A Genetic Algorithm Based Approach for Event Synchronization Analysis in Real-time Embedded Systems

Yan Chen¹, Yann-Hang Lee²*
¹The School of Information Science and Technology, Xiamen University, China.
²Computer Science and Engineering Department Arizona State University, U.S.A.

Abstract
In real-time embedded systems, due to race conditions, synchronization order between events may be different from one execution to another. This behavior is permissible as in concurrent systems, but should be fully analyzed to ensure the correctness of the system. In this paper, a new intelligent method is presented to analyze event synchronization sequence in embedded systems. Our goal is to identify the feasible sequence, and to determine timing parameters that lead to these sequences. Our approach adopts timed event automata (TEA) to model the targeted embedded system and use a race condition graph (RCG) to specify event synchronization sequence (SYN-Spec). A genetic algorithm working with simulation is used to analyze the timing parameters in the target model and to verify whether a defined SYN-Spec is satisfied or not. A case study shows that the method proposed is able to find potential execution sequences according to the event synchronization orders.

Keywords – embedded system, event, synchronization sequence, timed event model, race condition, genetic algorithm.

1 Introduction
Real-time embedded systems are computing systems that must react within precise time constraints to current events in the application environment. A reaction that occurs late could not only be useless but also catastrophic. A real-time system must have some notion of time. The time base can be absolute, corresponding to a physical clock, or relative, based on specific events. A synchronization primitive can be implemented to establish and maintain ordered execution between computational tasks [1]. However, because of race conditions [2], which are caused due to the nondeterminism in the inter-tasks communication and synchronization mechanisms, the order that the synchronization operations take place may be different from one execution to another. This can result in different order of concurrent operations which may lead to incorrect behavior and output. Even if the orders of concurrent operations are permissible, they should be analyzed thoroughly to ensure the correctness of the systems [1].

Concurrent tasks in real-time embedded systems usually communicate through message queues or shared variables. Enforced by RTOS, the operations on message queues are atomic. Similarly the accesses to shared variables should be guarded by semaphores. However, multiple accesses to message queues or shared variables may be interleaved in arbitrary orders. This leads to the so-called message races [3] and semaphore races [4]. Figure 1. shows an example of a semaphore race where tasks 1 and 2 can take the semaphore in distinct orders in different execution scenarios.

In the past two decades, many static and dynamic approaches [3], [4], [5], [6] are proposed to detect race conditions in concurrent programs. However, the problem of detecting all feasible race conditions is NP-hard in general cases [2]. Due to the scheduling and timing characteristics in real-time embedded systems, many race conditions detected by these approaches may be infeasible in practice. For instance, in the example of Figure 1., if we assume the system uses priority-based preemption scheduling algorithm, the priority of Task 1 is higher than Task 2 and they are released at the same instant, then the execution sequence (1) would always happen, while the execution sequence (2) would never happen. So, in order to ensure the correctness of real-time embedded systems, it is necessary to know exactly whether a race condition is feasible or not, i.e., whether a specific case of synchronization order of some potential events can happen in an execution. If so, it is necessary to find a corresponding execution sequence which satisfies the case for understanding, analysis and testing.

To find the possible execution sequences, we can consider a parametric analysis about the timing instants that tasks are released and external events occur. Such an analysis is not only time consuming, but also have to face the exponential increase of state spaces in the execution model. In this paper, a new method, based on a genetic algorithm, is presented to analyze event synchronization in real-time embedded systems. There are several contributions of this method:

1. Timed event automata (TEA) are presented to describe the timing behavior of a target real-time system, and each state in the TEA is a synchronization event.
2. A race condition graph (RCG) [17] is used to specify the synchronization order of events which have races.
3. A genetic algorithm (GA) working with a simulation-based approach [14], [15] is used to verify whether the
events synchronization order specified (SYN-Spec) is satisfied or not by the target TEA; if it is satisfied, a corresponding execution sequence is given.

