Easy Modelling and Verification of Unpredictable and Preemptive Interrupt-driven Systems

Minxue Pan[†‡*], Shouyu Chen[†§], Yu Pei[¶], Tian Zhang[†§*] and Xuandong Li[†§*]
[†]State Key Laboratory for Novel Software Technology, Nanjing University, China
[‡]Software Institute, Nanjing University, China
[§]Department of Computer Science and Technology, Nanjing University, China
[¶]Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China
mxp@nju.edu.cn, remainxy@gmail.com, csypei@comp.polyu.edu.hk, {ztluck,lxd}@nju.edu.cn

***Abstract*—The widespread real-time and embedded systems are mostly interrupt-driven because their heavy interaction with the environment is often initiated by interrupts. With the interrupt arrival being unpredictable and the interrupt handling being preemptive, a large number of possible system behaviours are generated, which makes the correctness assurance of such systems difficult and costly. Model checking is considered to be one of the effective methods for exhausting behavioural state space for correctness. However, existing modelling approaches for interrupt-driven systems are based on either calculus or automata theory, and have a steep learning curve. To address this problem, we propose a new modelling language called *interrupt sequence diagram* (ISD). By extending the popular UML sequence diagram notations, the ISD supports the modelling of interrupts' essential features visually and concisely. We also propose an automata-based semantics for ISD, based on which ISD can be transformed to a subset of hybrid automata so as to leverage the abundant off-the-shelf checkers. Experiments on examples from both real-world and existing literature were conducted, and the results demonstrate our approach's usability and effectiveness.**

***Index Terms*—Interrupt-driven systems, Sequence diagrams, System modelling, Model checking.**

I. Introduction

Interrupt-driven systems, where processing is initiated by interrupt requests, are gaining popularity since interrupts are a key design primitive for software systems that actively make interactions among system components and closely interact with the environment. They are commonplace in all styles of computing platforms, including safety-critical embedded platforms, low-power mobile platforms and high-end information systems [1]. Particularly, most cyber-physical systems are interrupt-driven, since interrupts are an extremely common form of concurrency that the control software uses to obtain sensor data from its physical environment [2], and enable timely response to outside stimuli in a power-efficient way [1]. However, interrupts can cause problems, for many of them can happen at an arbitrary time and preempt the running tasks, which adds non-determinism and concurrency to the systems. This poses challenges to the development of reliable interrupt-driven systems, as designers have to explicitly handle unpredictable system behaviour caused by interrupts. As a consequence, interrupt-driven systems are error-prone [3], [4], [5], and need extensive efforts for quality assurance.

$*$ Corresponding author.

Testing is one of the primary ways to assure the quality of systems. However, existing testing techniques for sequential programs [6], [7] or even concurrent programs [8], [9] have not addressed the problems caused by interrupts adequately. They often cannot identify or capture the concurrency brought about by interrupts precisely. To address this limitation, researchers have tried interrupt scheduling algorithms that fire interrupts at proper points of time [5], or suitable test adequacy criteria to guide and evaluate the testing process [4]. Nonetheless, testing of interrupt-driven systems can still be insufficient. The generation of interrupt requests is usually random and non-deterministic, and the interrupt handling is often preemptive and nested, which results in that the number of possible system behaviours grows exponentially in the number of occurred interrupts, while defects related to specific behaviour are difficult to detect by testing approach [5].

Model checking, on the other hand, can rigorously verify a system's behaviour by exhaustively exploring the state space of a software system. Recent studies have noticed the significance and uniqueness of interrupt-driven systems, and new modelling languages are proposed. For example, interrupt time automata (ITA) [10], [11] are proposed to model multi-task systems with interrupts. They form a subclass of stopwatch automata [12], where the real-valued variables (with a rate of 0 or 1) are organised along priority levels. ITA are powerful in expressiveness, however, we argue that industrial designers may find them difficult to use. When modelling with ITA, designers have to consider all possible interleavings of states, as well as the clocks that specify the timing requirements. On the other hand, UML sequence diagrams [13] offer an intuitive and visual way of describing interactions among system components and the environment. They are widely used in industry. According to a survey conducted in [14], sequence diagrams were recognised as one of the most frequently used diagrams, and system analysts and programmers admitted that they rely most on sequence diagrams along with class diagrams to capture requirements and exchange information. However, sequence diagrams are still inadequate to model interrupt-driven systems. Interrupts' arrival can be unpredictable, and their handling is preemptive and prioritised. Time could be a complex concept too, since the execution time of tasks and interrupt service routines (ISRs) can be interrupted and resumed. These interrupt-specific features are not supported by sequence diagrams. To overcome these limitations, we propose the *Interrupt Sequence Diagram (ISD)* which extends the sequence diagram with interrupt modelling mechanisms.

The extension adopts the UML standard notation for easy comprehension. We propose to introduce a new *Combined-Fragment "int"* to specifically model interrupts. A Combined-Fragment `int` can model an interrupt's unpredictable arrival and its handler's prioritised preemptive execution. As for time, tasks/ISRs in interrupt-driven systems can often be preempted during execution, which means the actual execution time does not equal to the time duration from the beginning to the completion of the tasks/ISRs. Considering that the UML sequence diagram only supports the latter form of timing constraints, which is not sufficient for interrupt-driven systems, we propose a new kind of timing constraints called *task constraints* to model the *actual* execution time of tasks/ISRs.

To facilitate formal verification, we also provide an automata-based semantics for ISD. We use integration automata (IA) [15], [16], a subset of hybrid automata, to interpret the semantics of ISD, which, also enables the employment of the abundant off-the-shelf hybrid automata checkers for verification. We conducted multiple case studies with the state-of-the-art tool SpaceEx [17], and the experiment results consistently showed that our approach can effectively find defects. Particularly, we believe that the best trait of ISD is that it is easy to learn and use, and therefore, we conducted an experiment to compare the usability of ISD and ITA. The results confirmed ISD's good usability. The main contributions of this paper are:

- We propose a novel modelling language ISD to specify the interrupt-driven systems. To our best knowledge, ISD is the first sequence diagram based language which supports the specification of interrupt's unpredictable arrival, prioritised preemptive handling and the time suspension and resumption;
- We propose an automata-based semantics for ISD, based on which ISD can be transformed to hybrid automata for correctness checking;
- We developed a tool named ISDChecker, which is, to our knowledge, the first available tool that checks graphical models for interrupt-driven systems. Evaluation on previous studied and real-world cases shows ISDChecker's effectiveness and usability.

The rest of the paper is organised as follows. In the next section, we introduce the UML sequence diagram and a modelling example. Section 3 proposes the ISD's notation, and Section 4 presents its automata-based semantics. Section 5 conducts a series of experiments to evaluate our approach's usability, effectiveness and efficiency. Section 6 discusses the related work, and Section 7 draws the conclusion.

