A Timed CSP Model for the Time-Triggered Language Giotto

Yanhong Huang[1], Yongxin Zhao[1], Shengchao Qin[2],
Guanhua He[2], and João F. Ferreira[2,3]
[1]Shanghai Key Laboratory of Trustworthy Computing
East China Normal University, Shanghai, P. R. China
[2]School of Computing, Teesside University, United Kingdom
[3]HASLab/INESC TEC, Universidade do Minho, Portugal
Email: {yhhuang, yxzhao}@sei.ecnu.edu.cn
{s.qin, g.he, jff}@tees.ac.uk

***Abstract*—Giotto is a time-triggered embedded programming language which provides an abstract programming model for hard real-time applications. It effectively decouples the implementation from the design. A Giotto program focuses on the functionality and timing of periodic tasks. All the actions, e.g., task invocations, actuator updates, and mode switches, described in Giotto programs are triggered by real time. We take the views of the concerns of Giotto programs, including the reaction to the environment, the communication between tasks, the timing predictability, etc. Our goal is to simulate Giotto programs using a timed CSP-based model which can effectively express the concerns and can be used to verify safety properties. This paper is a first step that presents the timed CSP model for Giotto programs. We also give a case study to illustrate the utility of the timed CSP model. Based on the existing research for CSP with time, we believe that our model can support to analyze and verify safety properties of Giotto programs.**

Keywords: Embedded Systems, Time-Triggered Language, Giotto, Timed CSP, Simulation

I. Introduction

Giotto [1] is designed specifically for hard real-time, reactive, safety-critical embedded systems, such as aircraft control system. The correctness of such kind of systems is dependent both on the logical correctness and the timing of their computing results. Giotto is a time-triggered language for embedded programming. It defines a software architecture of the implementation which specifies the functionality and timing of the real-time control system (Fig. 1). There are three levels: the first one is *control design*, the second one is *Giotto program*, and the third one is *Giotto compilation*. It effectively decouples the implementation task from the design task for the software engineers. A Giotto program decomposes the necessary computational activities into periodic tasks. It assigns periodic tasks into different modes and switches modes according to different requirements. In this level, software engineers do not need to specify where, how, and when tasks are scheduled. But it should present the functionality and timing of each task, as well as the communication between tasks. Giotto compilation will guarantee the correctness of the mapping and scheduling on a specific hardware configuration together with a real-time operating system. In this paper, we focus on the Giotto program level, which effectively and independently describe the logical concerns (functionality and timing) of Giotto-modeled systems.

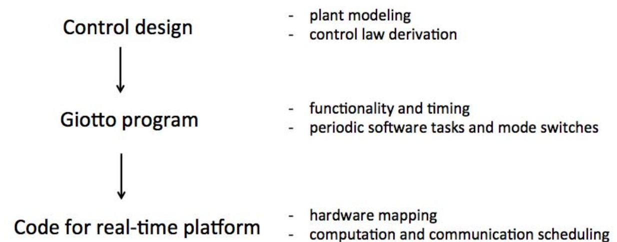

Fig. 1. Giotto-based control-systems development

Formal methods with their rigorously mathematical description and verification techniques are considered as approaches to ensure that system requirements are always correct and satisfied in the full development process, from logical specification to physical implementation. The goal of this paper is to simulate Giotto programs using a timed CSP-based model which can be used to analyze and verify this kind of programs. Timed CSP provides a timed stability model, which assumes that all events recorded by processes within the system relate to conceptual global clock, and a process can engage in only finitely many events in a bounded period of time. The two assumptions are in accordance with the characteristics of Giotto programs. All the actions, e.g., the periodic task invocations, the actuators updates, and the mode switches, are triggered by real time. The time operators in timed CSP are sufficient to express the timing of such a program which ensures the time of some actions take place is determined.

In this paper, we assume that all tasks in Giotto programs always terminate in their respective periods and prepare the results for updating the interrelating ports. Based on the assumption, our work aims to define a timed CSP-based model for Giotto programs. We simulate each component of a Giotto program by using a timed CSP process. The timed CSP model

1550-6215/13 $26.00 © 2013 IEEE
DOI 10.1109/SEW.2012.18

IEEE computer society

can easily represent the reaction of a Giotto program to the environment, the communication between periodic tasks, and when all interactions should take place. This is an essential step in our whole work. Some researchers defined kinds of semantics for CSP including time [9], [10], [11], while some researchers developed some tools to support the simulation, analysis and verification of timed CSP model [17], [18]. With these techniques, one of our further work is to use our model to analyze and verify safety properties of Giotto programs in the further.

Related Work. Henzinger et al. have done much work on Giotto. They presented the syntax and operational semantics of the Giotto language discussed in [1]. The operational semantics presents the behaviors of Giotto programs. They also presented an approach to the compilation of Giotto which provides time-safety checking [2]. Poddar and Bhaduri studied a translation scheme for building timed automata in UPPAAL for real-time systems written in Giotto. They also analyzed and verified the functional and timing properties using UPPAAL [6]. However, the focus and novelty of our work is on simulating a Giotto program using a timed CSP-based model. Timed CSP is a powerful language to model real-time reactive systems [16]. Some researchers have already applied timed CSP in formalizing systems with different features, such as job-shop scheduling problems, Eiffel' SCOOP, and Safety-Critical Java [12], [13], [14], [15].

The remainder of the paper is organized as follows: Section 2 introduces Giotto language as well as the characteristics of Giotto programs. We also list part of grammars of timed CSP which are used to model Giotto programs. In Section 3, we define a timed CSP model for Giotto programs. We simulate every component of a Giotto program into a timed CSP process, e.g., port, driver, task, and mode. In Section 4, we use the developed model in a case study. Finally, we conclude the paper in Section 5, where we discuss related work and further directions to develop the work presented.

II. Background

In this section, we describe the syntactic components of the Giotto language together with an example. We also introduce the language timed CSP which we will use to represent the Giotto program in section 3.

A. Giotto

Henzinger et al. [1] presented a time-triggered language Giotto which provides an abstract model for embedded control systems with hard real-time constraints. Giotto defines a software architecture which specifies functionality and timing, but abstracts away from the realization of the software architecture on a specific platform. It also does not care about issues such as hardware performance and scheduling mechanism. A Giotto program does not concern where, how and when tasks are scheduled, but specifies the time when the output ports of tasks are updated.