Binary Semaphore $s$ = available;

Task 1
.....
semTake(s);
//Access the critical resource
semGive(s);
.....

Task 2
.....
semTake(s);
//Access the critical resource
semGive(s);
.....

(a) A program with two tasks

(b) Two execution sequences with different event synchronization orders

Figure 1. An example of semaphore race

The rest of the paper is organized as follows. In Section 2, some related works are given. Section 3 presents the real-time system modeling and simulation with TEA. Section 4 describes how to use RCG to specify the events synchronization order. In Section 5, a GA is presented to work with simulation approach to analyze the timing parameters. Then, a case study of real-time dining philosopher program is shown up, followed by a conclusion and the future work.

2 Related Works

A well-known approach which can be used to analyze and verify the real-time embedded systems is model checking technology. In this approach, a system is represented by a kind of formal model based on finite state machine, such as timed automata [7], timed I/O automata [8], and linear hybrid automata [9] and so on. Properties required are specified by a kind of temporal logic language, such as LTL, CTL, and TCTL [10], or even by another formal model. The algorithms for model checking are typically based on an exhaustive state search model of the target system: for each state of the model it is checked whether it behaves correctly, that is, whether the state satisfies the desired specification. In its most simple form, this technique is known as reachability analysis. The disadvantage of model checking technology is the state explosion problem. So, its capacity is restricted by the huge program state spaces.

Perry [11] described a system that automatically detects races in a parallel program. In this approach, the dynamic execution trace of the program was used to build a task graph and logged points of event style synchronization. David [12] developed a static analysis method for determining which dependences in a program cannot possibly result in a data race because of schedule restrictions enforced by event posts and waits. Fumihiko [13] presented a new parallel computational model, the LogGPS model, which was useful to analyze synchronization costs of parallel programs that used message passing. All these approaches do not consider scheduling and timing, so they cannot be used for checking the feasibility of a race in real-time system.

Johan and Anders [14], [15] presented a simulation-based approach to do the impact analysis of real-time system. In their approach, a simulation model of a target system, which is described by ART-ML, is extracted from both static and dynamic analysis. Though it was only used for timing analysis, the simulation-based approach can be used to do much further analysis of real-time systems.

In our previous work, we have proposed a technology based on model checking to verify race conditions in real-time embedded systems [16], and a race condition graph to analyze the concurrent program behavior is presented in [17]. In this paper, we propose a new method to analyze the event synchronization order based on simulation-based technology according to race conditions in the target systems.

3 Embedded System Modeling And Simulation

The analysis approach for event synchronization order is shown in Figure 2. There are two inputs to the analysis: a target system which is represented as TEA and a specific synchronization order SYN-Spec which is described by RCG. Then, an execution algorithm is used to simulate the target TEA based on a priority-based preemptive scheduling algorithm, inter-task synchronization and communication constructs and virtual clock. At last, GA working with simulation is used to analyze the timing parameters in the TEA to verify whether the SYN-Spec is satisfied or not. If the SYN-Spec is satisfied, a corresponding execution sequence is obtained.

3.1 Syntax and Semantics of TEA

A real-time system is composed of multiple concurrent tasks with a scheduling algorithm to control
CPU resource. These tasks synchronize and communicate with each other by using message passing, shared variables, etc. In this paper, timed event automata (TEA) are used to describe the timing behaviors of a real-time system. It includes synchronization events (SYN-Event), such as sending message (MS), receiving message (MR), taking semaphore (ST), giving semaphore (SG) and task delay (TD) and so on.

**Definition 1: SYN-Event.** Each SYN-Event e in a real-time embedded system is defined as a 5-tuple $e := p, \lambda, o, x, i >$, where $p$ is the calling task; $\lambda$ maps $e$ into one of the following types: MS, MR, ST, SG, TD; $o$ is the operating object of $e$, which may be a message queue, a semaphore or even null; $x$ is the related data of $e$, which may be a string, an integer, a parameter or even null; $i$ is an unique identifier of $e$. The relation of $\lambda, o$ and $x$ is shown in TABLE I.