II. Sequence Diagram and Motivating Example

UML sequence diagrams form a class of important UML interaction models. Each of them describes a set of interactive scenarios, as Fig. 1 shows an example. There are two

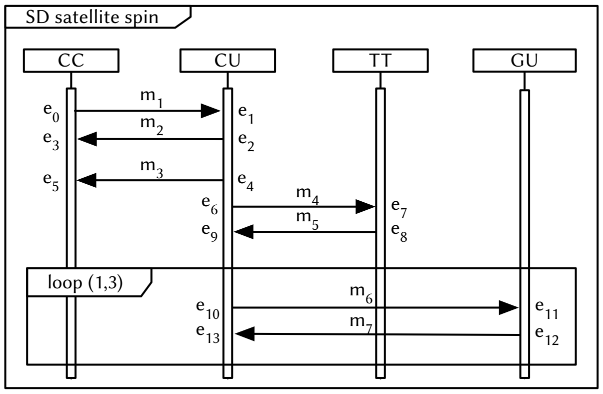

Fig. 1. A sequence diagram example

dimensions in a sequence diagram. The vertical dimension represents time, and the horizontal dimension consists of different lifelines representing participating entities. Information exchange between lifelines is carried out by messages represented by arrows. In the simplest form, a sequence diagram depicts the desired exchange of messages and corresponds to a single execution of the system. To specify complex scenarios conveniently, sequence diagram supports operations such as choice and iteration through *CombinedFragments* (or *fragments* for short). A fragment is defined by an interaction operator and one or more interaction operands. The notation for a fragment is a solid-outline rectangle, and the operator is shown in a pentagon in the upper left corner of the rectangle. Each operand is composed of a subset of messages in the diagram. In this paper, we consider three most frequently used interaction operators, which are *loop*, *alt* and *opt*. The operator `loop` designates that the fragment represents a loop, so its operand will be repeated a number of times. A guard that may include a lower and an upper number of iterations of the loop can be associated with the fragment. Fig. 1 shows an example of the `loop` fragment, where the iteration times is restricted from 1 to 3 times. The operator `alt` designates that the fragment represents a choice of behaviour. At most one of the operands will be chosen, and the chosen operand must have a guard expression that evaluates to true at this point in the interaction. The operator `opt` has a similar meaning as the operator `alt`, except that it has only one operand. The fragment `opt` specifies the behaviour where either the operand happens or nothing happens. In this paper, we enforce strict sequencing on the fragments. Consequently, a fragment will cover all lifelines, so that when the execution control of flow enters a fragment, all lifelines enter the fragment.

Sequence diagrams are popular in industry, because they help designers focus on the most frequent or critical scenarios. However, they are insufficient to model interrupt-driven systems. To support this claim, we present a real-world case of designing the spin action in a satellite controlling system. The original sequence diagram provided by the designers was an informal sketch, which we revised to conform to the UML standard and to exclude sensitive information. As shown in Fig. 1, the spin action, involving four participating entities, consists of three operations. In the first operation, the Command Centre (CC) sends an inquiry to the satellite's Computing Unit (CU) about the status of the satellite (Message m_1 and m_2). Then, in the second operation, CU informs CC that it is going to produce the instructions for the satellite to spin an angle to better absorb sunlight, and in the meantime, will not response to CC commands (Message m_3). Then CU sends the instructions to the thruster (TT) to be executed (Message m_4) and TT acknowledges CU when the spin action is completed (Message m_5). In the third operation, the satellite communicates with the Ground Unit (GU) periodically to exchange information (Message m_6 and m_7). The communication cannot be interrupted by other operations, or the communication link would be lost. Should the interpretation of the diagram in Fig. 1 strictly follow the UML specification, the system should be running without problems, because in a UML sequence diagram, events corresponding to message sending and receiving are subject to predefined partial orders deduced from the visual order of the diagram (see [13] for more details about the partial orders among events). Thus, in Fig. 1, e_4 would be considered to happen after e_2, and e_{10} after e_9, which means the three operations happen in sequential order. However, this sequence does not cover all the actual system behaviours. In reality, the operation of CC querying CU and the operation of CU communicating with GU are both interrupts, so their arrivals are unpredictable. Even worse, the former has a higher priority than the latter, which means that the inquiry from CC to CU is not obliged to happen before CU and GU's communication, but can interrupt the communication process and cause a problematic behaviour. The engineers had not noticed the problem and implemented the system as designed, which resulted in a costly failure during the system integration test. To prevent this kind of disaster, it is expected that the problematic behaviour be captured by the designs and found by some sort of checking methods. Unfortunately, because of the partial orders enforced on the events, it is impossible for a UML sequence diagram to specify unpredictable interrupts. In the following section, we propose a smooth extension to the UML sequence diagrams to allow the specification of unpredictable and preemptive interrupt-driven system behaviours.

III. Interrupt Sequence Diagram

The Interrupt sequence diagram (ISD) is an extension of the UML sequence diagram to model interrupt-driven systems. We notice that the occurrences of interrupts are often spontaneous and unpredictable, and therefore, propose a new fragment called *int*, consisting of one operator and one operand, to model interrupts. The operator `int` declares that the fragment represents an interrupt behaviour, whose syntax is:
'int' ⟨priority_exp⟩ [⟨occur_bound⟩] ['['⟨mask_cond⟩']'].*
priority_exp models the interrupt's priority, *occur_bound* specifies the times the interrupt can occur and the separation in time between two consecutive interrupt arrivals, and *mask_cond* models the interrupt mask condition. In the following, we give detailed explanations about different parts of the `int` fragment.

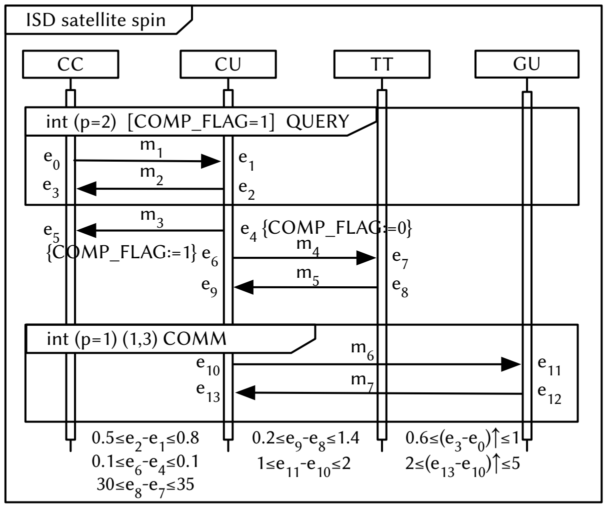

Fig. 2. An interrupt sequence diagram example

A. Order of Events

The events in the operand of the `int` fragment specify the execution process of the interrupt arrival and handling. As an event will not belong to two interrupts, no overlapping of two `int` fragments is allowed. To depict interrupt's unpredictability, it is required that there should be no partial orders deduced from the visual orderings of messages between the events from inside and outside an `int` fragment, respectively. For example, in Fig. 2, e_2 and e_4 are two events inside and outside an `int` fragment, respectively, and thus there is no ordering requiring that e_2 occur before e_4.

To specify the prioritised preemption of interrupts, a *priority expression* is proposed, defined as *⟨priority_exp⟩::= '(p=' ⟨priority⟩ ')'*, where *⟨priority⟩::=non-negative natural*. The *priority* of the `int` fragment comes from the priority of the interrupt and applies to all the events in the fragment. To be consistent, we assign a default priority to the entire diagram, which applies to events not in any `int` fragment. In fact, excluding the `int` fragments, the entire diagram with its enclosed `loop`, `alt` and `opt` fragments can also be viewed as a fragment, except its priority is the lowest. From now on, we use the term *interaction fragment* indiscriminately to represent the `int` fragment or the diagram excluding all `int` fragments. The priority of the interaction fragments introduces a new kind of orderings among events: when the execution control of flow is in an interaction fragment of priority p_1, it can move to another interaction fragment of priority p_2 if $p_2 > p_1$, but not vice versa. Formal semantics of the `int` fragment is presented in Sec. IV-B4.

Let us revisit the example of the satellite spin and re-model it with ISD, as shown in Fig. 2. There are two `int` fragments: QUERY and COMM. QUERY has a priority of 1, and COMM has a priority of 2. So, events in `int` COMM can occur after the occurrence of e_{10} and before the occurrence of e_{12}, causing the communication between CU and GU to fail. With the help of `int` fragments, a defect of the design can be revealed.