The abstract syntax of the Giotto language is shown as below:

$Prog ::= PortDecls^*\ \ TaskDecl^*\ \ DriverDecl^*\ \ ModeDecl^*\ \ Start$
$PortDecls ::= PortType\ \ PortDecl^*$
$PortType ::= \texttt{sensor} \mid \texttt{actuator} \mid \texttt{input} \mid \texttt{output} \mid \texttt{private}$
$PortDecl ::= \texttt{port}\ p\ \ \texttt{type}\ pt\ \ [\texttt{init}\ n]$
$TaskDecl ::= \texttt{task}\ t\ \ \texttt{input}\ p^*\ \ \texttt{output}\ p^*\ \ \texttt{private}\ p^*\ \ \texttt{function}\ f$
$DriverDecl ::= \texttt{driver}\ d\ \ \texttt{source}\ p^*\ \ \texttt{guard}\ g\ \ \texttt{destination}\ p^*\ \ \texttt{function}\ h$
$ModeDecl ::= \texttt{mode}\ m\ \ \texttt{period}\ \pi\ \ \texttt{port}\ p^*\ \ Invoke^*\ \ Update^*\ \ Switch^*$
$Invoke ::= \texttt{frequency}\ \omega\ \ \texttt{invoke}\ t\ \ \texttt{driver}\ d$
$Update ::= \texttt{frequency}\ \omega\ \ \texttt{update}\ d$
$Switch ::= \texttt{frequency}\ \omega\ \ \texttt{switch}\ m\ \ \texttt{driver}\ d$
$Start ::= \texttt{start}\ m$

A Giotto program consists of the following components: a *port* declaration set, a *task* declaration set, a *driver* declaration set and a *mode* declaration set, and it specifies a model as the *start* mode. The basic functional unit in Giotto is periodic *task*. Several concurrent tasks make up a *mode*. Different modes can share the same tasks. When switching from one mode to another, tasks can be added or removed from the model. Tasks communicate with each other, as well as with sensors and actuators, by so-called *drivers*, which are used to transport and convert values between *ports*. The detail of each component is discussed below.

Ports.

In Giotto, a port represents a typed variable through which data in the program is communicated. Each port has a unique identity p and preserves its value until it is updated. The data type pt of a port can be Integer, Real, Boolean or others specifies which type of value can be stored in the port. The initial value n is optional which initializes the port before the Giotto system starts. In a virtual concept, each port has a unique location in a globally shared name space which simplifies the definition of Giotto.

Ports are partitioned into three kinds: *sensor ports* which are updated by the environment; *actuator ports* which are updated by Giotto programs and are usually used to store the feedback information of Giotto programs; *task ports* which are also updated by Giotto programs and are used to communicate data between concurrent tasks. Moreover, task ports are divided into three sub-kinds: *input ports*, *output ports* and *private ports* to represent input, output and local state of a task respectively. In a mode, the output ports of tasks which are invoked in this mode are also called *mode ports*. When one mode starts, all the mode ports need to be initialized.

Drivers.

A driver d in Giotto transforms the data from a set of source ports Src to a set of destination ports Des. The transformation process $h : \mathsf{Vals}[\mathsf{Src}] \rightarrow \mathsf{Vals}[\mathsf{Des}]$ is a function which expresses how to convert the values of source ports to values of destination ports. This transformation by a driver

is regarded to take no time. Drivers can be guarded with the guard g : $\mathsf{Vals[Src]} \to \mathbb{B}$. The function h can only be executed if the guard g evaluates to *true* for the current source ports; otherwise, the Giotto program ignores the corresponding actions.

Drivers are needed when a task is invoked, an actuator port is updated or a mode switch happens. The source ports Src and destination ports Des of the driver d may be distinct in different situations. When a task is invoked, Src of d are sensor ports and mode ports (subset of output ports) of the current mode, and Des of d are the input ports of the current invoked task. If an actuator port is updated, Src of d are output ports of corresponding tasks and Des of d are the updated actuator ports. In each mode, an actuator port can be updated by at most one driver. In a mode switch scenario, Src of d are sensor ports and mode ports of the source mode, and Des of d are the mode ports of the target mode.

Tasks.

Task t is a functional unit in Giotto programs, and includes a set of input ports In, a set of output ports Out and a set of private ports Priv. The input ports In of t are distinct from all other ports in Giotto, and the output ports Out may be shared with other tasks if they are not in the same mode. The private ports Priv record the state of t and are inaccessible from the outside of the task. The function $f : \mathsf{Vals[In \cup Priv]} \to \mathsf{Vals[Out \cup Priv]}$ can be implemented by any sequential program, and indicates the functionality of the task. No synchronization points occur during the execution of tasks, all synchronization occurs outside of tasks, in other words, it happens between tasks.

All Giotto tasks are periodic. The invoked time of each task is at regularly spaced points. One task can be involved in different modes with different invocation frequencies. On the one hand a task cannot terminate prematurely, on the other hand it must have enough time to execute (the worst case execution time of every task needs to be provided for a given platform). An important point of Giotto tasks is that the time of updating the result of a task is determined.

Modes.

A Giotto program is made up by a set of modes. A mode m repeatedly executes a fixed set of tasks in every fixed period time π. The Giotto program is in one mode at a time. A Giotto program does not specify the computations of tasks as well as the physical scheduling. Moreover, the mode also needs to update some actuator ports with given frequencies. A mode may be switched to another mode at a specific time during the execution of the program. Each mode contains a set of mode switches, each of which denotes the target mode with a switch frequency. Giotto requires that only one of the switch conditions can be true at a time.

A mode m consists of a set of mode ports ModePorts, a set of task invocations Invokes, a set of actuator updates Updates and a set of mode switches Switches. ModePorts is the union of output ports of invoked tasks in this mode.

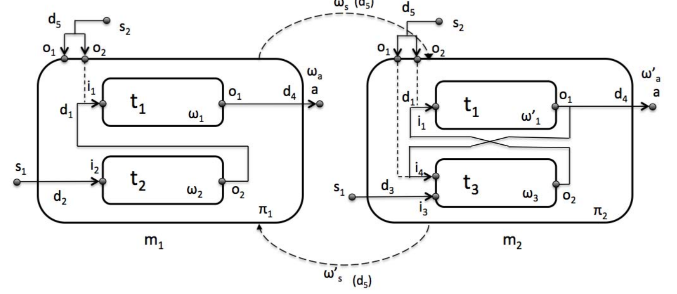

Fig. 2. An Example of Giotto

Invokes consists of a set of tasks, and each task t_{I} is assigned with a driver d_{I} and an invocation frequency ω_{I}. The mode invokes the task t_{I} every π/ω_{I} time units through the driver d_{I}, where π is the period of the mode m.

A driver d_{U} in Updates is used to update the actuator ports which are the destination ports of d_{U} in every π/ω_{U} time units, where ω_{U} denotes the frequency of update within the time period π of the mode m.

Every element of model switch Switches consists of a target mode m_{S}, the corresponding switch driver d_{S} and its invoked frequency ω_{S}. The mode switch evaluates periodically, as specified by the switch frequency ω_{S}. The guard of the switch driver d_{S} may only depend on the sensor ports and mode ports of the source mode, and the convert function of d_{S} transforms the values of the senor and mode ports of the source mode to the values of the mode ports of the target mode. A mode switch may occur while a task is logically running, i.e. the mode logically interrupts the task invocation.