<table>
<thead>
<tr>
<th>Event type ($\lambda$)</th>
<th>Operating Object ($o$)</th>
<th>Related Variable ($x$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS</td>
<td>Message Queue</td>
<td>Data Sent</td>
</tr>
<tr>
<td>MR</td>
<td>Message Queue</td>
<td>Data Received</td>
</tr>
<tr>
<td>SG</td>
<td>Semaphore</td>
<td>Null</td>
</tr>
<tr>
<td>ST</td>
<td>Semaphore</td>
<td>Null</td>
</tr>
<tr>
<td>TD</td>
<td>Null</td>
<td>Delay Time</td>
</tr>
</tbody>
</table>

In this paper, we focus on the synchronization and communication constructs with asynchronous message passing, binary semaphore and counting semaphore, but other synchronization events can also be added according to the requirement of cases.

**Definition 2: TEA.** A real-time embedded system is described as a TEA which is a 7-tuple: $S := \langle P, X, E, A, \tau, \mu, \gamma \rangle$ with the following restrictions. $P$ is a set of tasks; $\forall p \in P$ is a periodic or an aperiodic task, including task name, task ID, task priority, and entry and exit nodes. $X$ is a set of variables which may be integer variables ($int$), timing parameters ($para$), message queues ($mq$), binary semaphores ($bs$) or counting semaphores ($cs$). $E$ is a set of SYN-Events. $A \subseteq E \times E$ is a set of transitions. $\forall a(e_i, e_j) \in A$, such that

- $\tau(a)$ is a timing predicate describing the effective execution time and timing constraint from a source event $e_i$ to a target event $e_j$ of $a$, and $\tau (a) := [x] | [y, z] | [x] | [y, \infty]$, where $x, y, z \in R^\ast$ and $x, y, z \geq y$, $[x]$ is the effective execution time, and $[y, z]$ is a timing constraint.
- $\mu(a)$ is a local predicate describing the triggering condition of $a$, and $\mu(a):= true | (x - n) | (x - y) | \neg \mu | (\mu \land \mu \land \mu)$, where $x$ and $y$ are variables; $n \in R^\ast$; $\neg \in \{\leq, <, =, \geq, >\}$; $\neg$ and $\land$ are Boolean negation and disjunction respectively. If $\mu(a):= true$, it is usually omitted.
- $\gamma(a)$ is a set of assignments. If $\tau(a) \land \mu(a) = true$, the transition $a$ is triggered, and then $\gamma(a)$ is executed. The syntax of $\gamma(a)$ is: $\gamma(a) := L := \sigma | (\gamma_1 \lor \gamma_2)$, and $L := x$ and $R := n \mid (x \sim y) | (x - n)$, where $x$ and $y$ are variables, $n \in R^\ast$.

### 3.2 Simulating the TEA

The simulator engine is based on four parts: scheduler, synchronization & communication constructs, virtual clock and execution algorithm. Let $S := \langle P, X, E, A, \tau, \mu, \gamma \rangle$ be a TEA, $\forall p \in P$ has three states: ready, run, wait. A priority-based preemptive scheduling algorithm [20] is used to schedule the tasks and control the states transitions of tasks. With the synchronization & communication constructs which include message-passing, semaphore and task delay, $\forall e \in E$ is processed according to the characteristic of each type of SYN-Events during an execution.