B. Interrupt Mask

Designers often use interrupt masks to prevent some less important interrupts from interfering with more crucial tasks. The ISD supports modelling the interrupt masks by using *mask variable* updates and tests. Each interrupt mask is modelled by a mask variable, with a value from $\{0, 1\}$. The enablement or disablement of the interrupt mask is achieved by setting the variable's value to 1 or 0, respectively. In UML sequence diagrams, the values of variables are updated via actions associated with messages, whereas in ISD we need a more accurate mechanism to model mask variables since the enablement/disablement of interrupts always happens at the beginning or the end of uninterruptible operations. Thus, the ISD supports variable updating actions to be associated with events, since it is the events that represent the beginning and the ending of message processing.

An **int** fragment can have *mask conditions*, which are defined as *⟨mask_cond⟩::= ⟨mask_variable⟩'=1'*, where *⟨mask_variable⟩::=letter tokens*. The events in the **int** fragment can happen when all mask conditions are evaluated as 1, or no mask condition is provided. For example, the **int** fragment COMM in Fig. 2 has one mask condition "COMP_FLAG=1". Therefore, the events in this fragment can happen when variable COMP_FLAG is set to 1.

C. Modelling of Time

The other mechanism to reduce the unpredictability imposed by interrupts is the timing specification. Timing specification can specify the timing properties of the system. Designers can designate the time point or interval in which an interrupt could happen, or the duration of the interrupt handler, to restrict interrupts' behaviour and reduce their uncertainty. For example, suppose that a system consists of one uninterruptible task and one interrupt. If some timing constraints specify that the interrupt would not occur during the execution of the task, no interrupt mask would need to be disabled, which could save the computing resource and time. In UML sequence diagrams, the timing mechanisms are all about the time duration between two events, which specifies how much time has passed from the occurrence of one event till the other. By using event names to represent the occurrence time of events, the timing constraint can be defined as $a \leq e - e' \leq b$ (a, b are real numbers, b may be ∞), which requires that the time duration from the occurrence of e' till the occurrence of e be within the range $[a, b]$. Although this kind of timing constraints is sufficient for most systems, they cannot meet the need to specify interrupt-driven systems. The designers of interrupt-driven systems often need to specify the actual execution time of a task, however, it is not the time duration between the task's start and completion events, for its execution can be interrupted by unpredictable interrupt occurrences. For example, in the satellite spin example in Fig. 2, although one may know that executing the interrupt COMM (from the occurrence of e_{10} to that of e_{13}) requires 2 to 5 time units, it is incorrect to model this time information using $2 \leq e_{13} - e_{10} \leq 5$, because the execution of COMM can be suspended by QUERY. We provide a new mechanism called *task constraint*. A task constraint, denoted as $a \leq (e - e') \uparrow\leq b$, is about two events e and e' satisfying that both events are from the same interaction fragment f. The value of $(e - e') \uparrow$ is computed as the time duration from the occurrence of e' till the occurrence of e, subtracting the time when the diagram's execution control of flow is not in f. For example, the task constraint $2 \leq (e_{13} - e_{10}) \uparrow\leq 5$ in Fig. 2 specifies the time that the execution control of flow stays in the fragment COMM is between 2 to 5 time units. Formal semantics of timing and task constraints are given in Section IV-B5.

Furthermore, it is possible to specify the minimum and maximum times an interrupt can occur, and, for interrupt that can occur multiple times, the minimum separation in time between two consecutive interrupt arrivals. We use the expression *occur_bound* to specify these timing requirements, which is defined as *⟨occur_bound⟩::='('⟨min⟩','⟨max⟩ [','⟨separation⟩]')'*, where *⟨min⟩::=positive natural, ⟨max⟩ ::= positive natural (greater than or equal to ⟨min⟩) | '∞', ⟨separation⟩::=positive real.* Without an explicit *occur_bound*, the interrupt is supposed to occur exactly once by default, and, without the *separation* field provided, arbitrary separation time (positive real) is allowed. For example, the *occur_bound* $(1, 3)$ of the **int** fragment COMM specifies that it can occur 1 to 3 times, and the minimum separation time can be arbitrary.

D. ISD Syntax

Now we can formally define the syntax of ISD as follows.

Definition 1: An interrupt sequence diagram (ISD) is a tuple $D = (L, E, M, R, V, U, F, C)$, where

- L is a finite set of lifelines;
- E is a finite set of prioritised events whose elements are pairs (e, p), where e is the event and p is its priority.
- M is a finite set of messages. For each $m \in M$, $m = (e, e')$, where $e, e' \in E$ correspond to the sending and the receiving of m, respectively;
- $R : E \to L$ is a labelling function which maps each event $e \in E$ to a its sending (receiving) lifeline $R(e) \in L$;
- V is a finite set of message orderings whose elements are a pair (m, m') $(m, m' \in M)$ such that m visually precedes m';
- U is a finite set of mask variable updates, whose element is a tuple (var, val, e) where var is a variable, $val \in \{0, 1\}$ is the updated value and $e \in E$ is the event associated with the variable update;
- $F = F_{loop} \cup F_{alt} \cup F_{opt} \cup F_{int}$ is a finite set of fragments. Each element $f \in F$ is a tuple (o, g, M_f) where $o \in \{loop, alt, opt, int\}$ is the operator, g is the guard expression of f, and $M_f \in 2^M$ is a subset of M;
- C is a finite set of timing constraints and task constraints.

IV. IA Based Semantics

We define an automata-based semantics for ISD to facilitate formal verification. The semantics of ISD is interpreted by integration automata (IA), which are a special case of hybrid automata. In this section, we will first visit the concept of IA, followed by the presentation of the semantics definition.

A. *Integration Automata*

Hybrid automata are finite automata extended with a finite set of real typed variables whose values change continuously at each location. The change rates of the variables are designated by the *flow conditions* associated with locations. There are also *invariants* in each location indicating that the conditions need to be satisfied when the location is active. Transitions between locations are guarded by *jump conditions* on the variables, and their executions may reset some of the variables by the *reset actions*. If the invariants, jump conditions and reset actions are all linear expressions over the variables, and the flow conditions specifying the allowed values of the first derivatives of the variables are either 0 or 1, then the hybrid automata are called integration automata. We can see that the restriction on flow conditions of integration automata makes it suitable to model time suspension and resumption. Formal definitions of the integration automata can be found in [16].

B. *ISD Semantics*

We follow the generally agreed semantics of the basic fragment and the **`alt`**, **`opt`** and **`loop`** fragments. Furthermore, we define the semantics for the **`int`** fragment, task constraints and mask variables designed for modelling interrupts.

A common approach to interpreting sequence-based diagrams using automata is to use one automaton for every object and their parallel composition for the whole diagram [18], [19]. The composition requires the synchronisation between different object automata, which is achieved by using synchronisation labels corresponding to the message names. Since transitions with the same synchronised label must happen simultaneously, it is impossible to distinguish the message sending and receiving events over time. In interrupt-driven systems, the time duration between events plays a vital role in correct system functions. Thus, in contrast to this approach, we propose to use fragment as a basic unit for semantics interpretation, similar to [20].