Fig. 2 shows an example of Giotto which has two modes m_1 and m_2. The mode m_1 consists of two tasks t_1 and t_2, an actuator a, and a mode switch to mode m_2. The period of mode m_1 is π_1. The mode ports of mode m_1 are the output ports of two tasks: port o_1 and o_2. It invokes task t_1 with a frequency ω_1 and task t_2 with a frequency ω_2. The input/output ports of task t_1 are port i_1 and o_1 respectively. The input/output ports of task t_2 are port i_2 and o_2 respectively. The driver d_1 reads the output port o_2 of task t_2 and initializes the input port i_1 of task t_1. The driver d_2 reads the sensor port s_1 in environment and initializes the input port i_2 of task t_2. The driver d_4 gets the result of task t_1 from output port o_1 and updates the actuator port a with a frequency ω_a. The driver d_5 is a mode switch driver from mode m_1 to mode m_2 with a frequency ω_s. It prepares the mode ports o_1 and o_2 for mode m_2 from the sensor port s_2 when mode switch happens. The components of mode m_2 are much similar to mode m_1. One difference is the source ports of task t_3 are two ports: one is sensor port s_1, the other is the output port o_1 of task t_1. This example written in Giotto language is shown in Appendix.

B. Timed CSP

Communicating Sequential Processes (CSP) [5] is a well studied approach for synchronization and communication by introducing a concept "channel". Timed CSP is an extension of CSP by adding some time operators, which is proposed by Reed and Roscoe [3], and later modified by Davies and Schneider [4]. It extends the original CSP by adding some notations with timing constraints, e.g., $Wait\ t$. It is widely applied in modeling and verification. The syntax of a subset of Timed CSP which we will use to represent Giotto programs is given as follows.

$$P, Q \ ::= \ \mathsf{Stop} \mid \mathsf{Skip} \mid \mathit{Wait}\ t \mid a \to P \mid P;Q \mid P \,\square\, Q \mid P \sqcap Q \mid$$
$$c!a \to P \mid c?x \to P \mid f(P) \mid P \backslash A \mid P|||Q \mid P\|Q \mid \mu X \bullet F(X)$$

Stop is a broken program. Skip is defined as a program that does nothing but terminates immediately. $Wait\ t$ is a delay form of Skip which also does nothing but terminates successfully after t time units. Skip can be denoted as $Wait\ 0$. The program $a \to P$ is initially prepared to engage in synchronisation a. If this event occurs, it immediately begins to behave as P. The sequential program $P;Q$ behaves like P first and behaves like Q until P terminates. $P \square Q$ is an external choice between programs P and Q which is influenced by environment on the very first step. $P \sqcap Q$ is internal choice which is wholly nondeterministic.

$c!a \to P$ sends a value a via a channel c, while $c?x \to P$ is prepared to accept any value on channel c. $f(P)$ has a similar control structure to P with observable events renamed according to function f. The program $P \backslash A$ behaves as P except the events from set A are concealed form the environment of the program. Each action of the interleaving program $P|||Q$ is one of P or Q. If both of two sub programs have engaged in the same action, the choice between them is nondeterministic. The parallel program $P\|Q$ behaves like the program composed of P and Q interacting in lock-step synchronization. The recursive program $\mu X \bullet F(X)$ behaves as $F(X)$ with each instance of variable X representing a recursive invocation.

III. A Timed CSP Model for Giotto

In this section, we simulate a Giotto program using timed CSP. We define processes to denote all syntactic components of a Giotto program.

A. Modeling Ports

We define a process named $Port$ which contains two parameters $portname$ and $value$ to denote port in Giotto programs. Here, we assume the value of a port always matches the type of the port, so that we do not consider the type of the value mentioned in *PortDecl* in the syntax of the Giotto language. Moreover, we also use this process to denote all types of ports including sensor ports, actuator ports, input ports, output ports and private ports, in other words, we do not distinguish *PortType*. Process $Port$ can write or read value via the channels $write.portname$ and $read.portname$. If a new value arrives for a port over the channel $write.portname$, then the value is set to the new one. Otherwise, if another process requests the current value of a port, then the corresponding process $Port$ passes the value via channel $read.portname$. When declaring a port, we should set a possible initial value $init$ for it, like $Port(portname, init)$.

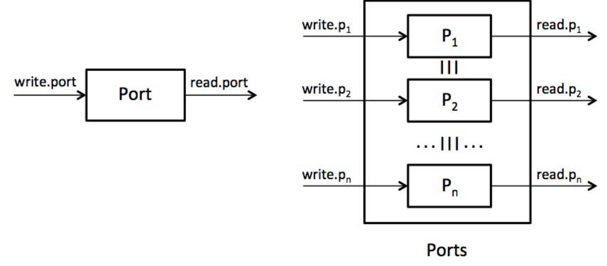

Fig. 3. A timed CSP process for port

$$\begin{aligned} &Port(portname, value) =_{df} \\ &\quad (write.portname?newvalue \to Port(portname, newvalue)) \\ &\quad \square\ (read.portname!value \to Port(portname, value)) \end{aligned}$$

All the ports in a Giotto program can be defined as if all $Port$ processes are running in interleaving. We assume there are n ports, each of which has a name p_i and an initial value $init_i$, where $i \in \{1, ..., n\}$.

$$Ports =_{df} |||_{i \in \{1,...,n\}}\ Port(p_i, init_i)$$

B. Modeling Drivers

We define a timed CSP process $Driver$ which consists of five parameters $drivername$, a guard g, source ports $srcports$, destination ports $desports$, and a function h. The source ports and destination ports may contain a set of ports. Here, the $srcports$ and $desports$ are the lists of $portnames$ of ports involved in this driver. The guard g and the function h are both many-to-many relations. The process $Driver$ first gets the values of $srcports$ via some channels $get.srcports$ and use variables Src to keep the values, where "$get.srcports?Src$" and some other similar expressions will be explained later. Secondly, the guard g is used to check whether the value of Src is true or not. If it is true, the function h is executed to calculate the values for $desports$ which are kept by variables Des. Then the process updates the destination ports via channels $send.desports$. Moreover, we introduce two events $drivername.start$ and $drivername.skipped$ to realize the synchronization with other related processes (e.g., the task invocation, the actuator update, and the mode switch) in different conditions.