For timing-accurate simulation, TEA uses timing predicate $\tau$ to describe the amount of CPU time required by a task to execute from one SYN-Event to the next, i.e. the execution time of the code between these model events. In this simulation approach, a virtual clock $C$ is used to calculate the execution time of TEA. When an execution is started, $C$ is increased steadily until the execution is ended. Let $\delta(a_e, e_{i+1}) \in A$ be a transition, and $\tau(a_e) = [t_a(e)], [min(a_e)], [max(a_e)]$ be a timing predicate of $a_e$, then the algorithm for time consume from $e_i$ to $e_{i+1}$ is shown in Figure 3. In this algorithm, the function `getGlobalClockValue()` is used to read the value of virtual clock $C$.

![Figure 3. An algorithm for time consume](image)

In a simulation, all tasks in TEA are concurrent tasks and the execution algorithm of each task is shown in Figure 4. In this algorithm, statement (2) means a task executes from a “start” node; statement (3) searches for the next transition $a$ according to $e_i$ and the corresponding local predicates; statement (6) calls `timeConsume()` function to consume execution time from current event to the next event; statement (7) is used to do a set of assignments; statement (8) gets the next event and statement (9) uses `executeEvent()` to execute the event according to the event type. Note that an executing task may be preempted at any time by the other ready task with higher priority.
Let $t(a_i)$ be the total time spent on the transition $a_i$, we have $t(a_i) = t_1(a_i) + t_2(a_i)$, such that:

- $t_1(a_i)$ is the effective execution time of $a_i$. It is the time spent executing run-time or system services on its behalf, without being pended or delayed.
- $t_2(a_i)$ is the sleeping time of $a_i$. It is caused by a task delay event $TD$.
- $t_3(a_i)$ is the time spent by $a_i$ blocked due to the unavailability of a resource, such CPU, semaphore, or message.

Therefore, a feasible execution of TEA should satisfy the following inequality: $\forall a_i \in A, \min(a_i) \leq t(a_i) \leq \max(a_i)$.

Figure 5. (a) and (b) show a simple example of TEA. It includes two tasks T1 and T2, and the priority of T1 is higher than T2. In these two tasks, there are timing constraints between $e_i$ and $e_j$, $x$ and $y$ are two timing parameters.

4 Specification Of Event Synchronization Order

An execution of TEA exercises a sequence of synchronization events. Let $S \leftarrow P, X, E, A, \tau, \mu, \gamma >$ be TEA, an execution sequence generated by a simulation is denoted as: $Q = \{\langle e, t \rangle | t \in R \}$ is the happened time of $e, e \in E \}$. There are two characteristics of $Q$:

- $Q$ satisfies happened-before relation, denoted as $\langle \overrightarrow{\text{HB}} \rangle$, which is a partial order over the SYN-Events and shows the sequence of events that potentially affect one another [19].
- $Q$ has a synchronization order, which is a total order over all of the SYN-Events of an execution.

With a postmortem approach [3], an execution sequence is analyzed to detect race conditions. In this section, $RCG$ is used to specify the synchronization order of SYN-Events which have races.

4.1 Race Set

Let $Q$ be an execution sequence. A race set of $Q$ is given as a triple: $RS = \langle E_R, \overrightarrow{\text{RD}}, \overrightarrow{\text{BD}} \rangle$, where $E_R$ is a finite set of events and $E_R \subseteq E$ and $\overrightarrow{\text{BD}}$ are relations defined over $E_R$.

The race related events in the set of $E_R$ are the events that have direct relationship with race conditions in $Q$. The race-dependence relation, denoted by $\overrightarrow{\text{BD}}$, shows the relative order in which events execute and the race dependence with each other. In this paper, we focus on message races and semaphore races. Assume there are 3 events $a, b, c \in E$, i.e., $a, b$ and $c$ are events in the
SYN-sequence $Q$:

- Assume $a$ and $b$ are sending message events, $c$ is the receive message event, and there is a message race between events $a$, $b$ with respect to $c$, i.e., $c$ may receive the message from $a$ first in one execution or even receive the message from $b$ first in another execution. If $a$ comes before $b$ in the $Q$, then $a \xrightarrow{RD} b$, otherwise, $b \xrightarrow{RD} a$.
- Assume $a$ and $b$ are taking semaphore events, and there is a semaphore race between events $a$ and $b$, i.e., event $a$ may take the semaphore firstly in one execution or $b$ may take the semaphore firstly in another execution. If $a$ comes before $b$ in the $Q$, then $a \xrightarrow{RD} b$, otherwise, $b \xrightarrow{RD} a$.