Algorithm 1: Algorithm of constructing an automaton for a basic fragment

```
1  construct the initial location $q_0$;
2  for each non-final location $q$ that has no outgoing edges do
3      Acquire the set $L$ of events in the path from $q_0$ to $q$;
4      for any event $e \in (E - L)$ do
5          if any event $e'$ satisfying $(e' \prec e) \in O$ is in $L$ then
6              construct a location $q'$ and a transition $(q, e, q')$;
7          for any location $q''(q'' \neq q')$ do
8              $L' \leftarrow$ set of events in the path from $q_0$ to $q'$;
9              $L'' \leftarrow$ set of events in the path from $q_0$ to $q''$;
10             if $L' = L''$ then
11                 $q'' \leftarrow$ merge $q'$ and $q''$;
12                 change $(q, e, q')$ to $(q, e, q'')$;
13             end
14     end
15     if $q$ has no outgoing edges then
16         mark $q$ as final;
17 end
```

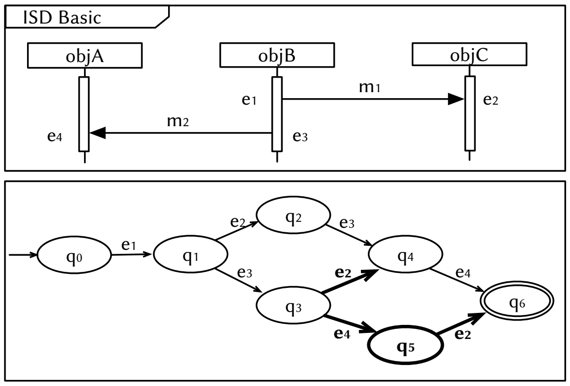

Fig. 3. A basic fragment and its corresponding automaton

1) The Basic Fragment: The simplest form of ISD is a basic fragment that does not have any nested fragments. The behaviour of a basic fragment is a set of event sequences, which are permutations of events, satisfying a partial order relation of events deduced from the visual orderings. So, to interpret the behaviour of a basic fragment, we can use an automaton, of which the set of event sequences along all paths is equivalent to the set of event sequences of the fragment. Given a basic fragment, let E be its event set and O be the set of event orderings where $e \prec e'$ indicates that e must occur before e'. Algorithm 1 takes E and O as its input and outputs an automaton to interpret the fragment. It focuses only on the event sequences, while the timing and task constraints are handled in Sec. IV-B5. We use an example to explain the algorithm. Fig. 3 shows the example of a basic fragment and its corresponding automaton. The fragment has a set of events $E = \{e_1, e_2, e_3, e_4\}$ and a set of event orderings $O = \{e_1 \prec e_2, e_1 \prec e_3, e_3 \prec e_4\}$. Suppose we follow Algorithm 1 and has constructed a part of the automaton, which is drawn in thin lines. Now for location q_3, we acquire the set $L = \{e_1, e_3\}$ which comprises events in the path from q_0 to q_3 *(line 3)*. Following *line 4* we pick event e_2 because it is in $E - L$. Since e_1 is already in L, the condition in *line 5* is satisfied, and a location q' and a transition (q_3, e_2, q') is constructed (*line 6*). Location q' is not shown in Fig. 3 because when executing *line 7-10*, we find that the set of events in the path from q_0 to q' is the same as from q_0 to q_4. Since q_4 is an existing location, we merge q' into q_4 *(line 11)* and change (q_3, e_2, q') to (q_3, e_2, q_4) *(line 12)*. Then we loop back to *line 4*, pick event e_4, and construct new locations and transitions in the same manner. When q_6 is picked, since there are no events left for q_6 to grow outgoing edges, we mark it final. Obviously, for the constructed automaton, there is only one initial location, and it is easy to prove that there is only one final location as follows. Suppose there are more than one final location. Since the events along each path to the final locations are all the same, the final locations can be merged as one. Guaranteeing that there is only one initial location and one final location is useful when the automaton needs to be connected with other automata to interpret nested fragments.

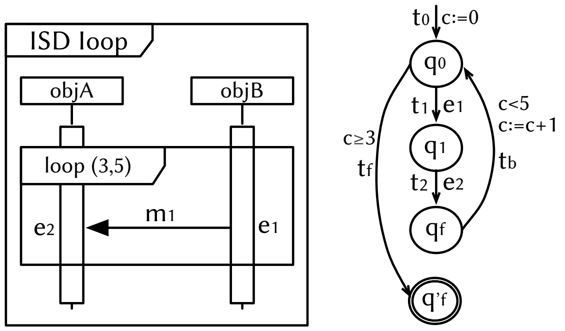

Fig. 4. A **loop** fragment and its corresponding automaton

*2) The **loop** Fragment:* We require a strict sequencing for the **loop** fragment, which means that the events in one iteration can happen only when all the events in previous iterations have happened. If the **loop** fragment contains just one basic fragment, we interpret it with the automaton A constructed by Algorithm 1 with the following modifications. Let q_0 be the initial location, and q_f be the final location of A. The automaton A_L interpreting the **loop** fragment is constructed based on A as follows:

- add a new transition t_b from q_f to q_0 to form loops;
- construct a new location q'_f, and add a new transition t_f from q_0 to q'_f. Make q'_f as the final location and q_f as the non-final location;
- use a variable c to count the iteration times. Initialise c to 0 on the transition to q_0, and increase c by 1 on t_b;
- for the guard (a, b) restricting the iteration times, assign a constraint $c < b$ to t_b, and a constraint $c \geq a$ to t_f.

Fig. 4 shows an example. For the ISD in Fig. 4, locations q_0, q_1 and q_f and transitions t_0, t_1 and t_2 constitute the automaton obtained by applying Algorithm 1. Transition t_b is added to form the loop, and its condition $c < 5$ ensures that the iteration times will not exceed 5 (this condition is checked before the increase of c). Location q'_f and transition t_f are added so that when exiting the loop, there is a condition $c \geq 3$ ensuring that the iteration times will be at least 3.

If there are other fragments nested, the interpretation can be done from inside to outside recursively. Specifically, for an enclosing fragment that has nested fragments, we first interpret the nested ones. Without the nested fragments, the rest of the enclosing fragment is divided into separate parts which are later interpreted individually. Then the automaton corresponding to the entire fragment can be obtained by connecting one automaton's final location to the other's initial location, following the visual orderings of the nested fragments and the parts they separated in the enclosing fragment. The only exception is that the nested fragment is an **int** fragment, which will be discussed in Sec. IV-B4.

*3) The **alt** and **opt** Fragments:* In the **alt** fragment, each operand has a guard condition. The events in an operand can happen only if the guard of this operand is true when the control of flow reaches the fragment, similar to the `if-else` structure in programming languages. To interpret an **alt** fragment, we first use two automata A_1 and A_2 for the two operands $o1$ and $o2$, respectively, and then compose them into a single one as follows:

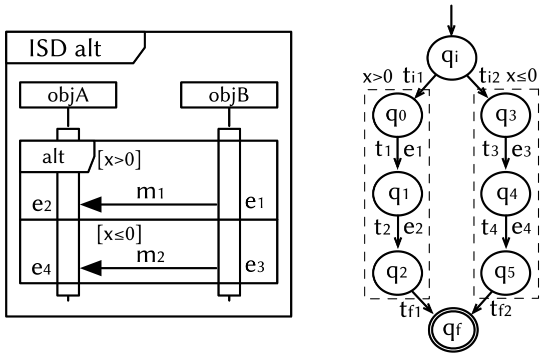

Fig. 5. An **alt** fragment and its corresponding automaton

- construct a location q_i and add transitions t_{i1} and t_{i2} from q_i to the initial locations of A_1 and A_2, respectively;
- construct a location q_f and add transitions t_{f_1} and t_{f_2} from the final locations of A_1 and A_2 to q_f, respectively;
- add the guard condition of $o1$ to transition t_{i1}, and that of $o2$ to transition t_{i2};
- make q_i the initial location and q_f the final location of the composed automaton.

An example is shown in Fig. 5. The parts in dotted boxes are the two automata interpreting the two operands of the **alt** fragment. The reason we use two new location q_i and q_f is to ensure that every automaton corresponding to one fragment has only one initial location and one final location, to simplify the interpretation of nested fragments.

