Investigating Time Properties of Interrupt-Driven Programs

Yanhong Huang[1], Yongxin Zhao[1], Jianqi Shi[1], Huibiao Zhu[1], and Shengchao Qin[2]

[1] Shanghai Key Laboratory of Trustworthy Computing
East China Normal University, Shanghai, P.R. China
{yhhuang,yxzhao,jqshi,hbzhu}@sei.ecnu.edu.cn
[2] School of Computing, Teesside University
S.Qin@tees.ac.uk

Abstract. In design of dependable software for real-time embedded systems, time analysis is an important but challenging problem due in part to the randomicity and nondeterminism of interrupt handling behaviors. Time properties are generally determined by the behavior of the main program and the interrupt handling programs. In this paper, we present a small but expressive language for interrupt-driven programs and propose a timed operational semantics for it which can be used to explore various time properties. A number of algebraic laws for the computation properties that underlie the language are established on top of the proposed operational semantics. We depict a number of important time properties and illustrate them using the operational semantics via a small case study.

Keywords: time, interrupt, operational semantics.

1 Introduction

With the rapid development of the computer industry, multitudinous operating systems spring up in the past forty years. An operating system (OS), as a particular software running on computers, not only manages the computer hardware, but also provides the common platform for efficient execution of various application software. It acts as a bridge between the computer hardware and application programs. A real-time OS is a multitasking OS that aims at executing real-time applications. This kind of OS involves both logical correctness and timeliness. Usually the interrupt mechanism is introduced as a technique to support multi-threads, device drivers and OS in real-time computing, which enables OS to handle time-sharing tasks and concurrency.

An interrupt-driven system indicates that the OS can schedule the tasks' execution and perform reasonable allocation of time and other resources in the form of hardware interrupt or software interrupt. The interrupts are usually implemented in terms of asynchronous signals and synchronous events. The generation of interrupt requests (signals/events) is usually random and nondeterministic,

R. Gheyi and D. Naumann (Eds.): SBMF 2012, LNCS 7498, pp. 131–146, 2012.

which make interrupt behaviors extremely difficult to reason about in the development of OS.

The analysis and verification of interrupt therefore becomes the focus of attention in both industry and academia. There have been proposals suggesting that interrupts can be regarded as threads and may be verified like threads using some similar verification methods [2–5]. Some researchers have attempted to apply different formal methods to the interrupt programs [6–8]. In an earlier work, we have developed a formal model of interrupt programs from a probabilistic perspective, designed the probabilistic operational semantics for interrupt program to capture the potential properties, and specified the time constraint of interrupt programs [1].

Most real-time operating systems require responsive interrupt handling to meet the real-time requirements. As this kind of OS has been widely used in our society, the correctness of timing behavior in this kind of OS becomes increasingly important. There has been work reported on analyzing the time properties of interrupt-driven programs. Jens Palsberg et al. have performed a series of studies on interrupt-driven Z86-based software. They have developed a tool to analyze interrupt latencies, stack sizes, deadline as well as verified fundamental safety and liveness properties [9–12]. John Regehra has proposed a set of design rules for interrupts in real-time and embedded software, where he believes it is necessary to consider the stack overflow, interrupt overload and real-time analysis problems [13].

There has also been work reported to improve the performance of the interrupt mechanism, in order to make real-time embedded operating systems to provide correct and timely services in the presence of constrained resources. Eleidermacher suggests that the most important characteristic that makes an operating system a real-time system is the ability to handle interrupts quickly. He proposes a few rules to minimize interrupt response time in worst case [14]. Jinkyu et al. [15] suggest a novel scheme to minimize the performance degradation in embedded operating systems with real-time support, where they present transparent and selective real-time interrupt services which transparently monitor the system and postpone interrupt handling that are not relevant to real-time tasks.

With the development of various formal methods and emergence of the corresponding tools, such as automata theory, B method, Z notation, CSP, VDM, etc., formal methods can be applied with the assistance of automated and human-assisted tools. This makes the analysis and verification of programs more and more viable. In this paper, we develop a formal model of interrupt-driven programs from a timing perspective, in order to analyze time properties during the development of such programs. We propose an interrupt-driven programming language and define a timed operational semantics for interrupt-driven programs written in this language and explore various time properties using the semantics. The main contributions of our work includes:

- **Interrupt-driven Programs.** We present a language of the interrupt-driven programs including some interrupt operators like enable/disable/set. In our model, the system can enable or disable interrupts to decide whether

the system should enable/disable the interrupt mechanism which is to interact with the environment via interrupt handling. Moreover, the system can request any interrupt itself by setting a interrupt signal which help the system schedules multiple tasks.

- **Timed Operational Semantics.** Time is introduced into operational semantics to specify the meanings of the interrupt-driven programs. We provide two ways to handle the interrupt requests. One is an ordinary way that the received interrupt requests are always handled, and the other is a safe way that the system may ignore some interrupts so as to make sure that the program always meet the deadline. Meanwhile, the algebraic laws [16] that underlie the language are established in terms of the suggested operational semantics.
- **Time Properties.** We depict a number of important time properties which are essential to the real-time embedded operating systems in our framework: interrupt response time, interrupt activated time, interrupt overload and deadline. The analysis of four properties will help the analysis of the real-time embedded OS. Based on these, we give an example to present the feasibility and effectiveness of our approach.