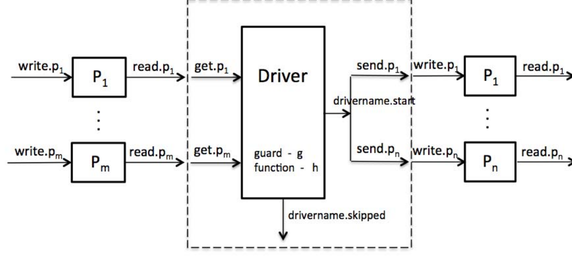

Fig. 4. A timed CSP process for driver

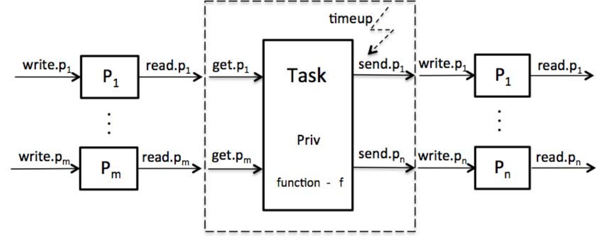

Fig. 5. A timed CSP process for task

$$
\begin{aligned}
&Driver(drivername, srcports, g, desports, h) =_{df} \\
&\quad get.srcports?Src \rightarrow \\
&\quad ((Des := h(Src) \rightarrow send.desports!Des \\
&\quad \rightarrow drivername.start \rightarrow \mathsf{Skip}) \\
&\quad \lhd\, g(Src) \rhd drivername.skipped \rightarrow \mathsf{Skip}); \\
&\quad Driver(drivername, srcports, g, desports, h)
\end{aligned}
$$

Here we explain the notations $get.ports?X$ or $send.ports!X$. We assume that $ports$ are used to denote a list of $portnames$ $\langle p_r\ p_{r+1}\ ...\ p_s\rangle$, and variables X are used to denote the corresponding values $x_r, x_{r+1}, ..., x_s$ got from / sent to those ports. We define

$$
\begin{aligned}
&get.ports?X \rightarrow P =_{df} \\
&\quad (get.p_r?x_r \rightarrow \mathsf{Skip} \mid\mid\mid get.p_{r+1}?x_{r+1} \rightarrow \mathsf{Skip} \\
&\quad \mid\mid\mid\ ...\ \mid\mid\mid get.p_s?x_s \rightarrow \mathsf{Skip}); P
\end{aligned}
$$

$$
\begin{aligned}
&send.ports!X \rightarrow P =_{df} \\
&\quad (send.p_r!x_r \rightarrow \mathsf{Skip} \mid\mid\mid send.p_{r+1}!x_{r+1} \rightarrow \mathsf{Skip} \\
&\quad \mid\mid\mid\ ...\ \mid\mid\mid send.p_s!x_s \rightarrow \mathsf{Skip}); P
\end{aligned}
$$

For example, there is a driver named d, in which the $srcports$ are p_1, p_2 and p_3, while the $desports$ are p_4 and p_5. The process can be described as below:

$$
\begin{aligned}
&Driver(d, \langle p_1\ p_2\ p_3\rangle, g, \langle p_4\ p_5\rangle, h) =_{df} \\
&\quad (get.p_1?x_1 \rightarrow \mathsf{Skip} \mid\mid\mid get.p_2?x_2 \rightarrow \mathsf{Skip} \mid\mid\mid get.p_3?x_3 \rightarrow \mathsf{Skip}); \\
&\quad ((y_1, y_2 := h(x_1, x_2, x_3) \rightarrow (send.p_4!y_4 \rightarrow \mathsf{Skip} \mid\mid\mid \\
&\quad\ \ send.p_5!y_5 \rightarrow \mathsf{Skip}); d.start \rightarrow \mathsf{Skip};) \\
&\quad \lhd\, g(x_1, x_2, x_3) \rhd d.skipped \rightarrow \mathsf{Skip}); \\
&\quad Driver(d, \langle p_1\ p_2\ p_3\rangle, g, \langle p_4\ p_5\rangle, h)
\end{aligned}
$$

In Giotto, the three actions: updating ports, the evaluation of a guard g and the execution of a function h in the driver take no time. So these two components of a Giotto program: $Port$ processes and $Driver$ processes described in the timed CSP model can be considered to execute without costing any time.

C. Modeling Tasks

We define a timed CSP process $Task$ which has five parameters $taskname$, input ports $inports$, output ports $outports$, private ports $privports$, and a function f. Like the definition of the process $Driver$, the $inports$, $outports$ and $privports$ are a list of $portnames$ to denote the corresponding ports needed in this task. The function f is a many-to-many relation. The process $Task$ firstly gets the values of $inports$ and $privports$ via channels $get.inports$ and $get.privports$ respectively. These values are kept in the variables Src and $Priv$ respectively. Secondly, the function f is executed to compute the results including the values of $outports$ and the new values of $privports$, which are kept by variables Des and $Priv'$ respectively. The private ports can be updated immediately by sending the values $Priv'$ via channels $send.privports$. But the time of updating output ports of a task is determined. The process needs to wait an event $taskname.timeup$ which is synchronized with other related process (a process $Invoke$ introduced later). Until this event occurs, the process can update the values of output ports via channels $send.outports$.

$$
\begin{aligned}
&Task(taskname, privports, inports, outports, f) =_{df} \\
&\quad get.inports?In \rightarrow get.privports?Priv \rightarrow \\
&\quad (Priv', Out) := f(Priv, In) \rightarrow send.privports!Priv' \rightarrow \\
&\quad taskname.timeup \rightarrow send.outports!Out \rightarrow \\
&\quad Task(taskname, privports, inports, outports, f)
\end{aligned}
$$

D. Modeling Modes

We define a timed CSP process $Mode$ which consists of $modename$, a period π, $modeports$ which is the union of output ports of all the invoked tasks in this mode, a set of task invocations $Invokes$, a set of actuator updates $Updates$ and a set of mode switches $Switches$. The notation $Invokes$ stands for a set of sub-processes $Invoke$ running in parallel, where the process $Invoke$ is simulated as a task invocation. The notation $Updates$ presents a set of sub-processes $Update$ running in parallel, where the process $Update$ is simulated as an actuator update. And the notation $Switches$ denotes a set of sub-processes $Switch$ running in parallel, where $Switch$

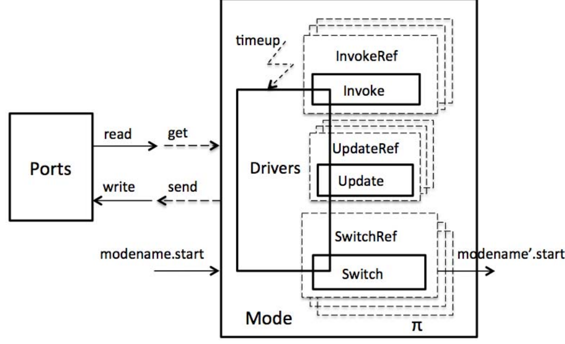

Fig. 6. A timed CSP process for Mode

is simulated as a mode switch. Now, we introduce the three sub-processes $Invoke$, $Update$, and $Switch$ at first.