The **opt** fragment can be viewed as a special case of the **alt** fragment with only one operand. Its interpretation is more straightforward: the operand is interpreted first, and the guard condition is added to the initial transition to the initial location.

*4) The **int** Fragment:* Since the occurrence of interrupts is unpredictable, it is required that the events in the **int** fragment have no partial orders with events outside the fragment. In other words, the events in the **int** fragment and the enclosing fragment form two independent event sets, and therefore, the **int** fragment and its enclosing fragment can be interpreted separately.

In interrupt-driven systems, when an interrupt request occurs, it is accepted when its priority is higher than the current task and the interrupt mask (if there is any) is enabled. When the preemptive interrupt completes, it returns the control of execution to the preempted task if there is no interrupt request of higher priority. We refer to this interrupt handling mechanism to interpret the relationship between the two automata corresponding to an **int** fragment and its enclosing fragment. Let A_I (q_{I_0} and q_{I_f} are the initial and final locations, respectively) be the IA interpreting an **int** fragment and A_E (q_{E_0} and q_{E_f} are the initial and final locations, respectively) be the IA interpreting its enclosing fragment. It is assumed that all events in A_E have lower priorities than those in A_I, which can be guaranteed by composing the automata corresponding **int** fragments in ascending order of priorities. Then the IA

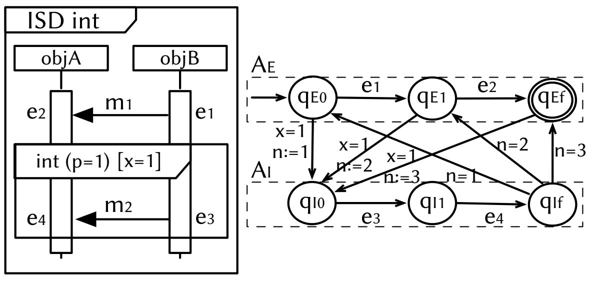

Fig. 6. An **int** fragment and its corresponding automaton

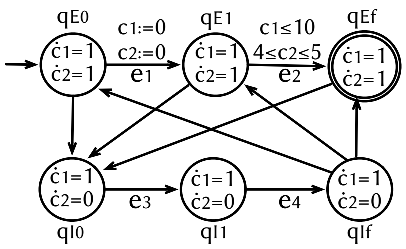

Fig. 7. Interpretation of task and timing constraints

A having the equivalent behaviour as the ISD consisting of both the **int** and its enclosing fragments can be obtained by composing A_I and A_E as follows:

- for any location q_i of A_E, add a transition from q_i to q_{I0} of A_I and associate it with a variable assignment $n := i$ and the interrupt mask condition of the **int** fragment if there is any;
- for any location q_i of A_E, add a transition from q_{If} to q_i and associate it with a guard condition $n = i$;
- make q_{E0} and q_{Ef} as the initial and final locations of A, respectively.

In the example shown in Fig. 6, the two parts in dotted boxes labelled with A_I and A_E are the automata corresponding to the **int** fragment and its enclosing fragment, respectively. In a location of A_E, e.g., q_{E1}, the control of flow can transfer to A_I, provided that the interrupt mask x equals to 1. The assignment $n := 2$ on the transition from q_{E1} to q_{I0} ensures that when exiting from A_I, the control of flow would return to q_{E1} by taking the transition with guard condition $n = 2$.

5) The Timing and Task Constraints: Besides the common timing constraints, the ISD offers the new notion of task constraints. Whereas timing constraints can be easily interpreted by clock constraints in timed automata [21], the task constraints are beyond the expressiveness of the classic clock constraints. Suppose that we have a task constraint $a \leq (e - e') \uparrow\leq b$. When the control of flow is not in the fragment to which e and e' belong, the clock needs to freeze, which means a change in the clock flow rate. Fortunately, the IA supports this change of rate for variables.

For each timing constraint $a \leq e - e' \leq b$, we use one variable c to represent a clock. We make the initialisation $c := 0$ at the transition with label e', and set a flow condition $\dot{c} = 1$ in every location. At the transition with label e, the value of c equals to the time duration $e - e'$, so we add the jump condition $a \leq c \leq b$ to the transition. Fig. 7 shows the same automaton as the one in Fig. 6 with the addition of the interpretation of the timing and task constraints (assignments and conditions irrelevant to time are omitted). Suppose for the ISD in Fig. 6, there is a timing constraint $e_2 - e_1 \leq 10$ specifying that the deadline for the completion of transferring message m_1 should within 10 time units, and a task constraint $4 \leq (e_2 - e_1) \uparrow\leq 5$ specifying that the actual time for transferring m_1 takes 4 to 5 time units. Then for the timing constraint $e_2 - e_1 \leq 10$, we generate a variable c_1, initialise c_1 on the edge labelled with e_1, and add a constraint $c_1 \leq 10$ on the edge labelled with e_2. The flow condition of c_1 is set to $\dot{c}_1 = 1$ in all locations.

For each task constraint, we also generate a variable c' and initialise it to 0 at the transition labelled e'. Different than the timing constraints, the variable c' does not increase in all the locations but only in those corresponding to the interaction fragment to which e and e' belong (recall that the term interaction fragment can represent either the **int** fragment or the diagram excluding all **int** fragments). Therefore, locations corresponding to the interaction fragment to which e and e' belong are equipped with a flow condition $\dot{c}' = 1$; while other locations are equipped with a flow condition $\dot{c}' = 0$. In this way, when reaching the transition with label e, the value of the variable c' would represent the actual execution time that the control of flow stays in the same interaction fragment between the occurrences of e and e', so the jump condition $a \leq c' \uparrow\leq b$ can be added to that transition. For example, for the task constraint $4 \leq (e_2 - e_1) \uparrow\leq 5$, a variable c_2 is used and initialised on the edge labelled with e_1, and a constraint $4 \leq c_2 \leq 5$ is associated with the edge labelled with e_2. The flow conditions in locations q_{I0}, q_{I1} and q_{If} corresponding to the **int** fragment are set to $\dot{c}_2 = 0$, and in locations q_{E0}, q_{E1} and q_{Ef} corresponding to the enclosing fragment are set to $\dot{c}_2 = 1$.

C. Specification and Verification of Properties

Whereas the ISD models how the system behaves, the properties to be checked are captured by property specification. For ISD, the property needs to specify both temporal orderings and time durations between events. Motivated by our goal of easy modelling and verification, we propose a simple specification language defined as $\langle$*spec_clause*$\rangle$::=$\langle e_1\rangle$'$\prec$'$\langle e_2\rangle$ $|\langle$*min*$\rangle$'$\leq$'$\langle e_2\rangle$'-'$\langle e_1\rangle$'$\leq$'$\langle$*max*$\rangle$, where $\langle e_1\rangle$*::=event name*, $\langle e_2\rangle$*::=event name* ($e_2 \neq e_1$), $\langle$*min*$\rangle$*::=non-negative real*, $\langle$*max*$\rangle$ *::= non-negative real (greater than or equal to* $\langle$*min*$\rangle$*)* | '∞'. A specification is composed of one or more spec clauses. $e_1 \prec e_2$ specifies that e_1 occurs before e_2 in temporal order, and $min \leq e_2 - e_1 \leq max$ specifies that the time duration from the occurrence of e_1 to that of e_2 is between $[min, max]$ time units. The specification language is sufficiently expressive in specifying various properties in interrupt-driven systems. For example, for the property of timeout freeness, one can use $0 \leq e_2 - e_1 \leq bound$ to specify that for a task starting by event e_1 and completing by event e_2, it shall not exceed the given time bound.