The remainder of the paper is organized as follows: Section 2 introduces the interrupt handling mechanism which we discuss about in this paper and provides an approach to describing the program's operating environment. Section 3 defines the language which has two parts, i.e., the main program and the interrupt handlers. Section 4 is devoted to a timed operational semantics for our interrupt-driven language. Section 5 lists some interesting algebra laws for the computation properties of our programs. The time properties of interrupt programs are specified and a corresponding case study is presented in Section 6, followed by our concluding remarks in Section 7.

2 Overview of the Interrupt Mechanism

In this section, we depict the interrupt mechanism in our model, which has been used in some real-time embedded operating systems. The interrupt mechanism provides an efficient way for an operating system to interact with and react to its operating environment. Such a mechanism is illustrated in Figure 1. In our model, the interrupts are implemented in terms of signals which can be produced by either the software program or the hardware device; in other words, by either the system or the environment. Firstly, when an interrupt is received, the program that is currently running is suspended and its state is saved. Secondly, the code that has already been associated with the interrupt starts to run. Such code can be found in the interrupt vector. At last, the control returns to where the interrupt has occurred and all the preserved states should be returned as if the interrupt has never happened.

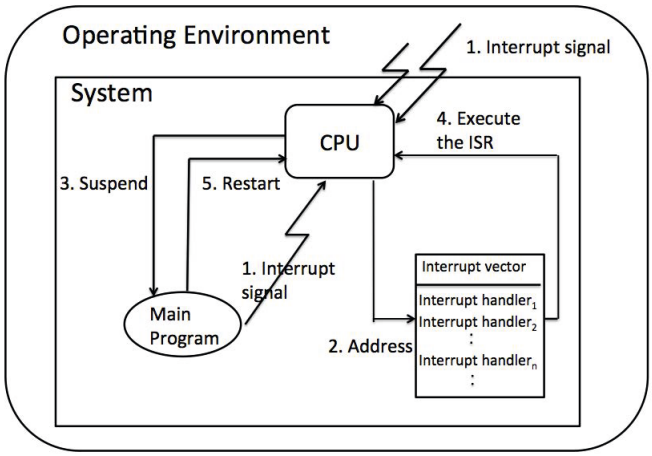

Fig. 1. The Mechanism of Interrupt

Based on the interrupt mechanism, we give an informal introduction about the system and the environment in our model in what follows.

2.1 System

In our model, the system is divided into two parts, i.e., the main program and the interrupt handlers. The main program provides basic services and can be expressed as a particular sequential program. The interrupt handlers interact with the environment to make sure the system can provide correct and timely services. Although they both are programs from a coding perspective, they still have their own characteristics due to different duties.

The main program's characteristics:

- A real-time embedded operating system usually supports multiple tasks, and it always has a scheduling strategy to manage the tasks and to locate the limited resources. In our model, we assume there is one processor and only one task is running at all time. So the system can be described as a sequential program ("the main program" called here).
- The main program can enable or disable the interrupt handling. An interrupt signal is used to denote an interrupt request. Only when the interrupt handling is enabled, the system can receive interrupt signals which are then handled by the corresponding interrupt handlers in the order in which they were received. On the other hand, if the interrupt handling is disabled, the system ignores all interrupt signals received during the disabled period. Moreover, when the interrupt handling option is switched from enabled to disabled, the accumulated interrupt requests are cleared.
- The main program prevents itself from continuously being interrupted by requiring itself to step forward once returning back from an interrupt handler.

The interrupt handlers' characteristics:

- An operating system usually tries to ensure that the time spent on interrupt-handling is kept to a minimum. It saves the state of the interrupted program when the context is switched to that of an interrupt handler. In our model, interrupt handlers may require a little handling time but they will not modify the data states of their interrupted programs. Moreover, for simplicity, we assume no priorities for interrupt handlers here; all interrupt requests are therefore dealt with in the order they were received. It is acceptable for the system to ignore some interrupt signals in order to meet its deadline.
- Both the system and environment can request interrupts by issuing interrupt signals. In our model, the system may issue any interrupt signal in any place wherever it needs. Meanwhile, the environment may also produce any interrupt signal unexpectedly at any time. As we mentioned above, only when the interrupt handling is enabled, interrupt requests can be received and handled as soon as possible (but not necessarily immediately). In our model, we assume that there would not be two or more interrupt signals happened at the same time (such a scenario rarely happens in fact).
- In our model, the system forbids interrupt nesting. But during the execution of an interrupt handler, the system can still receive and record interrupt requests.

2.2 Environment

The correctness of an real-time embedded operating system depends not only on its logical correctness but also its correct response to the operating environment. For better interaction with the environment, the system must respond the interrupts timely on one hand, and the main program of the system must meet its time deadline on the other hand. So analyzing such kind of systems must take into account the variable environment.

In our model, we assume there are only a finite number of different kinds of interrupt signals, and the environment may produce different sequences of interrupt requests made up of these signals. For a given sequence, we can analyze the behavior of the system in the corresponding environment and investigate its time properties. To have a better analysis of the behavior of the system in a specific environment, we assume that each signal in the sequence is labeled with its arrival time. Despite of the randomicity and nondeterminism of the interrupts, we would still expect that any user-given interrupt sequence can reflect the real scenarios to a large extent, so that our analysis can reveal more accurate performance of the system in such reasonable situations.