A process $Invoke$ contains three parameters, an invocation frequency ω_{task}, a $taskname$ which refers to a related process $Task$ with the same task name, and a $drivername$ which refers to a related process $Driver$ with the same driver name. This process is defined by a parallelism among three processes: one is the process $Driver$, one is an external choice process which is related with the process $Task$, one is a process in charge of timing. For the first two processes, the process $Driver$ runs first, if the event $drivername.start$ offered by the process $Driver$ occurs, then the process $Task$ here can run. As explained earlier, if the guard g of the driver is evaluated to be true, the related process, such as the process $Task$ can run. Otherwise, the event $drivername.skipped$ takes place, then the corresponding task will not be invoked this time. At the same time, the process $Wait\ \pi/\omega_{task}$ in the third process consumes the time which should be taken by the task invocation in this mode. Afterwards, the third process offers an event $taskname.timeup$ or an event $drivername.skipped$ synchronized with the other two processes to ensure that the process $Invoke$ terminates after π/ω_{task} time units.

$$
\begin{aligned}
&Invoke(\omega_{task}, taskname, drivername) =_{df}\\
&\ Driver(drivername, srcports, g, desports, h) \parallel\\
&(drivername.start \rightarrow Task(taskname, privports, inports, outports, f)\\
&\ \Box\ drivername.skipped \rightarrow \mathsf{Skip}) \parallel\\
&(Wait\ \pi/\omega_{task}; (taskname.timeup \rightarrow \mathsf{Skip}\\
&\qquad\qquad\qquad \Box\ drivername.skipped \rightarrow \mathsf{Skip}))
\end{aligned}
$$

A process $Update$ contains two parameters, an update frequency ω_{act} and a $drivername$ which refers to a process $Driver$ with the same driver name. Actually, the updated actuator ports are included in the destination ports of the driver. This process contains two paralleling processes. One is the process $Driver$ which waits to be started with an event $drivername.timeup$, and the other is a process which is in charge of timing. As mentioned before, the driver will not take any time, so after π/ω_{act} time units (denoted by a process $Wait\ \pi/\omega_{act}$), the second process offers an event $drivername.timeup$, then there comes an external choice between two events $drivername.start$ and $drivername.skipped$ synchronized with the process $Driver$. Actually, only when the guard g of the driver is true, the actuator can be updated successfully.

$$
\begin{aligned}
&Update(\omega_{act}, drivername) =_{df}\\
&\quad (drivername.timeup \rightarrow\\
&\quad Driver(drivername, srcports, g, desports, h)) \parallel\\
&\quad (Wait\ \pi/\omega_{act}; drivername.timeup \rightarrow\\
&\quad\ (drivername.start \rightarrow \mathsf{Skip}\ \Box\ drivername.skipped \rightarrow \mathsf{Skip}))
\end{aligned}
$$

A process $Switch$ includes three parameters, a switch frequency ω_{switch}, a target mode name $modename'$, and a $drivername$ which refers to a process $Driver$ with the same driver name. This process which is much similar to the process $Update$ also contains two processes running in parallel: one is the process $Driver$, the other is the process with timing. After waiting π/ω_{switch} time units, an event $driver.timeup$ occurs to start the process $Driver$. When the guard g of the driver is true, the process $Driver$ will offer an event $drivername.start$ to synchronize with the second process. In the second process, an event $modename'.start$ following $drivername$ is added to start the target mode. That means the system switches to another mode. Otherwise, the current mode continues running when the guard g is false.

$$
\begin{aligned}
&Switch(\omega_{switch}, modename', drivername) =_{df}\\
&\quad (modename'.timeup \rightarrow\\
&\quad Driver(drivername, srcports, g, desports, h)) \parallel\\
&\quad (Wait\ \pi/\omega_{switch}; modename'.timeup \rightarrow\\
&\quad\ (drivername.start \rightarrow modename'.start \rightarrow \mathsf{Skip}\\
&\quad\ \ \Box\ drivername.skipped \rightarrow \mathsf{Skip}))
\end{aligned}
$$

According to the introduction of mode in section 2, we define a process $Mode$ which contains several parallel processes: a set of $Invoke$ processes, a set of $Update$ processes, and a set of $Switch$ processes. Each of these processes includes a parameter "a frequency ω", which denotes the execution times in one period of mode. In order to better describe the execution of these actions, we define another three processes $InvokeRef$, $UpdateRef$ and $SwitchRef$, which are correspond to $Invoke$, $Update$ and $Switch$ respectively to simulate the three actions in the mode. Each pair of them has the same parameters, i.e., both $InvokeRef$ and $Invoke$ have the same ω, $taskname$ and $drivername$. We also introduce a variable N to denote the execution times left for each of processes $Invoke$, $Update$ and $Switch$.

We assume there are p task invocations, q actuator updates, and r mode switches in one mode. Here we take task

invocations for example, the parameter $Invokes$ stands for all the p task invocations, and we use $i \in \{1, ..., p\}$ to denote these task invocations respectively. A process $InvokeRef(\omega_{task_i}, taskname_i, drivername_i)$ indicates that the mode invokes $taskname_i$ with a frequency ω_{task_i}, and a related driver is $drivername_i$. For process $InvokeRef$ of task $takename_i$, we use a variable N_i which initially equals the invocation frequency ω_{task_i} to denote the execution times in one period of the mode. Then the process $InvokeRef$ runs. If $N_i > 0$, it means that the task can be invoked in this period, then the corresponding process $Invoke$ can execute, while the invocation times variable N_i subtracts one. Otherwise, all the invocations of the task in the period of mode terminates. The processes about actuator updates and mode switches are similar to the task invocations, so we do not introduce them repeatedly.

$$
\begin{aligned}
&Mode(modename, \pi, modeports, Invokes, Updates, Switches) =_{df}\\
&\quad (\|_{i\in\{1,...,p\}} (N_i := \omega_{task_i};\\
&\qquad InvokeRef(\omega_{task_i}, taskname_i, drivername_i))\\
&\quad \|_{i\in\{1,...,q\}} (N_i := \omega_{act_i}; UpdateRef(\omega_{act_i}, drivername_i))\\
&\quad \|_{i\in\{1,...,r\}} (N_i := \omega_{switch_i};\\
&\qquad SwitchRef(\omega_{switch_i}, modename'_i, drivername_i)));\\
&Mode(modename, \pi, modeports, Invokes, Updates, Switches)
\end{aligned}
$$

$$
\begin{aligned}
&InvokeRef(\omega_{task_i}, taskname_i, drivername_i) =_{df}\\
&\ (N_i := N_i - 1; Invoke(\omega_{task_i}, taskname_i, drivername_i);\\
&\ InvokeRef(\omega_{task_i}, taskname_i, drivername_i))\\
&\ \lhd N_i > 0 \rhd \mathsf{Skip}
\end{aligned}
$$

$$
\begin{aligned}
&UpdateRef(\omega_{act_i}, drivername_i) =_{df}\\
&\quad (N_i := N_i - 1; Update(\omega_{act_i}, drivername_i);\\
&\quad UpdateRef(\omega_{act_i}, drivername_i))\\
&\quad \lhd N_i > 0 \rhd \mathsf{Skip}
\end{aligned}
$$

$$
\begin{aligned}
&SwitchRef(\omega_{switch_i}, modename'_i, drivername_i) =_{df}\\
&\ (N_i := N_i - 1; Switch(\omega_{switch_i}, modename'_i, drivername_i);\\
&\ SwitchRef(\omega_{switch_i}, modename'_i, drivername_i))\\
&\ \lhd N_i > 0 \rhd \mathsf{Skip}
\end{aligned}
$$