The verification is to check whether there is a behaviour in the IA that can reach the final location and satisfies the negation of the properties, and if so, the properties are not satisfied, and a counterexample represented by the behaviour is reported. For a property of the form $e_1 \prec e_2$, its negation is $e_2 \prec e_1$, and we first find all paths in which e_2 occurs before e_1 and then check if any behaviour of such paths can reach the final location. For a property of the form $min \leq e_2 - e_1 \leq max$, its negation is $e_2 - e_1 < min$ and $e_2 - e_1 > max$ (if $max \neq \infty$). We translate the negation to clock constraints, add them to the IA, and check if there is a behaviour that reaches the final location. The checking of the existence of behaviours reaching the final location is essentially a reachability analysis problem and can be solved by exploiting the existing hybrid automata checkers.

V. Experimental Evaluation

We implemented our approach as a prototype called *ISDChecker*, which supports the modelling and verification of ISD. The graphical modelling interface is based on UMLet [22] which is a free, open-source UML tool. The verification is conducted by transforming the ISD to an IA based on the ISD semantics, and feeding the IA to SpaceEx [17], which is a state-of-the-art tool for verifying safety properties of hybrid systems. Note that there is no restriction of the choice of hybrid automata checkers so long as the verification of IA is supported.

Our approach of modelling and verifying interrupt-driven systems with ISDs aims to guarantee the correctness of the systems. Ideally, it should be effective in finding counterexamples, and at the same time, easy to use. Therefore, our evaluation addresses the two following research questions:

- ***RQ1:*** *Can our approach effectively detect problems in interrupt-driven system models?*
- ***RQ2:*** *Is ISD easy to use compared with existing modelling languages for interrupt-driven systems?*

A. RQ1: Approach Efficacy

To conduct the experiments, we searched papers published after year 2010 using keywords "interrupt driven", "interrupt program", "interrupt software", "interrupt system", "embedded system" or "real-time system" combined with keywords "verification", "testing", "analysis" or "model checking", and collected 18 closely related to interrupt-driven system papers, from whose references we snowballed 7 more related papers published after year 2000. We studied these papers to collect cases that satisfy the following criteria: (1) they are not toy cases without realistic settings, and (2) they have been detailedly presented using models, programs and/or textual descriptions. As most papers just use one or two cases and many of them are toy cases, in the end, 5 cases were collected from the existing literature. We modelled these cases with ISD, which are shown in Column *case name* with references. To evaluate the effectiveness of ISDChecker, we expect the cases to contain problems so that we can check whether ISDChecker can find these problems. The case *ADC_Bug* itself has a data race problem. For the other 4 cases, *medical_monitor* and *car_controller* have execution time bound requirements, and we modified the time values in ISDs to violate the requirements. The timeout problems in cases *attitude_display* and *fridge_controller* were manually inserted, as the original examples did not mention any temporal or time properties. We also consulted the experts in the aerospace area and acquired 6 flawed design cases. These cases were modelled with UML sequence diagram and we revised them with ISD. In total, 11 cases were studied, as shown in Table I.

The experiments were conducted on a DELL PC with 3.4GHz Quad-Core CPU, 16GB RAM and OS of Ubuntu 16.04. The version of SpaceEx is 0.9.8f. The results are shown in Table I. For each of the 11 cases, ISDChecker was able to find a counterexample, whose error type is shown in Column *type*. In total, 3 *race* counterexamples and 8 *timeout* counterexamples are found, which are consistent with the ones identified by manual inspection. This confirms that ISDChecker can effectively find problems in the design models by exhausting the state space. We did not compare our approach with other approaches such as model checking with ITA, since our approach is the only one targeting sequence diagram based models, and moreover, to our best knowledge, there are no tools for model checking interrupt-driven systems available at the time of this writing. For the evaluation of efficiency, we have recorded the sizes of the ISDs and the corresponding IA, and the model transforming time and the checking time. The size of the ISD includes the number of entities (*# entity*), messages (*# msg.*), constraints (*# cons.*), `int` fragments (*# int.*) and different priorities of the `int` fragments (*# prior.*). The size of the IA includes the number of the total locations (*# loc.*), transition (*# trans.*), and variables (*# var.*). The verification time consists of two parts: one from the transformation of models (*transform*), and the other from the execution of the checker (*check*). The cases are arranged from top to bottom in the table in ascending order of first *# int.* and then *#msg.*. The number of `int` fragments in an ISD has a dominant impact on the number of transitions in the corresponding IA, as it is possible for each location transformed from a non-int fragment to have transitions connecting locations transformed from an `int` fragment. The number of messages has a dominant impact on the number of locations, since events and their partial orders, which decides the number of locations, are deduced from messages. As we can see from Table I, the number of each kind of elements in a case's ISD is all below or around 10, which is clearly manageable for human designers, however, a small increase in the scale of an ISD can result in a significant increase in that of the corresponding IA. For example, a moderate sized ISD for the *satellite_spin* example in Fig. 2 has an corresponding automaton with 21 locations and 76 transitions; and with just 1 more `int` fragment and 7 more messages, the *task_rotate* case has a corresponding automaton with 80 locations and 560 transitions. Nevertheless, the model transformation process only took 12-123 ms, which indicates that the cost is almost negligible. By exploiting the advanced hybrid automata checker, the checking time is also

TABLE I
EXPERIMENTAL RESULTS OF VERIFYING INTERRUPT-DRIVEN CASES

case name	type	#entity	# msg.	# cons.	# int.	# prior.	# loc.	# trans.	# var.	transform (ms)	check (s)
ADC_Bug[5]	race	4	5	6	1	1	12	28	10	12	0.05
fridge_controller[23]	timeout	7	8	8	1	1	18	38	12	15	0.40
altitude_display[24]	timeout	6	9	5	1	1	20	36	10	16	0.16
medical_monitor[25]	timeout	7	9	11	1	1	20	44	15	16	3.78
time_sync	race	4	11	4	1	1	30	80	11	73	1.24
orbit_upload	race	5	6	6	2	1	21	82	16	58	19.7
backup_computing	timeout	4	6	7	2	2	16	56	14	62	1.70
system_tick	timeout	4	7	6	2	2	17	56	12	16	1.62
satellite_spin	timeout	4	7	7	2	2	21	76	14	24	5.01
car_controller[2]	timeout	8	11	9	3	2	31	170	18	96	1879
task_rotate	timeout	11	14	7	3	1	80	560	15	123	2751

TABLE II
COMPARISON OF FAMILIARITY DEGREES WITH ISD AND ITA

familiar with	none of SD	simple SD	fragments
none of FSM	$2(2,0,0)$/I	$6(5,1,0)$/II	$5(4,1,0)$/III
FSM	$0(0,0,0)$/IV	$5(2,3,0)$/V	$7(1,5,1)$/VI
HA	$0(0,0,0)$/VII	$0(0,0,0)$/VIII	$1(0,0,1)$/IX

TABLE III
COMPARISON OF USABILITY DEGREES OF ISD AND ITA

Group	I		II		III		V		VI		IX		Total	
no./time(s)	n	t	n	t	n	t	n	t	n	t	n	t	n	t
1st:ISD	0	$*$	2	16	4	12	4	15	5	13	1	10	16	13
1st:ITA	0	$*$	0	$*$	0	$*$	0	$*$	1	47	1	36	2	42
2nd:ISD	1	14	4	13	4	11	3	13	7	12	1	10	20	12
2nd:ITA	0	$*$	0	$*$	0	$*$	2	46	2	41	1	27	5	40

* : the average time is not computed since no subjects have completed the task.

acceptable for taking several seconds for moderate sized cases and 30-40 minutes for large sized ones to exhaust all state space.