3 The Language

In this section, we present our language to specify interrupt-driven programs, which includes some ordinary program constructs as well as three new constructs related to interrupt handling, namely enable, disable and set. We also define a function to estimate the execution time of program which supports the time analysis of interrupt-driven programs.

3.1 Syntax

In our model, a system is composed of a main program and a set of interrupt handlers. We use the notation $Sys ::= [M, I]$ to describe a system, where M and I denotes the main program and the set of interrupt handlers respectively. The main program can be interrupted by interrupt signals corresponding to any of the interrupt handlers in the interrupt set I. As mentioned earlier, the interrupt nesting is forbidden in our model. That is, an interrupt handler cannot be interrupted by other handlers. The abstract syntax of the language is defined in the following.

$$M ::= \mathsf{enable} \mid \mathsf{disable} \mid \mathsf{set}(is) \mid P \mid M; M \mid M \lhd b \rhd M \mid [b * M]^n$$

$$(is \rightarrow P) \in I$$

When the system starts, the interrupts are usually enabled so that the system can interact with the environment timely. After that, the main program can enable or disable interrupts. The main program itself can also set an interrupt signal is via $\mathsf{set}(is)$ to take the initiative to request an interrupt. The notion $(is \rightarrow P) \in I$ denotes an interrupt handler P identified by the interrupt signal is. Note that P, which appears in both the main program and interrupt handlers above, stands for an ordinary program and is defined in what follows.

$$P ::= \mathsf{skip} | x := e | P; P | P \lhd b \rhd P | [b * P]^n | \mathsf{atomic}(P)$$

skip is a program that does not change anything. $x := e$ assigns the value of e to the variable x. The programs $P; Q$ denotes the sequential composition of P and Q (similarly for $M_1; M_2$). The program $P \lhd b \rhd Q$ behaves like P if the boolean expression b is true, or Q otherwise (similarly for $M_1 \lhd b \rhd M_2$). The iteration $[b * P]^n$ iterates P whenever b is true. For simplicity, we assume the number of iterations is statically known and is given by the annotation n. The assignment $x := e$ and the evaluation of boolean expression b are atomic, that is, their computation cannot be interrupted. Informally $\mathsf{atomic}(P)$ behaves like P except that any interrupts occurred would not dealt with during the execution. However, it is not equal to $(\mathsf{disable}; P; \mathsf{enable})$. The difference lies in that the system can still receive interrupt requests during the execution of $\mathsf{atomic}(P)$ while $(\mathsf{disable}; P; \mathsf{enable})$ will make the system to ignore all interrupt requests including previously received ones.

3.2 Workload Function

In this subsection, we give a definition of *workload function* $f : M \times \sigma \rightarrow \mathbb{N}$ to estimate the execution time of program. The time that elapses during the program's execution is interrelated with the program's structure and data states. If the initial state of program is definite, then the finial state is definite. M is program while σ stands for the program's current data states. And $\mathbb{N}$ is natural

number denoting time. The user can define the execution time of each program by f, here we assume that the interrupt operations enable, disable and set don't cost any time and all of them can be considered as instantaneous operation following the previous ones, where $f(\mathsf{enable}, \sigma) = 0$, $f(\mathsf{disable}, \sigma) = 0$ and $f(\mathsf{set}(is), \sigma) = 0$. In particular, we define skip won't cost time $f(\mathsf{skip}, \sigma) = 0$. Moreover, we assume the computation of expression e or b won't cost time as they are prepared to evaluate their values in the former operations.

Our group has developed a virtual machine called $xBVM$ which we introduced in [17]. This machine which based on $xBIL$ language can be used to execute the $xBIL$ code and calculate the program's execution time. With this example, we can assure that the workload function is feasible and practical. Absolutely, the user can use any feasible machine to help estimate the program's execution time, which makes better use of our approach for analyzing the time properties of programs.

Property 1. For a sequential program $P; Q$, its execution time is the sum of the time cost by P and Q. The program P starts with initial state σ and Q performs at the state passed from P which is a definite state $P(\sigma)$.

P-1 $f(P; Q, \sigma) = f(P, \sigma) + f(Q, P(\sigma))$

Property 2. This function distributes over conditional operator.

P-2 $f(P \lhd b \rhd Q, \sigma) = f(P, \sigma) \lhd b(\sigma) \rhd f(Q, \sigma)$

Property 3. The $\mathsf{atomic}(P)$ costs the same time as program P.

P-3 $f(\mathsf{atomic}(P), \sigma) = f(P, \sigma)$

Property 4. The execution time of iteration program is interrelated with the boolean expression and loop times.

P-4 $f([b * P]^n, \sigma) = f(P; [b * P]^{n-1}, \sigma) \lhd b(\sigma) \rhd f(\mathsf{skip}, \sigma)$

4 Operational Semantics

In the section, we present an operational semantics for the interrupt-driven program. The operational semantics specifies how the effect of a computation is produced. It is given in terms of transitions between *configuration*. The configuration is defined as a tuple $\langle M, \sigma, t, i, q \rangle$ consisting of the following components:

- M describes the program to be executed. We can use *workload function* to estimate the execution time of this program.
- A state $\sigma \in \Sigma : Vars \to \mathbb{N}$ which is a mapping of the given finite set $Vars$ of variables to the set $\mathbb{N}$ of natural numbers. The data states of the main program and the interrupt handlers have no intersection.
- $t \in \mathbb{N}$ denotes the time spent by the running program.
- The identifer i indicates the running state of the system. It has three values, i.e., 0, 1, and 2. 0 stands for the interrupts are enabled and the main program can be interrupted at any time. 1 denotes that the interrupts are enabled, but the main program cannot be interrupted until progressing one step. And

it makes there won't be more than one interrupt handled in the same place. At last, 2 stands for the interrupts are disabled.
– We employ interrupt signals queue q to record the received signals.