4.2 Race Condition Graph

Let $RS = < E_R, \xrightarrow{H}, \xrightarrow{RD}>$ be a race set of an execution sequence $Q$. An $RCG$ is a graph $G = <V, L>$, where $V = E_R$ is the set of vertices of the graph $G$ and $L \subseteq V^2$ is the set of relations of $V$. There are two kinds of $L$ in an $RCG$:

1) If events $a \in E_R$, $b \in E_R$ and $a \xrightarrow{H} b$, a solid arrow “$\xrightarrow{H}$” is used to show their happened-before relation in the RS.
2) If events $a \in E_R$, $b \in E_R$ and $a \xrightarrow{RD} b$, a dashed arrow “$\xrightarrow{RD}$” is used to show their race-dependence relation in the RS.

The SYN-Spec, i.e., the synchronization order of SYN-Events, can be described by an RCG. For example, Figure 6. shows a SYN-Spec of the dining philosopher case (which will be introduced in Section 6).

![Figure 6. An example of SYN-Spec](image)

In this SYN-Spec, there are 5 tasks ($T_1, T_2, \ldots, T_5$), 5 binary semaphores ($s_1, s_2, \ldots, s_5$), and 10 ST events. Assume the notation $\lambda(p, i, o)$ denotes the $i^{th}$ SYN-event in the task $p$ operating on the object $o$, the corresponding race set of this example is shown in Figure 7.

$$RS = \{ ST(T_1, 2, s_1) \xrightarrow{H} ST(T_4, 2, s_1); ST(T_2, 2, s_2) \xrightarrow{H} ST(T_4, 4, s_3); ST(T_3, 2, s_3) \xrightarrow{H} ST(T_4, 4, s_2); ST(T_4, 2, s_4) \xrightarrow{H} ST(T_5, 4, s_4); ST(T_5, 2, s_5) \xrightarrow{H} ST(T_5, 4, s_3); ST(T_1, 2, s_1) \xrightarrow{RD} ST(T_2, 4, s_1); ST(T_2, 2, s_2) \xrightarrow{RD} ST(T_3, 4, s_2); ST(T_3, 2, s_3) \xrightarrow{RD} ST(T_4, 4, s_3); ST(T_4, 2, s_4) \xrightarrow{RD} ST(T_5, 4, s_4); ST(T_5, 2, s_5) \xrightarrow{RD} ST(T_5, 2, s_5); \}$$

![Figure 7. The corresponding race set of the example](image)

Let $G = <V, L>$ be an RCG with respect to an execution sequence $Q$, and $G' = <V', L'>$ be a SYN-Spec. If every vertex $v \in V'$ also belongs to $V$ and every edge $v \in L'$ also belongs to $L$, i.e., $G'$ belongs to $G$, denoted by $G' \subseteq G$, then we say that $Q$ satisfies the SYN-Spec $G'$, denoted by $Q \models G'$.

Proposition. A real-time system $S$ described by TEA satisfies a SYN-Spec $G$ described by $RCG$, if and only if there is at least one execution sequence $Q$ obtained by simulating TEA and $Q \models G$.

5 Genetic Algorithm Based Approach for Parametric Analysis

Because of the non-determinacy of real-time embedded systems, there usually have some parameters unknown which influence the timing of the execution. In this paper, a GA based approach is used to search heuristically these timing parameters in a TEA. The goal of the analysis is to identify a set of timing parameters that result in a specific SYN-Spec. If no such timing parameters can be found, we can conclude that the SYN-Spec is not a valid one.