E. Modeling A Giotto Program

A Giotto program consists of several modes, one of which is the start mode decided initially. Moreover, only one mode is running at any time. We assume there are n modes in a Giotto program, denoted by $i \in \{1, ..., n\}$. We define a process $System$ which may have one parameter: the start mode name (denoted by $startname$), or have no parameter. In this process, we combine the processes about the modes and the ports in the Giotto program in parallel by replacing channel name "read/get" by "left", and replacing channel name "write/send" by "right". Each of the processes in $Mode$ awaits an event $modename.start$ to start to run. At the very beginning, the system runs the start mode by offering the event "$startname.start$". During the execution of the system, there is no need to set the start mode name again, which is denoted by $System()$.

$$
\begin{aligned}
&System(startname) =_{df}\\
&\ (startname.start \rightarrow \mathsf{Skip}) \parallel\\
&(((\Box_{i\in\{1,...,n\}} modename_i.start \rightarrow\\
&\ (Mode(modename_i, \pi_i, modeports_i, Invokes_i, Updates_i, Switches_i)\\
&\ \ \|System()))[left/get, right/send] \parallel\\
&\ Ports[left/read, right/write])\backslash\{left, right\})
\end{aligned}
$$

IV. The Case Study

In this section, we apply our approach to a case study: an elevator Giotto program [7]. We simulate this system by using our model formalized in the last section.

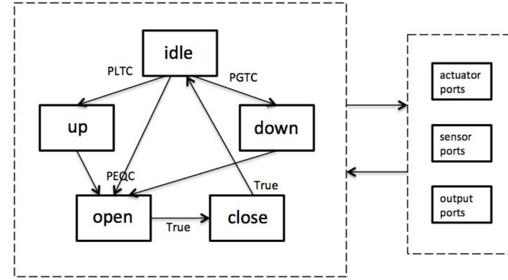

Fig. 7. A framework of an elevator Giotto program

The elevator Giotto program is controlling an one degree of freedom elevator: the elevator can move up/down to reach the requested floors; and only when the elevator reaches the requested floor, the elevator stops and the door opens. The elevator has two sensors in the environment: *buttons* and *position*. The *buttons* reads a request from each floor or the elevator. The *position* reads on which floor the elevator stays. There are two actuators: *mvtion* and *door*, which are used to respond to the request from sensors. The *mvtion* controls the elevator move up or down, while the *door* controls the door open or close. Both the sensors and actuators have corresponding ports. Moreover, the system also contains some task ports which are listed later. There are five modes, and each of them invokes only one task: *idle*, *up*, *down*, *open*, *close* (e.g., the *up* mode invokes the *Up* task). Each task reads the value of the *buttons* as input value, and uses its function to calculate the output values to update the *mvtion* and *door*. The period of every mode is 500 time units, and each mode invokes the corresponding task once in one period. The idle mode is the start mode of this system. When the requested floor is higher than the position of elevator, the system switches to the *up* mode. Conversely, the system switches to the *down* mode. After the elevator reaches the requested floor, the system switches to the *open* mode. Then the system switches to the *close* mode, afterwards switches back to the *idle* mode. The mode switches are shown in Fig.7.

Step 1. We use the process $Port(portname, values)$ to describe ports in the elevator Giotto program as below. According to the elevator Giotto program, we define eight interleaving processes to simulate all the ports in elevator system, and assign each of them a reasonable initial value. For the sensor port $buttons$, we set 0 as its initial value, which denotes there is no input value of this sensor port at the very beginning (process $Port(buttons, 0)$). If someone pushes the button in floor 2 for example, the value of $buttons$ is updated as 2. Similarly, process $Port(position, 0)$ indicates that the initial position of the elevator is floor 0. Processes $Port(m, stop)$ and $Port(d, close)$ means that the elevator stops and the door closes at the very beginning. The other four processes denote three output ports $tmotion, tdoor, openwin$ and one input port b. The initial value of $tmotion$ is $stop$ which accords with the corresponding actuator port m. Similarly, we set the initial values of the other three ports to $close, false, 0$.

$$Ports =_{df} Port(buttons, 0) \; ||| \; Port(position, 0) \; ||| \; Port(m, stop) \; ||| \; Port(d, close) \; ||| \; Port(tmotion, stop) \; ||| \; Port(tdoor, close) \; ||| Port(openwin, false) \; ||| \; Port(b, 0)$$

Step 2. We use the process $Driver(drivername, srcports, g, desports, h)$ to simulate seven drivers in elevator Giotto program. Note that we use "$\langle \; \rangle$" to denote that more than one port belong to one type of port (e.g., two ports p_1, p_2 are $srcports$ can be written as $\langle p_1 \; p_2 \rangle$), while use "$\cdot$" to stand for no port. Take the driver $Move$ for example, the corresponding code in Giotto is written as:

```
driver Move(tmotion) output (PortMovem) {
    if constant_true() then copy_PortMove(tmotion, m)}
```

Here, we define $Driver(Move, tmotion, constant_true(), m, copy_PortMove \; (tmotion, m))$ to describe this driver. The driver name is $Mode$, the source port is $tmotion$, the destination port is m, the guard is $constant_true()$, and the function is $copy_PortMove(tmotion, m)$. All the information of Giotto programs is included in this process. Similarly, the other six drivers are simulated using our timed CSP model.

$$\begin{aligned}
&Driver(Move, tmotion, constant_true(), m, \\
&\quad copy_PortMove(tmotion, m)) \\
&Driver(Door, tdoor, constant_true(), d, \\
&\quad copy_PortDoor(tddor, d)) \\
&Driver(getButtons, buttons, constant_true(), b, \\
&\quad copy_PortButtons(buttons, b)) \\
&Driver(PGTC, \langle buttons \; position \rangle, \\
&\quad CondPosGTCall(buttons, position), \cdot, dummy()) \\
&Driver(PLTC, \langle buttons \; position \rangle, \\
&\quad CondPosLTCall(buttons, position), \cdot, dummy()) \\
&Driver(PEQC, \langle buttons \; position \rangle, \\
&\quad CondPosEQCall(buttons, position), \cdot, dummy()) \\
&Driver(True, \cdot, constant_true(), \cdot, dummy())
\end{aligned}$$