We give a few of the operational rules for the interrupt-driven program as follows. In our framework, each program has a deadline (denoted by d) which is given in specification to help analyze the behavior and time properties of program. For all the transitions, we assume they satisfy a precondition that a good program M to be executed will always meet its deadline, so there exists such a invariance $f(M, \sigma) \leq d$ during the program normally running. Moreover, the system contains a set of interrupt handlers, and the arrow $\rightarrow_I$ means the transition happens within the interrupt set I.

Nontermination
We employ $\langle M, \sigma, t, i, q\rangle$ where $t > d$ to indicate the nontermination of the program. The notation $t > d$ means program M has already missed its deadline.

Assignment

$$\langle x := e, \sigma, t, 0, nil\rangle \rightarrow_I \langle \mathsf{skip}, \sigma[e/x], t', 0, q\rangle$$
$$\langle x := e, \sigma, t, 1, q\rangle \rightarrow_I \langle \mathsf{skip}, \sigma[e/x], t', 0, q'\rangle$$
$$\langle x := e, \sigma, t, 2, nil\rangle \rightarrow_I \langle \mathsf{skip}, \sigma[e/x], t', 2, nil\rangle \quad \text{where } t' = t + f(x := e, \sigma)$$

The assignment $x := e$ is an atomic action which cannot be interrupted. We write $\sigma[e/x]$ for the state that agrees with σ except at x, which is mapped to $\sigma(e)$. The $\sigma(e)$ means the natural number value of e in σ.

There are three kinds of transitions for assignment. Firstly, system is in 0 state, the assignment can execute only when $q = nil$, otherwise the interrupt in q will be handled before the assignment running. During its execution, the environment may produce interrupt signal, where nil may change into q. Secondly, system is in 1 state which means the system returns from handling an interrupt just now, the assignment will always execute no matter whether there is any more interrupt request in q or not. And q may extend to q' as the environment may issue interrupt signal. The assignment is considered as one step, the system can turn its state from 1 to 0 after it executed. At last, when the system is in 2 state, there shouldn't be any interrupt. When the assignment finishes, the execution time t adds the time consumed by itself.

Sequential Composition

$$\frac{\langle M_1, \sigma, t, i, q\rangle \rightarrow_I \langle \mathsf{skip}, \sigma', t', i', q'\rangle}{\langle M_1; M_2, \sigma, t, i, q\rangle \rightarrow_I \langle M_2, \sigma', t', i', q'\rangle}$$

$$\frac{\langle M_1, \sigma, t, i, q\rangle \rightarrow_I \langle M_1', \sigma', t', i', q'\rangle}{\langle M_1; M_2, \sigma, t, i, q\rangle \rightarrow_I \langle M_1'; M_2, \sigma', t', i', q'\rangle}$$

The sequential composition of two programs $M_1; M_2$ is executed by running M_1 first and running M_2 until M_1 terminates. If M_1 is unable to terminate, so is $M_1; M_2$.

Choice

$$\frac{b(\sigma) = \mathsf{true}}{\langle M_1 \lhd b \rhd M_2, \sigma, t, i, q\rangle \rightarrow_I \langle M_1, \sigma, t, i, q\rangle}$$

$$\frac{b(\sigma) = \mathsf{false}}{\langle M_1 \lhd b \rhd M_2, \sigma, t, i, q\rangle \rightarrow_I \langle M_2, \sigma, t, i, q\rangle}$$

The notation $b(\sigma)$ is defined for the boolean value of b in σ, and the evaluation of expression b cannot be interrupted. In our framework, we assume b's computation won't cost time as it is prepared to evaluate true or false in the former operations. The program behaves like M_1 if the boolean expression b is true, or M_2 if false.

Iteration

$$\frac{b(\sigma) = \mathsf{true} \wedge n \geq 1}{\langle [b * M]^n, \sigma, t, i, q\rangle \rightarrow_I \langle M; [b * M]^{n-1}, \sigma, t, i, q\rangle}$$

$$\frac{b(\sigma) = \mathsf{false} \vee n = 0}{\langle [b * M]^n, \sigma, t, i, q\rangle \rightarrow_I \langle \mathsf{skip}, \sigma, t, i, q\rangle}$$

The iteration is similar with the choice program that the interrupt cannot happen during the evaluation of b. The real-time embedded system usually forbids infinite iteration, especially the interrupt handlers, so it is always limited to a number of cycles.