5.1 Chromosome Encoding

Assume there are $n$ timing parameters in the TEA. In the GA approach, each timing parameter $x_i$ ($1 \leq i \leq n$) is represented as a binary string with the length of $L_x$, which is decided by the range and precision of $x_i$. A chromosome is composed of a set of binary strings, which is shown in Figure 8.

![Figure 8. A chromosome encoding](image)

5.2 Fitness Function

The fitness function gives a quantitative measure of the suitability of the generated timing parameters for the purpose of satisfying the SYN-Spec. Let $G' = <V', L'>$ be a SYN-Spec. According to the proposition mentioned in Section 4.2, if the TEA satisfies $G'$, then there is a new RCG $G = <V, L>$ derived from a new execution sequence obtained by simulating the TEA, and $G' \subseteq G$. Therefore, in this paper, the fitness function is used to measure the similarity of $G'$ and $G$, i.e., the similarity of two RCGs.

In order to know the similarity of two RCGs, it needs to calculate the difference between them. Figure 9. shows an algorithm to calculate the difference between a SYN-Spec $G'$ and an RCG $G$ derived from a new execution sequence obtained by simulating the target
The statement (1) initializes a variable named difference. For each edge \( l'=<v', u'> \) from \( G' \), the statement (3) aims to find the same vertices \( v \) and \( u \) in \( G \) according to the SYN-event information described by \( v' \) and \( u' \). If it fails to find \( v \) or \( u \) in \( G \), then we say that the difference between \( G \) and \( G' \) is infinity. The statement (4) and (5) show that if the synchronization relation between \( v \) and \( u \) isn’t the same as \( v' \) and \( u' \), then the difference is increased.

### Process calculateDifference \((G', G)\)

1. Set the difference = 0 initially.
2. For each edge \( l'=<v', u'> \in L' \): (2)
3. Get the vertices \( v = v' \) and \( u = u' \) from \( V' \); (3)
4. If the synchronization relation between \( v \) and \( u \) isn’t the same as \( l' \), then (4)
   - difference = difference + 1;
5. End process
6. Loop
7. Return difference;
8. End process

Figure 9. An algorithm of calculating the difference between two RCGs.

Therefore, in the GA, the fitness of a chromosome (i.e., a set of timing parameters) is dominated by the difference value calculated using the algorithm shown in Figure 9. if the difference value between the SYN-Spec \( G' \) and a new RCG \( G \) becomes smaller, then \( G' \) becomes more similar with \( G \), i.e., the corresponding chromosome has higher contribution and suitability; if the difference value equals to 0, then \( G' \subseteq G \), i.e., the SYN-Spec is satisfied by the target TEA.

### 5.3 Termination Condition

The genetic algorithm is terminated at the following conditions: (1) it has successfully found a group of timing parameters with which an execution sequence generated by simulating the target TEA satisfies with the SYN-Spec; (2) the number of generations exceeds the upper limit and the genetic algorithm is terminated, at the situation we consider that the target model does not satisfy the SYN-Spec.

### 5.4 Analysis Process

Let \( S =<P, X, E, A, \tau, \mu, \gamma> \) be TEA and \( M \) is a set of timing parameters. The process of analysis is described in Figure 10.

#### Step 1: Initialization.

Generate a set of initial chromosomes randomly.

#### Step 2: Calculate fitness.

for each chromosome, do
- Decode it and get a set of values \( X \);
- Substitute \( X \) for \( M \) in \( S \);
- Simulate \( S \) and get a new execution sequence \( Q \);
- If all timing constraints are satisfied in \( Q \), then
  - Generate a new RCG \( G' \) from \( Q' \);
  - Output the \( Q \) and \( X \);
- Return success;

### Figure 10. Analysis process of the GA

Specially, if a chromosome cause deadlock during a simulation, the algorithm will mutate the chromosome and repeat the simulation again and again until the system runs smoothly.

