. The major superiority of iDola is the capability to capture both behavior of handlers and complex interrupt handling mechanism. Based on the advance feature, iDola can model delayed process and buffered service of interrupts, which are supported in many RTOS and target platforms. We further presents a transformation from iDola to timed automata for verification and code generation algorithm to generate executable platform-specific codes. The proposed approach is applied on MVBC, an industrial IDS which executes in a real-time environment

with eCos operating system to handle 7 interrupts. With iDola, we model a dispatcher and all the interrupt handlers. The dispatcher is an abstraction of eCos, which leverages DSR for delayed process and buffer to reduce invoking overhead. Compared to written codes, iDola excels at smaller size and declarative expressiveness. Via the proposed transformation, we verify MVBC on data race and reachability properties. Verification uncovers design defects which may lead to data races for two pairs of interrupts. Furthermore, we run our algorithm to perform code generation on the MVBC model. With almost the same size as written codes, the generated codes are readable and shows comparatively equal performace through real-time communication with strict timing constraints.

VI. ACKNOWLEDGEMENT

This research is sponsored in part by NSFC Program (No. 61202010, 91218302), National Key Technologies R&D Program (No.SQ2012BAJY4052), 973 Program (No.2010CB 328003) of China and Tsinghua University Initiative Scientific Research Program(20131089331).

REFERENCES

[1] B. Berard, S. Haddad, and M. Sassolas. Real time properties for interrupt timed automata. In *2010 17th International Symposium on Temporal Representation and Reasoning*, pages 69–76. IEEE, 2010.

[2] B. Brard and S. Haddad. Interrupt timed automata. *FoSSaCS*, 2009.

[3] B. Brard, S. Haddad, and M. Sassolas. Interrupt timed automata: verification and expressiveness. *Formal Methods in System Design*, 40(1):41–87, 2012.

[4] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE'01*, pages 47–56. IEEE, 2001.

[5] D. Brylow and J. Palsberg. Deadline analysis of interrupt-driven software. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 198–207. ACM, 2003.

[6] R. Chen, X. Guo, Y. Duan, and B. Gu. Static data race detection for interrupt-driven embedded software. In *2011 5th International Conference on Secure Software Integration Reliability Improvement Companion*, pages 47–52. IEEE, 2011.

[7] M. Higashi, T. Yamamoto, and Y. Hayase. An effective method to control interrupt handler for data race detection. In *Proceedings of the 5th Workshop on Automation of Software Test, AST '10*, pages 79–86. ACM, 2010.

[8] C. Hoare. Communicating sequential processes. *Prentice Hall*, 1985.

[9] Y. Huang, J. F. Ferreira, G. He, S. Qin, and J. He. Deadline analysis of autosar os periodic tasks in the presence of interrupts. *Formal Methods and Software Engineering*, 8144:165–181, 2013.

[10] Y. Huang, Y. Zhao, J. Shi, and H. Zhu. A denotational model for interrupt-driven programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 15–20. IEEE, 2013.

[11] Y. Huang, Y. Zhao, J. Shi, H. Zhu, and S. Qin. Investigating time properties of interrupt-driven programs. *Formal Methods: Foundations and Applications*, 7498:131–146, 2012.

[12] J.C.M.Baeten, J.A.Bergstra, and J. W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae*, 1986.

[13] F. JOHNSON and J. M. PAUL. Interrupt modeling for efficient high-level scheduler design space exploration. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(10), 2008.

[14] Kotker, D.Sadigh, and S.A.Seshia. Timing analysis of interrupt-driven programs under context bounds. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 81–90. IEEE, 2011.

[15] K.Salah, Dhahran, and K. Elbadawi. Modeling and analysis of interrupt disable-enable scheme. In *21st International Conference on Advanced Information Networking and Applications*, pages 1000–1005. IEEE, 2007.

[16] B.-K. Lee, M.-H. Kang, K. C. Park, J. S. Yi, S. W. Yang, and Y.-K. Jun. Program conversion for detecting data races in concurrent interrupt handlers. *Software Engineering, Business Continuity, and Education Communications in Computer and Information Science*, 257, 2011.

[17] G. Li, S. Yuen, and M. Adachi. Environmental simulation of real-time systems with nested interrupts. In *Third IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE 2009*, pages 21–28. IEEE, 2009.

[18] J. Regehr. Safe and structured use of interrupts in real-time and embedded software. 2007.

[19] J. Regehr and N. Cooprider. Interrupt verification via thread verification. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 174(9), 2007.

[20] B. Schlich, T. Noll, J. Brauer, and L. Brutschy. Reduction of interrupt handler executions for model checking embedded software. *Hardware and Software: Verification and Testing*, 6405, 2011.

[21] G. M. Tchamgoue, K. H. Kim, and Y.-K. Jun. Dynamic race detection techniques for interrupt-driven programs. In *Proceedings of the 4th international conference on Future Generation Information Technology, FGIT '12*, pages 148–153. ACM, 2012.

[22] G. M. Tchamgoue, K. H. Kim, and Y.-K. Jun. Verification of data races in concurrent interrupt handlers. *International Journal of Distributed Sensor Networks*, 2013.

[23] X. Xu and C.-C. Liu. Modeling interrupts for software-based system-on-chip verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(6), 2010.

[24] Y. Zhao, Y. Huang, J. He, and S. Liu. Formal model of interrupt program from a probabilistic perspective. In *16th IEEE International Conference on Engineering of Complex Computer Systems*, pages 87–94. IEEE, 2011.