Atomic Action

$$\frac{\begin{array}{c}\langle \mathsf{atomic}(\mathsf{skip}), \sigma, t, i, t\rangle \rightarrow_I \langle \mathsf{skip}, \sigma, t, i, q\rangle \\ \langle P, \sigma, t, 0, nil\rangle \rightarrow_I \langle P', \sigma', t', 0, q\rangle\end{array}}{\langle \mathsf{atomic}(P), \sigma, t, 0, nil\rangle \rightarrow_I \langle \mathsf{atomic}(P'), \sigma', t', 0, q\rangle}$$

$$\frac{\langle P, \sigma, t, 1, q\rangle \rightarrow_I \langle P', \sigma', t', 0, q'\rangle}{\langle \mathsf{atomic}(P), \sigma, t, 1, q\rangle \rightarrow_I \langle \mathsf{atomic}(P'), \sigma', t', 0, q'\rangle}$$

$$\frac{\langle P, \sigma, t, 2, nil\rangle \rightarrow_I \langle P', \sigma', t', 2, nil\rangle}{\langle \mathsf{atomic}(P), \sigma, t, 2, nil\rangle \rightarrow_I \langle \mathsf{atomic}(P'), \sigma', t', 2, nil\rangle}$$

where $t' = t + (f(P, \sigma) - f(P', \sigma'))$

The user can define atomic action to ensure a series of actions complete without interrupted. The behavior of $\mathsf{atomic}(P)$ is the same as program P without interrupt. It has three kinds of transitions like the assignment.

Enable/Disable Interrupt

$$\langle \mathsf{enable}, \sigma, t, i, q\rangle \rightarrow_I \langle \mathsf{skip}, \sigma, t, 0, q\rangle$$

$$\langle \mathsf{disable}, \sigma, t, i, q\rangle \rightarrow_I \langle \mathsf{skip}, \sigma, t, 2, nil\rangle$$

The statement enable is an atomic action which can change the system' state into 0. So that the program can receive the interrupt request and handle the interrupt as soon as possible. The disable is also an atomic action and it makes the system's state turn into 2 and empties the interrupt signals queue q. Once the system is disabled, the received and the new interrupt requests are all be ignored. In our model, we mentioned both of the two operations are considered to take no time.

Request Interrupt

$\langle \mathsf{set}(is), \sigma, t, 0, nil\rangle \rightarrow_I \langle \mathsf{skip}, \sigma, t, 0, is\rangle$

$\langle \mathsf{set}(is), \sigma, t, 1, q\rangle \rightarrow_I \langle \mathsf{skip}, \sigma, t, 0, q^\frown is\rangle$

$\langle \mathsf{set}(is), \sigma, t, 2, nil\rangle \rightarrow_I \langle \mathsf{skip}, \sigma, t, 2, nil\rangle$

The main program can request any interrupt actively. The $\mathsf{set}(is)$ denotes that the main program requests the *is* interrupt. $\mathsf{set}(is)$ is an atomic action and it has the same transitions as assignment except that it won't cost time. Only when the interrupts enabled, the request signal will be accepted and put into the queue in occurred order.

Handle Interrupt

$$\frac{head(q) = is \wedge \langle I(is), \sigma_I, t, 2, nil\rangle \rightarrow_I \langle \mathsf{skip}, \sigma_I', t', 2, nil\rangle}{\langle M, \sigma, t, 0, q\rangle \rightarrow_I \langle M, \sigma, t', 1, q'\rangle} \quad \text{where } t' = t + f(I(is), \sigma_I)$$

As we mentioned before, there is no intersection of data states between the main program and interrupt handlers, where $\sigma \cap \sigma_I = \emptyset$. When the interrupts can be executed, they are always handled in *First In First Out* order. So the head signal is got out of queue and the corresponding interrupt handler executes. The interrupt handler cannot be interrupted, so its state is set to 2 and q is nil.

After the interrupt terminates, the time t of the main program should also records the time spent by the interrupt. And the system'state changes from 0 to 1 to denote the main program cannot be interrupted again in the same place. The environment may produce interrupt signal during the execution of interrupt handler, so $q'{=}is^\frown q$ or $q'{=}is^\frown q^\frown q''$. If the interrupt consumes so much time that make the main program miss the deadline, the deadline is negative.

Handle Interrupt Safely

In our model, we also provides a mechanism that can make the program always meet the deadline. Before the ready interrupt handler running, the system will evaluate whether there is enough time for the main program's execution. We use the arrow $\xrightarrow{s}_I$ to denote the transition is based on the safe mechanism.

$$\frac{head(q) = is \wedge f(I(is), \sigma_I) \leq T \wedge \langle I(is), \sigma_I, t, 2, nil\rangle \xrightarrow{s}_I \langle \mathsf{skip}, \sigma_I', t', 2, nil\rangle}{\langle M, \sigma, t, 0, q\rangle \xrightarrow{s}_I \langle M, \sigma, t', 1, q'\rangle} \quad \text{where } T = d - f(M, \sigma) \text{ and } t' = t + f(I(is), \sigma_I)$$

$$\frac{head(q) = is \wedge f(I(is), \sigma_I) > T}{\langle M, \sigma, t, 0, q\rangle \xrightarrow{s}_I \langle M, \sigma, t, 0, q'\rangle} \quad \text{where } T = d - f(M, \sigma)$$

The safe transition is the same as the normal transition when there is enough time for the interrupt to be executed, in other words, the inequality $f(I(is), \sigma_I) \leq (d - f(M, \sigma))$ establishes. If there isn't enough time and the system adopts safe transition, the system may reject the interrupt request to make the main program meet its deadline. The states of the system remain the same except removing the ignored interrupt signal out of q.

We present two equivalence relations in our framework like bisimulation [18, 19]. We assume two programs M_1 and M_2 with the same interrupt set I execute in the same environment, and they have same initial states except deadline. Formally, let $\rightarrow_I^1$ mean one step and $\rightarrow_I^*$ mean 0 or more steps under the operational semantics.