From the above discussion, we can derive the answer to RQ1: *Our approach can effectively detect design defects for interrupt-driven systems in an efficient way.*

B. RQ2: Approach Usability

In addressing RQ1, we have seen that the IA transformed from an ISD is much larger. It is natural to think that modelling with ITA would be more difficult, as larger models are often less manageable. In order to have a more convincing result about the usability of our approach, we conducted a controlled experiment. It is known that before users can use formal methods, they must be well trained [26]. The training cost not only affects the users' decision upon whether to use the method but also reflects usability. Thus, one of our goals is to evaluate the training cost of ISD compared with ITA. The other goal is to compare the time costs to correctly develop a specification with ISD and ITA. The subjects of the experiments are students in our department consisting of 14 undergrads, 10 graduate students and 2 PhD students. First, we profiled the subjects for their prior knowledge. We asked the subjects to first answer two questions: Are you familiar with the basic knowledge of sequence diagram (without fragments)/ finite state machines? If any answer is yes, then following up questions are asked: Are you familiar with the concepts of fragments/ hybrid automata? Based on the answers, each subject was assigned to one of the nine groups, as shown in Table II (subjects familiar with fragments are also familiar with simple SD, and those familiar with HA are also familiar with FSM). In each cell, the first number is the total number of subjects, and the three numbers in the parentheses are the number of undergrads, master students and PhD students, respectively. The Roman number after the slash is the label assigned to each group. As shown in Table II, among the 26 subjects, 24 subjects are familiar with sequence diagrams, and 13 subjects have learned the usage of fragments; whereas only half of the subjects know how to model with the finite state machine, and only 1 subject has used hybrid automata.

To evaluate the training cost, we prepared 2 training sessions. Each session lasted 2 hours: the first one hour for ISD and the second one hour for ITA. We were aware that both modelling languages target the interrupt-driven systems and share some common background knowledge, which makes it possible for the language trained later to benefit from the earlier training of the other one. In view of that, we made the schedule to have ISD trained first, to reduce the preferences that the result may have towards ISD. After the first session, we gave the subjects a description of the example in Fig. 2, and asked them to model the system. Again, since it was the same modelling task, we asked the subjects to model with ISD first to reduce the results' preference towards ISD. We carefully manual-checked the models and collected the number of subjects who correctly designed the models (Column *n*, Table III) in each group (Group IV, VII and VIII have no subjects and are not considered). We also recorded the modelling time of the subjects who gave correct models in each group, and present the average time (in seconds) in Table III (Column *t*). After the first training session, 16 subjects produced correct models with ISD, while with ITA only 2 subjects could complete the modelling correctly. Then we gave a more elaborative training session and assigned the subjects a new modelling task of Case *car_control* used in the experiment for RQ1. This time, 20 out of 24 subjects could model with ISD, but still, only 5 subjects could model with ITA. From these data, we can conclude that one needs less training time to master ISD than ITA, to which two factors contribute most. The first one is that most people have experience using sequence diagrams in system modelling, so little training is needed for the basics. The second factor is our ISDs follow the standard sequence diagram notations, and therefore do not require much effort to learn. On the other hand, ITA is a special kind of hybrid automata, which can be difficult to comprehend in a short time. We also compared the modelling time with ISD and ITA. In Table III, the average time of using ISD is about 12 to 13 minutes, while the average time of using ITA needs more than 40 minutes. The reason, as the subjects reported, is that they can design the ISD model by following a frequent scenario, and do not have to worry about the interrupt nesting and unpredictable occurrence.

From the above discussion, we can derive the answer to RQ2: *ISD is easier to learn and use compared to the automata theory based modelling language.*

VI. Related Work

We discuss some closely related work concerning modelling and correctness checking for interrupt-driven systems.

Modelling. Prioritised preemption and interrupt mask register (IMR) are the unique features in interrupt-driven systems, and most modelling approaches attempt to support them. Some of them are variants of flow graphs with extensions to model prioritised preemption and IMR [27], [28], [29]. In [27], a directed graph called interrupt preemption graph is proposed where each edge corresponds to a potential preemption by an interrupt handler, and IMR is exploited to remove unreachable branches in the graph. Time is not considered in these studies, whereas the behaviour of interrupt-driven systems largely depends on the timing properties.

Timed automata [21], as a mature model for specifying timing requirements though, lack the feature of time suspension which is critical to model the executions preempted by interrupts. Hybrid automata, or more specifically the subclasses such as integration automata, stopwatch automata [30] or suspension automata [31] are sufficiently expressive to model time suspension and resumption, but lack the mechanism to model interrupt priority. Therefore, in [10], [11], Interrupt Timed Automata (ITA) is proposed, where the states are organised according to interrupt priorities, ranging from 1 to n, with one active clock that can be suspended for one priority.

Whereas these models are visual and graphical, some others are based on calculus. In [32], the algebra of communicating processes (ACP) is augmented with priorities and non-deterministic choice to describe the working of interrupts. Work [33] studies the "interrupt driven round robin system" where tasks run in round-robin scheduling and interrupt service routines perform urgent actions, and proposes to model the system with a variant of Event B. Work [34] provides a calculus for reasoning about interrupt-driven systems and a type system for checking stack boundedness. Compared with these existing modelling methods using automata theory or calculus, the ISD is more friendly to users. With the `int` fragment, mask variable and task constraint, the unpredictable and prioritised preemptive behaviour and the time suspension and resumption can be easily modelled.

Correctness assurance. Testing has been widely used to create reliable embedded software [35], [5]. In [5], an interrupt scheduler is proposed to fire interrupts at specific time points and prohibit the firings of an interrupt at a time when the system cannot handle it. In [4], test adequacy criteria are introduced to measure the quality of test suites that test interrupt-driven applications. Both studies target nesC applications in TinyOS, as the simple scheduling policy adopted by TinyOS makes the interleaving between tasks more tractable [4]. Nevertheless, interrupt-driven software is still hard to be thoroughly tested since it usually contains a very large number of executable paths.

Different from testing, static analysis and verification of program codes focus on specific code problems, and most of them study the subject of stack size analysis [36], [3], [27]. Data inconsistency is of interest as well, due to the concurrency induced by interrupts. Work [37] analyses data race and transactional behaviour of procedures for interrupt-driven programs synchronised via the priority ceiling protocol. Work [38] proposes to sequentialise interrupt-driven programs into sequential programs, and using existing numerical analysis tools. In [39] nesC programs are transformed to POSIX threads programs for checking race conditions. Timing analysis is also considered, although most studies are restricted to worst-case execution time analysis [2], such as maximum interrupt latency [36]. Static analysis or verification of program codes need to work on specific languages, such as codes for the Z86 architecture [36], [40], [3], or for some Atmel's architectures [27], [29].

Differently, our work chooses the model checking approach. Targeting the design level, models are used to specify the system behaviour and are not bound for the types of implementing languages. When performing checking, the state space of the models is exhausted, guaranteeing the correctness w.r.t. the properties under checking. The closest work to ours is [10], [11], where the interrupt-driven systems are modelled by ITA for checking. As discussed, we propose ISDs for the purpose of easy modelling.

VII. Conclusion

In this paper, we introduce a novel approach to modelling and verifying interrupt-driven systems. We propose the ISD by extending the UML sequence diagram with interrupt fragment, mask variable and task constraint, to model the unpredictable and preemptive system behaviour. Following the automata-based semantics, the ISD can be automatically transformed to IA for checking. Experiments on previous studied cases and real-world aerospace applications consistently confirm our approach's effectiveness. In the usability experiment, user performance shows that both the training cost and usage cost of ISD are lower than the automata-based modelling language ITA. We have implemented the proposed approach as a prototype tool, in the hope that it could become a powerful assistant to system designers.