### 6 A Case Study

As a case study, a real-time dining philosopher program, which is implemented on the VxWorks RTOS, is used for the case study. There are 5 dining philosophers with the same priorities: T1, T2, T3, T4 and T5. 5 chopsticks are represented by 5 binary semaphores \( s1, s2, s3, s4, \) and \( s5 \). To eat their dinner, T1 takes \( s1 \) and \( s5 \), T2 takes \( s2 \) and \( s1 \), T3 takes \( s3 \) and \( s2 \), T4 takes \( s4 \) and \( s3 \), T5 takes \( s5 \) and \( s4 \). Thus, there have 5 semaphore races in the program. In addition, to represent a possible thinking time before picking up a chopstick, there is a task delay event for random time before each semaphore taking operation. So, there are 10 timing parameters in the system: \( x1, x2, \ldots, x10 \). In the study, we assume that for each timing parameter \( x1 \), there has \( 1x \leq x \leq 8s \); the timing constraint between each task delay event and semaphore taking event is \([1, +\infty]\).

Figure 11. shows the T1 described by TEA, the effective execution time of each transition is calculated from traces of original executions on the VxWorks OS and a target processor board. The time unit is second. Other tasks are described by TEA similarly. The static data of the case is shown in TABLE II.
whether these event synchronization orders are possible before T5. The purpose of the case study is to verify if the different orders of taking chopsticks of 5 dining philosophers. For example, in SYN-Spec 1, T1 takes s1 before T2, T2 takes s2 before T3, T3 takes s3 before T4, while T5 takes s4 before T4 and T1 takes s5 before T5. The purpose of the case study is to verify whether these event synchronization orders are possible or not.

Figure 12. shows 4 SYN-Specs described by RCGs, which mean the different orders of taking chopsticks of 5 dining philosophers. For example, in SYN-Spec 1, T1 takes s1 before T2, T2 takes s2 before T3, T3 takes s3 before T4, while T5 takes s4 before T4 and T1 takes s5 before T5. The purpose of the case study is to verify whether these event synchronization orders are possible or not.

![Diagram](image-url)

Figure 12. Specifications of events synchronization

<table>
<thead>
<tr>
<th>TABLE II. Static Data of the case study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items</td>
</tr>
<tr>
<td>Processes</td>
</tr>
<tr>
<td>Events</td>
</tr>
<tr>
<td>Transitions</td>
</tr>
<tr>
<td>Variables</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
</tbody>
</table>

In this case study, the crossover rate of GA based approach is 0.7, the mutation rate of GA is 0.001, the revision rate is 0.5 and the number of initial population of chromosomes is 6. After simulating the target TEA, the result is shown in the TABLE III. The result shows that SYN-Spec 1 and 2 are satisfied by the TEA but SYN-Spec 3 and 4 are not. SYN-Spec 3 is infeasible because a global circle is included in SYN-Spec 3 [17], while the SYN-Spec 4 causes a deadlock and the event e4 (take the 2nd chopstick) can never succeed.

![Diagram](image-url)

Figure 12. Specifications of events synchronization

<table>
<thead>
<tr>
<th>TABLE III. Result of the case study</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN-Spec No.</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

7 Conclusion

In this paper, a simulation-based analysis with GA based search is presented to analyze event synchronization in real-time embedded systems. The TEA presented is composed of SYN-Events but not system states. Instead of using reachability analysis, the target TEA is simulated to generate execution sequences. Via analyzing execution sequences with a postmortem method, it is able to check whether a SYN-Spec described by RCG is satisfied or not. In the mean time, a GA based approach is used to analyze timing parameters in the target TEA. Experiments show that the method proposed can be used to find execution sequences according to the event synchronization order defined by user. As a consequence, the execution sequence defined by SYN-Spec can be indentified. Also, test sequences can be generated to verify the system operations.

References