Definition 1. We say an equivalence relation $=_t$ over the interrupt-driven programs. $M_1 =_t M_2$ iff for any state σ,

$$\begin{array}{ll} \text{if} & \langle M_1, \sigma, t_1, 2, nil\rangle \rightarrow_I^* \langle \mathsf{skip}, \sigma_1', t_1', 2, nil\rangle \text{ and} \\ & \langle M_2, \sigma, t_2, 2, nil\rangle \rightarrow_I^* \langle \mathsf{skip}, \sigma_2', t_2', 2, nil\rangle \\ \text{then} & (\sigma_1' = \sigma_2') \wedge (t_1' - t_1 = t_2' - t_2) \end{array}$$

When discussing about $=_t$-equivalence, we assume the interrupt mechanism is disabled to analyze the main program's own behavior. When the two programs terminate, their data states are still same and they consume same execution time.

Definition 2. We define an equivalence relation $\mathcal{R}$ over configurations as a $_I$-bisimulation if $\langle M_1, \sigma, t, i, q\rangle \mathcal{R}\ \langle M_2, \sigma, t, i, q\rangle$ implies,

$$\begin{array}{ll} \text{if} & \langle M_1, \sigma, t, i, q\rangle \rightarrow_I^1 \langle M_1', \sigma_1', t_1', i_1', q_1'\rangle \\ \text{then} & \langle M_2, \sigma, t, i, q\rangle \rightarrow_I^* \langle M_2', \sigma_2', t_2', i_2', q_2'\rangle \text{ and} \\ & \langle M_1', \sigma_1', t_1', i_1', q_1'\rangle \mathcal{R} \langle M_2', \sigma_2', t_2', i_2', q_2'\rangle \\ \text{if} & \langle M_2, \sigma, t, i, q\rangle \rightarrow_I^1 \langle M_2', \sigma_2', t_2', i_2', q_2'\rangle \\ \text{then} & \langle M_1, \sigma, t, i, q\rangle \rightarrow_I^* \langle M_1', \sigma_1', t_1', i_1', q_1'\rangle \text{ and} \\ & \langle M_1', \sigma_1', t_1', i_1', q_1'\rangle \mathcal{R} \langle M_2', \sigma_2', t_2', i_2', q_2'\rangle \end{array}$$

Definition 3. (a) Two configurations C_1 and C_2 are $_I$-bisimilar, written as $C_1 =_I C_2$, if there exists a $_I$-bisimulation $\mathcal{R}$ such that $C_1 \mathcal{R} C_2$. **(b)** Two programs M_1 and M_2 are $_I$-bisimilar, denoted as $M_1 =_I M_2$, if for any states σ, t, i and q, $\langle M_1, \sigma, t, i, q\rangle \mathcal{R} \langle M_2, \sigma, t, i, q\rangle$.

According to the definition of operational semantics and the two kinds of equivalences, we can educe if $=_I$-bisimilar establishes, then $=_t$-equivalence must establish. Define if $M_1 =_I M_2$ then $M_1 =_t M_2$.

Example. $(x := x) \neq_t \mathsf{skip}$ since they may have different execution time. Moreover, $(\mathsf{disable}\ x := 1\ \mathsf{enable}) =_t (x := 1)$ but $(\mathsf{disable}\ x := 1\ \mathsf{enable}) \neq_I (x := 1)$. Although they cost same time, the latter can receive interrupt quest. So they may have different system state and interrupt queue.

5 Algebraic Laws

Program properties can be expressed as algebraic laws (equations or inequations), which can be verified by using the formalized semantics. We explore a set of important and useful algebraic laws which hold for the interrupt-driven program in this section. Proofs that the laws are sound with respect to the operational semantics are straightforward and have been omitted due to space limit.

Algebra is well-suited for direct use by engineers in symbolic calculation of parameters and the structure of an optimal design. Algebraic proof by term rewriting is the most promising way in which computers can assist in the process of reliable design [16]. From the point of view of language design it is desirable to impose as few constraints as possible on the programming constructs, and make the laws as widely applicable as possible. Here we will confine ourselves to those laws involving the introduced operators.

Atomic Statement

Atomic statement is idempotent.

A-1 $\mathsf{atomic}(P) =_I \mathsf{atomic}^2(P)$.

Atomic statement distributes over conditional choice.

A-2 $\mathsf{atomic}(P \lhd b \rhd Q) =_I \mathsf{atomic}(P) \lhd b \rhd \mathsf{atomic}(Q)$.

Interrupt Operation

The programs enable and disable are idempotent.

I-1 $\mathsf{enable}; P =_I \mathsf{enable}; \mathsf{enable}; P$

I-2 $\mathsf{disable}; P =_I \mathsf{disable}; \mathsf{disable}; P$

Program disable makes the following program $\mathsf{set}(is)$ no sense.

I-3 $\mathsf{disable}; \mathsf{set}(is) =_I \mathsf{disable}$

The atomic operator between disable and enable behaves no sense.

I-4 $\mathsf{disable}; \mathsf{atomic}(P); \mathsf{enable} =_I \mathsf{disable}; P; \mathsf{enable}$.

6 Time Properties and a Case Study

Real-time embedded operating systems support real-time applications; therefore, the designers of such systems should consider their real-time features. The correctness of this kind of operating systems involves both the logical correctness and timeliness. In this section, we will analyze some time properties about our interrupt-driven programs listed as below.