Step 3. The simulation of task is much similar to the simulation of driver. The five tasks in the elevator Giotto program, have the same input port b, the same output ports $tmotion$, $tdoor$, and no private port. Although they have the same task ports, even they share a common driver $getButtons$. Because the five tasks have different functions, we should define five different processes to denote them.

$$\begin{aligned}
&Task(Idle, \cdot, b, \langle tmotion \; tdoor \rangle, TaskIdle(b, tmotion, tdoor)) \\
&Task(Up, \cdot, b, \langle tmotion \; tdoor \rangle, TaskUp(b, tmotion, tdoor)) \\
&Task(Down, \cdot, b, \langle tmotion \; tdoor \rangle, TaskDown(b, tmotion, tdoor)) \\
&Task(Open, \cdot, b, \langle tmotion \; tdoor \rangle, TaskOpen(b, tmotion, tdoor)) \\
&Task(Close, \cdot, b, \langle tmotion \; tdoor \rangle, TaskClose(b, tmotion, tdoor))
\end{aligned}$$

Step 4. We use $Mode(modename, \pi, modeports, Invokes, Updates, Switches)$ to simulate five modes. This process consists of several parallel processes. Take mode $idle$ for example, it contains one task invocation, two actuator updates and three mode switches. Firstly, the mode name and the period of the mode is described as two parameters in the process $Mode$. As the Giotto program does not indicate the mode ports of mode $idle$, we use "$\cdot$" to denote there is no mode ports. The three parameters $Invokes$, $Updates$, $Switches$ model that there are several parallel sub-processes in process $Mode$. When simulating each of the parallel processes, take a task invocation for example, we not only need to add the basic information in our model, such as the invocation frequency, the task name, and the driver name presented in process $InvokeRef(1, Idle, getbuttons)$, but also need to assign the invocation frequency to the variable N ($N := 1$), to present the invocation times. The other four modes are simulated like simulating mode $idle$.

$$\begin{aligned}
&Mode(idle, 500, \cdot, Invokes, Updates, Switches) =_{df} \\
&\quad ((N := 1; InvokeRef(1, Idle, getbuttons)) \parallel \\
&\quad (N := 1; UpdateRef(1, Move)) \parallel \\
&\quad (N := 1; UpdateRef(1, Door)) \parallel \\
&\quad (N := 1; SwitchRef(1, up, PLTC))) \parallel \\
&\quad (N := 1; SwitchRef(1, down, PGTC)) \parallel \\
&\quad (N := 1; SwitchRef(1, open, PEQC))); \\
&\quad Mode(idle, 500, \cdot, Invokes, Updates, Switches) \\
&Mode(up, 500, \cdot, Invokes, Updates, Switches) =_{df} \\
&\quad ((N := 1; InvokeRef(1, Up, getbuttons)) \parallel \\
&\quad (N := 1; UpdateRef(1, Move)) \parallel \\
&\quad (N := 1; UpdateRef(1, Door)) \parallel \\
&\quad (N := 1; Switch(1, open, PEQC))); \\
&\quad Mode(up, 500, \cdot, Invokes, Updates, Switches) \\
&Mode(down, 500, \cdot, Invokes, Updates, Switches) =_{df} \\
&\quad ((N := 1; InvokeRef(1, Down, getbuttons)) \parallel \\
&\quad (N := 1; Update(1, Move)) \parallel \\
&\quad (N := 1; Update(1, Door)) \parallel \\
&\quad (N := 1; Switch(1, open, PEQC))); \\
&\quad Mode(down, 500, \cdot, Invokes, Updates, Switches) \\
&Mode(open, 500, \cdot, Invokes, Updates, Switches) =_{df} \\
&\quad ((N := 1; Invoke(1, Open, getbuttons)) \parallel \\
&\quad (N := 1; Update(1, Move)) \parallel \\
&\quad (N := 1; Update(1, Door)) \parallel \\
&\quad (N := 1; Switch(1, close, True))); \\
&\quad Mode(open, 500, \cdot, Invokes, Updates, Switches) \\
&Mode(close, 500, \cdot, Invokes, Updates, Switches) =_{df} \\
&\quad ((N := 1; Invoke(1, Close, getbuttons)) \parallel \\
&\quad (N := 1; Update(1, Move)) \parallel
\end{aligned}$$

$(N := 1; Update(1, Door)) \parallel$
$(N := 1; Switch(1, idle, True))$
$Mode(close, 500, \cdot, Invokes, Updates, Switches)$

Step 5. At last, we use process $System$ to simulate the elevator Giotto program. The process is shown as below, the mode $idle$ is the start mode, so we put the name of this mode as the parameter of process $System$. As explained in the definition of process $System$ before, we use five events, $idle.start$, $up.start$, $down.start$, $open.start$, $close.start$, to control the corresponding mode to start to run.

$$
\begin{array}{l}
System(idle) =_{df} \\
(idle.start \rightarrow \mathsf{Skip}) \parallel \\
((((idle.start \rightarrow Mode(idle, 500, \cdot, Invokes, Updates, Switches) \| \\
\quad System()) \,\Box \\
\quad (up.start \rightarrow Mode(up, 500, \cdot, Invokes, Updates, Switches) \| \\
\quad System()) \,\Box \\
\quad (down.start \rightarrow Mode(down, 500, \cdot, Invokes, Updates, Switches) \| \\
\quad System()) \,\Box \\
\quad (open.start \rightarrow Mode(open, 500, \cdot, Invokes, Updates, Switches) \| \\
\quad System()) \,\Box \\
\quad (close.start \rightarrow Mode(close, 500, \cdot, Invokes, Updates, Switches) \| \\
\quad System())) \\
[left/get, right/send] \parallel Ports[left/read, right/write]) \backslash \{left, right\})
\end{array}
$$

Overall, we use timed CSP based model to simulate the whole elevator Giotto program step by step. This model can be implemented in some CSP based tools, such as PAT (Process Analysis Toolkit) which is an enhanced simulator, model checker and refinement checker for concurrent and real-time systems [18]. Then we can analyze some safety properties, i.e., deadlock free, or some timing properties with these tools.

V. Conclusion

In this paper, we investigated a formal model for Giotto programs, emphasizing the functionality and timing of abstract periodic tasks as well as the communication between these tasks. We applied timed CSP in formalizing Giotto programs. We simulated all components of Giotto programs, including ports, drivers, tasks and modes. We illustrated our approach by a case study for an elevator Giotto program, which shows that our modeling system can be successfully applied to practicable hard real-time systems.