Acknowledgment

We thank the anonymous reviewers for the valuable comments. This research was supported by the National Key R&D Program of China (No. 2017YFA0700604), National Natural Science Foundation of China (Nos. 61502228, 61632015, 61561146394, 61472180, 61572249) and the Fundamental Research Funds for the Central Universities (020214380045).

References

[1] D. Kroening, L. Liang, T. Melham, P. Schrammel, and M. Tautschnig, "Effective verification of low-level software with nested interrupts," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 229–234.

[2] J. Kotker, D. Sadigh, and S. A. Seshia, "Timing analysis of interrupt-driven programs under context bounds," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc, 2011, pp. 81–90.

[3] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. A. Henzinger, and J. Palsberg, "Stack size analysis for interrupt-driven programs," *Information and Computation*, vol. 194, no. 2, pp. 144 – 174, 2004.

[4] Z. Lai, S. C. Cheung, and W. K. Chan, "Inter-context control-flow and data-flow test adequacy criteria for nesC applications," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2008, pp. 94–104.

[5] J. Regehr, "Random testing of interrupt-driven software," in *Proceedings of the 5th ACM International Conference on Embedded Software*. ACM, 2005, pp. 290–298.

[6] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Trans. Softw. Eng.*, vol. 14, no. 10, pp. 1483–1498, Oct. 1988.

[7] M. J. Harrold and M. L. Soffa, "Efficient computation of interprocedural definition-use chains," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 2, pp. 175–204, Mar. 1994.

[8] Y. Lei and R. H. Carver, "Reachability testing of concurrent programs," *IEEE Trans. Softw. Eng.*, vol. 32, no. 6, pp. 382–403, Jun. 2006.

[9] C.-S. D. Yang, A. L. Souter, and L. L. Pollock, "All-du-path coverage for parallel programs," in *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 1998, pp. 153–162.

[10] B. Bérard, S. Haddad, and M. Sassolas , "Real time properties for interrupt timed automata," in *Proceedings of the 2010 17th International Symposium on Temporal Representation and Reasoning*. IEEE Computer Society, 2010, pp. 69–76.

[11] B. Bérard, S. Haddad, and M. Sassolas, "Interrupt timed automata: Verification and expressiveness," *Form. Methods Syst. Des.*, vol. 40, no. 1, pp. 41–87, Feb. 2012.

[12] F. Cassez and K. G. Larsen, "The impressive power of stopwatches," in *Proceedings of the 11th International Conference on Concurrency Theory*. Springer-Verlag, 2000, pp. 138–152.

[13] OMG, "UML2.0 superstructure specification," *Available at http://www.uml.org*, 2005.

[14] B. Dobing and J. Parsons, "How UML is used," *Commun. ACM*, vol. 49, no. 5, pp. 109–113, May 2006.

[15] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," in *Hybrid Systems*. Springer-Verlag, 1993, pp. 209–229.

[16] Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine, "Integration graphs: A class of decidable hybrid systems," in *Hybrid Systems*. Springer-Verlag, 1993, pp. 179–208.

[17] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *Proceedings of the 23rd International Conference on Computer Aided Verification*. Springer-Verlag, 2011, pp. 379–395.

[18] T. Firley, M. Huhn, K. Diethers, T. Gehrke, and U. Goltz, "Timed sequence diagrams and tool-based analysis: A case study," in *Proceedings of the 2Nd International Conference on The Unified Modeling Language: Beyond the Standard*. Springer-Verlag, 1999, pp. 645–660.

[19] S. Uchitel, J. Kramer, and J. Magee, "Incremental elaboration of scenario-based specifications and behavior models using implied scenarios," *ACM Trans. Softw. Eng. Methodol.*, vol. 13, no. 1, pp. 37–85, Jan. 2004.

[20] A. Knapp and J. Wuttke, "Model checking of UML 2.0 interactions," in *Proceedings of the 2006 International Conference on Models in Software Engineering*. Springer-Verlag, 2006, pp. 42–51.

[21] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, Apr. 1994.

[22] M. Auer, T. Tschurtschenthaler, and S. Biffl, "A flyweight UML modelling tool for software development in heterogeneous environments," in *Proceedings of the 29th Conference on EUROMICRO*. IEEE Computer Society, 2003, pp. 267–272.

[23] F. Pereira, F. Moutinho, and L. Gomes, "Model-checking framework for embedded systems controllers development using IOPT Petri nets," in *2012 IEEE International Symposium on Industrial Electronics*, May 2012, pp. 1399–1404.

[24] C. Fidge and P. Cook, "Model checking interrupt-dependent software," in *Proceedings of the 12th Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2005, pp. 51–58.

[25] L. A. Cortes, P. Eles, and Z. Peng, "Formal coverification of embedded systems using model checking," in *Proceedings of the 26th Euromicro Conference.*, vol. 1, Sept 2000, pp. 106–113 vol.1.

[26] A. Hall, "Seven myths of formal methods," *IEEE Softw.*, vol. 7, no. 5, pp. 11–19, Sep. 1990.

[27] J. Regehr, A. Reid, and K. Webb, "Eliminating stack overflow by abstract interpretation," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 4, pp. 751–778, Nov. 2005.

[28] W. Le, J. Yang, M. L. Soffa, and K. Whitehouse, "Lazy preemption to enable path-based analysis of interrupt-driven code," in *Proceedings of the 2Nd Workshop on Software Engineering for Sensor Network Applications*. ACM, 2011, pp. 43–48.

[29] B. Schlich, T. Noll, J. Brauer, and L. Brutschy, "Reduction of interrupt handler executions for model checking embedded software," in *Proceedings of the 5th International Haifa Verification Conference on Hardware and Software: Verification and Testing*. Springer-Verlag, 2011, pp. 5–20.

[30] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*. ACM, 1995, pp. 373–382.

[31] J. McManis and P. Varaiya, "Suspension automata: A decidable class of hybrid automata," in *Proceedings of the 6th International Conference on Computer Aided Verification*. Springer-Verlag, 1994, pp. 105–117.

[32] J. Bergstra, J. Baeten, and J. W. Klop, "Syntax and defining equations for an interrupt mechanism in process algebra," *Fundamenta informaticae: quarterly*, vol. 9, pp. 127–167, 1986.

[33] B. Stoddart, D. Cansell, and F. Zeyda, "Modelling and proof analysis of interrupt driven scheduling," in *Proceedings of the 7th International Conference on Formal Specification and Development in B*. Springer-Verlag, 2006, pp. 155–170.

[34] J. Palsberg and D. Ma, "A typed interrupt calculus," in *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2*. Springer-Verlag, 2002, pp. 291–310.

[35] B. M. Broekman, *Testing Enbredded Software*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[36] D. Brylow, N. Damgaard, and J. Palsberg, "Static checking of interrupt-driven software," in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 47–56.

[37] M. D. Schwarz, H. Seidl, V. Vojdani, P. Lammich, and M. Müller-Olm, "Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2011, pp. 93–104.

[38] X. Wu, L. Chen, A. Miné, W. Dong, and J. Wang, "Numerical static analysis of interrupt-driven programs via sequentialization," in *Proceedings of the 12th International Conference on Embedded Software*. IEEE Press, 2015, pp. 55–64.

[39] J. Regehr and N. Cooprider, "Interrupt verification via thread verification," *Electron. Notes Theor. Comput. Sci.*, vol. 174, no. 9, pp. 139–150, Jun. 2007.

[40] 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*. ACM, 2003, pp. 198–207.