Interrupt Response Time. The system usually takes time to respond to an interrupt request. This is the time between the arrival of the interrupt signal and the start of the execution of the corresponding interrupt handler. In order for the system to have better interaction with the environment, the system should handle the interrupt request timely. So we think the analysis of interrupt-driven programs should take this property into account. The system requirement usually give worst-case upper bounds on interrupt response time. In our model, we use σ_w to denote the worst case response time of the analyzed program.

Interrupt-Activated Time. This time denotes the total period when the interrupts are enabled. The system allows the main program to enable or disable interrupts. The system requirement usually extends interrupt activated time to make sure that the system can interact with the environment promptly. So it is

necessary to consider the interrupt activated time in a specified environment to evaluate the system's real-time behaviors.

Interrupt Overload. If there come too many interrupt requests, the system may not have enough time to handle all of them, or it may miss its deadline. Interrupt overload is a problem to consider in real-time analysis. The designer usually adopts a few strategies to help make interrupt overload less likely or impossible in real-time embedded operating systems. For example, they may keep the execution time of interrupt handlers short or bound the arrival rates of interrupt signals. Given its importance, we shall also analyze this property of the system in a given environment.

Deadline. A reliable real-time embedded operating system should be such a system that it meets its deadline in most cases but may miss its deadline very rarely (the probability for such cases is so low that it can be tolerated). There may be a deadline for every program, but we assume the interrupt handler always meet its deadline in our model. Here we just consider whether the main program meet the deadline or not in a variable environment.

The Case Study

To carry out the study, we can define a specific environment which contains a sequence of interrupt signals to happen as well as their happening time. For instance, the user can assume that there are three kinds of interrupts in the system, denoted as is_1, is_2, and is_3. For convenience, we assume each of the interrupt handlers costs the same time in different data states, and we assume that $f(I(is_1), \sigma_I) = \tau$, $f(I(is_2), \sigma_I) = 2\tau$ and $f(I(is_3), \sigma_I) = 4\tau$, where τ indicates the time unit. We also assume that the environment may produce a sequence such as $\langle is_1^{2\tau}, is_1^{3\tau}, is_2^{4\tau} \rangle$ or another sequence: $\langle is_1^{\tau}, is_2^{3\tau}, is_3^{5\tau} \rangle$, e.g., $is_1^{2\tau}$ means at the second time unit, the environment will produce the interrupt signal is_1. We can analyze the behavior and the time properties about the system in such given environments.

A main program P whose deadline is 10τ is defined as below, and meanwhile the interrupts are enabled at the very beginning. We define the execution time for the program begins in a unique initial data state σ by workload function f. Here we assume $\sigma = \sigma_w$ that the analyzed program is running in the worst case scenario. We use superscripts like (n) to label each operation for simplicity.

$$
\begin{aligned}
P =_{df}\ & x := 1^{(1)}; y := 2^{(2)}; \\
& \mathsf{atomic}(x := x + 1; y := y + 1)^{(3)}; \\
& z := x + y^{(4)}; \mathsf{disable}^{(5)}; x := x - 2^{(6)};
\end{aligned}
$$

$f(x := 1, \sigma) = \tau$
$f(y := 2, \sigma_1) = \tau$ where $\sigma_1 = x := 1(\sigma)$
$f(x := x + 1, \sigma_2) = \tau$ where $\sigma_2 = y := 2(\sigma_1)$
$f(y := y + 1, \sigma_3) = \tau$ where $\sigma_3 = x := x + 1(\sigma_2)$
$f(z := x + y, \sigma_4) = \tau$ where $\sigma_4 = y := y + 1(\sigma_3)$
$f(x := x - 2, \sigma_5) = \tau$ where $\sigma_5 = z := x + y; disable(\sigma_4)$

Case 1: The program P executes in such an interrupt sequence: $\langle is_1^{2\tau}, is_1^{3\tau}, is_2^{4\tau} \rangle$. The table below has four parts: P presents the last executed program, t denotes the execution time of program, i indicates the system's state, and q certainly denotes the interrupt signal queue. Take the row 2 for example, when $x := 1$ finishes running, it costs one time unit τ. The system state is 0 and the interrupt queue has no interrupt signal.

Table 1. Steps shown in the case 1

	P	t	i	q
1		0	0	nil
2	1	τ	0	nil
3	2	2τ	0	is_1
4	$I(is_1)$	3τ	1	is_1
5	3	5τ	0	$is_1 \widehat{\ } is_2$
6	$I(is_1)$	6τ	1	is_2
7	4,5	7τ	2	nil
8	6	8τ	2	nil

We analyze four properties we mentioned above in case 1. The average of the interrupt response time is τ, where $(0 + 2\tau)/2 = \tau$ that the first two interrupt requests are handled. The interrupt activated time is the sum of time when the system in 0 and 1 states, and it is 6τ in case 1. During the whole execution of program P, there comes three interrupts but only two are handled before it finishes. At last, $t = 8\tau < d$ indicates P meet its deadline.

Case 2a: The program P executes in another interrupt sequence: $\langle is_1^{\tau}, is_2^{3\tau}, is_3^{5\tau} \rangle$.