With this model, we can analyze and verify some traditional safety properties, e.g., deadlock-free, and timing properties, e.g., the schedulability on different scheduling strategies. One of our further work is to translate our model to some tools, such as FDR [17] or PAT [18], to validate it, and use the tools to analyze and verify properties automatically. With the Unifying Theories of Programming (UTP) [8] techniques for reasoning, the other one is to analyze Giotto programs based on our formal model with the support of the UTP semantics of CSP with time [10], [11].

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).

The majority of the work reported in this paper was done while Yanhong Huang was a visiting researcher at Teesside University. The support of Teesside University is gratefully acknowledged.

References

[1] Thomas A. Henzinger, Benjamin Horowitz and Christoph Meyer Kirsch. *Giotto: A Time-Triggered Language for Embedded Programming*. Embedded Software. Lecture Notes in Computer Science, Volume 2211/2001, 2001, Pages 166-184.

[2] Thomas A. Henzinger, Christoph M. Kirsch, Rupak Majumdar and Slobodan Matic. *Time-safety checking for embedded programs*. Proceedings of the Second International Workshop on Embedded Software. Lecture Notes in Computer Science 2491, Springer-Verlag, 2002, Pages 76-92.

[3] G.M. Reed and A.W. Roscoe. *A Timed Model for Communicating Sequential Processes*. Automata, Language and Programing. Lecture Notes in Computer Science, Volume 226/1986, 1986, Pages 314-323.

[4] Jim Davies and Steve Schneider. *A Brief History of Timed CSP*. Meeting on the mathematical foundation of programing semantics. Volume 138, 1995, Pages 243-271.

[5] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[6] Rajiv Kumar Poddar and Purandar Bhaduri. *Verification of Giotto Based Embedded Control Systems*. Nordic Journal of Computing. Volume 13 Issue 4, 2006, Pages 266-293.

[7] An Elevator giotto program. http://embedded.eecs.berkeley.edu/giotto/examples/elevator.giotto. [Last accessed: 26 Nov 2012]

[8] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science, 1998.

[9] Steve Schneider. *An Operational Semantics for Timed CSP*. Information and Computation archive, Volume 116 Issue 2, Feb. 1, 1995, Pages 193-213.

[10] Jifeng He. *Integrating CSP and DC*. The Eighth IEEE International Conference on Engineering of Complex Computer Systems, 2002, Pages 47-54.

[11] Marcel Oliveira, Ana Cavalcanti and Jim Woodcock. *A UTP semantics for Circus*. Formal Aspects of Computing. Volume 21, Issue 1-2, 2009, Pages 3-32.

[12] Xian Zhang. *Reasoning about Timed CSP with Extensions*. Technical Report. http://www.comp.nus.edu.sg/~zhangxi5/tp.pdf.

[13] Xian Zhang. *Job-Shop Scheduling Problems using Timed Planning*. Secure Software Integration and Reliability Improvement Companion, 2010, Pages 110-117.

[14] Phillip J. Brooke, Richard F. Paige and Jeremy L. Jacob. *A CSP model of Eiffel's SCOOP*. Formal Aspects of Computing. Volume 19, Number 4, 2007, Pages 487-512.

[15] Frank Zeyda, Ana Cavalcanti, and Andy Wellings. *The Safety-Critical Java Mission Model: A Formal Account*. Formal Methods and Software Engineering. Lecture Notes in Computer Science, Volume 6991/2011, 2011, Pages 49-65.

[16] Steve Schneider. *Specification and Verification in Timed CSP*. Real-time Systems Specification, Verificaiton and Analysis, Chapter 6. Prentice Hall International, London, 2001.

[17] FDR2 User Manual : Available at: http://www.fsel.com/documentation/fdr2/html/index.html

[18] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. *PAT: Towards Flexible Verification under Fairness*. In 21th International Conference on Computer Aided Verification, Grenoble, France, 2009, Pages 702-708.

APPENDIX

We list the details of a Giotto program mentioned in the example in section 2. This Giotto program contains two modes, m_1 and m_2. Mode m_1 has a period of 6 ms, while mode m_2 has a period of 12 ms. The mode ports of the two modes are o_1 and o_2. Mode m_1 invokes task t_1 with a frequency 1 and invokes task t_2 with a frequency 2. Mode m_2 invokes task t_1 with a frequency 2 and invokes task t_3 with a frequency 3.

Task t_1 reads input port i_1 and writes output port o_1. Task t_2 reads input port i_2 and writes output port o_2. Task t_3 reads input ports i_3, i_4 and writes output port o_2. The drivers d_1, d_2 and d_3 prepare the input ports of tasks t_1, t_2 and t_3 respectively. The output port o_1 of task t_1 is read by driver d_4 to update the actuator port a. In both modes the actuator is updated every 6 ms. Mode m_1 may switch to mode m_2 every 3 ms, while m_2 may switch back to mode m_1 every 4 ms. These mode switches are controlled by driver d_5, which reads the sensor port s_2 and prepares mode ports o_1 and o_2 if the switch guard is true. In all, the start mode is mode m_1. The code is shown as below:

$\mathbb{R}$: the set of real numbers

$\mathbb{B}$:$\{true, false\}$

sensor

  port s_1 type $\mathbb{R}$

  port s_2 type $\mathbb{B}$

actuator

  port a type $\mathbb{R}$ init 0

intput

  port i_1 type $\mathbb{R}$

  port i_2 type $\mathbb{R}$

  port i_3 type $\mathbb{R}$

  port i_4 type $\mathbb{R}$

output

  port o_1 type $\mathbb{R}$ init 0

  port o_2 type $\mathbb{R}$ init 0

private

  port p_1 type $\mathbb{R}$ init 0

  port p_2 type $\mathbb{R}$ init 0

  port p_3 type $\mathbb{R}$ init 0

task t_1 input i_1 output o_1 private p_1 function f_1

task t_2 input i_2 output o_2 private p_2 function f_2

task t_3 input i_3, i_4 output o_2 private p_3 function f_3

driver d_1 source o_2 guard g_1 destination i_1 function h_1

driver d_2 source s_1 guard g_2 destination i_2 function h_2

driver d_3 source s_1, o_1 guard g_3 destination i_3, i_4 function h_3

driver d_4 source o_1 guard g_4 destination a function h_4

driver d_5 source s_2 guard g_5 destination o_1, o_2 function h_5

mode m_1 period 6 ports o_1, o_2

  frequency 1 invoke t_1 driver d_1

  frequency 2 invoke t_2 driver d_2

  frequency 1 update d_4

  frequency 2 switch m_2 driver d_5

mode m_2 period 12 ports o_1, o_2

  frequency 2 invoke t_1 driver d_1

  frequency 3 invoke t_3 driver d_3

  frequency 2 update d_4

  frequency 3 switch m_1 driver d_5

start m_1