Table 2. Steps shown in the case 2a

	P	t	i	q
1		0	0	nil
2	1	τ	0	is_1
3	$I(is_1)$	2τ	1	nil
4	2	3τ	0	is_2
5	$I(is_2)$	5τ	1	is_3
6	3	7τ	0	is_3
7	$I(is_3)$	11τ	1	nil

Case 2b: According to Case 2a, we get program P miss the deadline where $t = 11\tau > d$ at last. Here we analyze the behavior of the system by following the safe interrupt handler transition rules. In this case, the is_3 interrupt which costs so much time won't be handled so P would meet the deadline.

We compare the performance of program P in the same environment but following different transition rules, namely, an ordinary rule in case 2a, and a safe rule in case 2b. The interrupt response time is respectively $2/3\tau$ and 0.

Table 3. Steps shown in the case 2b

	P	d	i	q
1		0	0	nil
2	1	τ	0	is_1
3	$I(is_1)$	2τ	1	nil
4	2	3τ	0	is_2
5	$I(is_2)$	5τ	1	is_3
6	3	7τ	0	is_3
7	4,5	8τ	2	nil
8	6	9τ	2	nil

The interrupt activated time is respectively 11τ and 7τ. Three interrupts are all handled in case 2a, while only the first two interrupts are handled in case 2b. Due to the is_3 interrupt executes or not, P misses the deadline in case 2a, but meets the deadline in case 2b.

According to the examples above, it's convenient for the user to analyze the time properties of the interrupt-driven program through the language and the timed operational semantics in our framework.

7 Conclusion and Future Work

It remains a challenging problem to analyze time properties for programs in the presence of interrupts. In this paper we make a small step forward in tackling this problem. We provide a small interrupt-driven programming language and propose a timed operational semantics for it. To simplify the analysis, we consider only finite programs where the number of iterations are statically known. We make use of a workload function to estimate the execution time of programs. We also have some preliminary discussions on algebraic laws based on the proposed operational semantics. Several time properties for such programs are introduced and analyzed under several scenarios with the help of the operational semantics.

As for future work, we will extend the model to cover more advanced issues, such as interrupt priorities, interrupt nesting, enabling/disabling some interrupt services (but not all). We shall also try to provide a formal specification for the time properties and offer a more formal analysis as to how good a programming model would behave in terms of those time properties.

Acknowledgment. This work is supported by National Basic Research Program of China (No. 2011CB302904), China Core Electronic Components, High-end Universal Chips and Infrastructure Software series Significant Project (No. 2009ZX01038-001-07), National High Technology Research and Development Program of China (No. 2011AA010101 and No. 2012AA011205), National Natural Science Foundation of China (No. 61061130541 and No. 61021004), Shanghai Leading Academic Discipline Project (No. B412), and East China Normal University Overseas Research Foundation (No. 79622040).

References

1. Zhao, Y., Huang, Y., He, J., Liu, S.: Formal Model of Interrupt Program from a Probabilistic Perspective. In: ICECCS (2011)
2. Regehra, J., Coopridera, N.: Interrupt Verification via Thread Verification. ENTCS (2007)
3. Regehra, J.: Random testing of interrupt-driven software. ACM (2005)
4. Kleiman, S., Eykholt, J.: Interrupts as threads. In: ACM SIGOPS (1995)
5. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads. J. Autom. Reasoning 42(2-4) (2009)
6. Hills, T.: Structured interrupts. In: ACM SIGOPS (1993)
7. Cobben, T., Engels, A.: Interrupt and Disrupt in MSC: Possibilities and Problems (1998)
8. Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: Syntax and Defining equations for an interrupt mechanism in process algebra. In: FIIX (1986)
9. Brylow, D., Damgaard, N., Palsberg, J.: Static Checking of Interrupt-driven Software. In: ICSE (2001)
10. Palsberg, J., Ma, D.: A Typed Interrupt Calculus. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 291–310. Springer, Heidelberg (2002)
11. Chatterjee, K., Ma, D., Majumdar, R., Zhao, T., Henzinger, T.A., Palsberg, J.: Stack Size Analysis for Interrupt-driven Programs. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 109–126. Springer, Heidelberg (2003)
12. Brylow, D., Palsberg, J.: Deadline Analysis of Interrupt-Driven Software. IEEE Transactions on Software Engineering (2004)
13. Regehra, J.: Handbook of Real-Time and Embedded Systems. In: Safe and Structured Use of Interrupts in Real-Time and Embedded Software (2007)
14. Kleidermacher, D.: Minimizing Interrupt Response Time, Design Strategies and Methodologies 4(1) (2005)
15. Jeong, J., Seo, E., Kim, D.-S., Kim, J.-S., Lee, J., Jung, Y.-J., Kim, D.-H., Kim, K.: Transparent and Selective Real-Time Interrupt Services for Performance Improvement. In: Obermaisser, R., Nah, Y., Puschner, P., Rammig, F.J. (eds.) SEUS 2007. LNCS, vol. 4761, pp. 283–292. Springer, Heidelberg (2007)
16. Hoare, C.A.R., He, J.: Unifying Theories of Programming, Prentice Hall International Series in Computer Science (1998)
17. Shi, J., Zhu, L., Huang, Y., Guo, J., Zhu, H., Fang, H., Ye, X.: Binary Code Level Verfication for Interrupt Safety Properties of Real-Time Operating System. In: TASE (2012)
18. Milner, R.: Communication and Comcurrency. Prentice Hall International Series in Computer Science (1990)
19. Milner, R.: Communication and Mobile System: $\pi-$ caculus. Cambirdge University Press (1999)