Distributed Schedule Carrying Code [⋆]

Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic

EECS, University of California, Berkeley

Abstract. We present an approach for the design and implementation of embedded real-time software running on a distributed platform. The approach consists of a high-level specification layer instantiated by Giotto programs and a low-level verification and execution layer instantiated by Schedule carrying code (SCC). We explain a methodology in which several code suppliers, coordinated by a resource manager, independently generate and verify portions of the software system to be implemented on different hosts. A scheme for compiling Giotto programs by taking into account task (port) allocation to suppliers and hosts is described. After semantics of distributed SCC is presented we investigate composability properties. Since SCC executable carries its schedule as code, in order for distributed SCC to be composable we introduce a type for it, that for each supplier specifies time instants in which it is allowed to use computation or communication resources. We formally prove that if supplier SCC programs individually satisfy certain properties, namely type conformance and time safety, then the distributed SCC program correctly implements the original Giotto program specification. We demonstrate composability by showing that time to check these properties is proportional to the size of individual Giotto program portions. Although we assume static (time-triggered) computation and communication we make sure that the approach is valid also for the multi-modal Giotto programs.

1 Introduction

We are motivated by one of the major issues a car manufacturer faces in the integration of software subsystems in the computer controlled vehicle architecture. Due to the required determinism a common solution consists of computer nodes interacting by exchanging messages according to a static time-triggered communication protocol and running tasks that were scheduled off-line. The software subsystems are generated independently by several contractors (developers, suppliers) from specifications provided by a resource manager (car manufacturer, integrator). A typical problem occurs if a modification is required for a next generation of a product. A simple modification in a single software subsystem may induce a series of modifications in other subsystems. A change of timing attributes, e.g. task execution times, of one component should not cause the static schedule of any other components to change.

[⋆] This research was supported in part by the AFOSR MURI grant F49620-00-1-0327 and the NSF grants CCR-0208875 and CCR-0225610.

Components need to be designed based on their specifications, but otherwise independently of each other. The contractors should know as few information about other subsystems as possible. Also, the larger system is constructed from components that can be integrated without violating the principle of composability, i.e. properties that have been established at the component level should also hold at the system level. These are all common points of interest of our research and the study of linking interfaces in the dependable distributed systems [5]. A linking interface is characterized by data properties, i.e. the structure and semantics of the data crossing the interface, and temporal properties, i.e. the temporal constraints required for data delivery validity.

We study these problems in the scope of programming paradigms we previously developed. In our approach data properties are given in code compiled from the high-level system specification whereas temporal properties are given in the form of a special type that the code has to comply with. In the proposed real-time programming languages the system response is predictable both in time and value. In particular, in the *Giotto* programming language [1], this is achieved by separation of interaction with the environment from task execution. As long as the scheduler maintains all the deadlines, the output values at particular time instants do not depend on task ordering or preemption mechanism. Even a change in hardware should not affect the real-time behavior of the application. In such an approach, in addition to generating the code, the task of the compiler is to show the existence of a feasible schedule, which usually means to produce the schedule itself. This is the idea of the Schedule carrying code [3]: once the feasible schedule is produced, it can be attached to the code to serve as a certificate for schedulability.

For a given Giotto program the compiler generates code that is interpreted on two virtual machines, namely the E code for the embedded (E) machine and the S code for the scheduling (S) machine. The *embedded* machine [4] handles interaction with the environment and all requests for computational tasks. It runs E code that specifies timing and control-flow of software processes, i.e. tasks and drivers. The embedded machine has three basic instructions: `call`(d), for a driver d, `schedule`(t) for a task t, and `future`(g, b), for a trigger g and an E code address b. A trigger is a predicate over the inputs of the embedded machine, evaluated with every input event. The `call`(d) instruction immediately invokes a driver d. The `schedule`(t) instruction releases a task t for execution, by inserting it into the internal ready queue. In general, the `future`(g, b) instruction marks the E code at the address b for execution at some time when the trigger g becomes true. A simple abstraction used for E programs is that each E code instruction executes instantenously, i.e. in a negligible amount of time.

The *scheduling* machine [3] determines when, and in what order, tasks requested by the E code are actually executed (dispatched). The code that it runs, S code, represents a feasible schedule for the generated E code. An S program also consists of three instructions: `dispatch`(t, v) for a task t and a timeout v, `idle`(v) for a timeout v and `call`(d), for a driver d. A timeout, similar to a trigger, is a predicate over the inputs of the scheduling machine. The `dispatch`(t, v)

resumes the execution of a released task t until the timeout v becomes true. The task executes until either it completes or the timeout becomes true, whichever comes first. The `idle`(v) instruction simply makes the scheduling machine idle until the timeout v becomes true.

Lastly, the *Schedule carrying code* is the E code annotated with the S code. SCC is executable code which eliminates the need for a system scheduler. In our previous research we were primarily interested in how this model supports verification of real-time (schedulability) properties. In particular, we tried to establish the computational difference between generation of the SCC that satisfies those properties and validation of the same properties given some SCC. In case we don't trust the source of the code, or the code is moved to different targets, the schedule is revalidated before use. This rests on the premise that validation can be performed more efficiently than generation.

In contrast to a pure Giotto or a single processor SCC program, in a distributed SCC program data transmissions take place over intervals of time and not only at synchronous time instants. The only requirement is that transmissions to the task input ports occur before the task starts with the execution, and that transmissions from the task output ports occur after the task ends with the execution. However, to keep the program behavior deterministic with respect to the environment, drivers that read sensors or write to actuators are executed at the time instants that mark beginning (end) of the task periods.

Regarding related work, paper [6] deals with the similar problem of distributing programs written in Lustre language which is somewhat similar to Giotto. In fact, entire design chain, from a Simulink model to an implementation on the TTA bus is described. However, in this framework compositional design is not studied. Instead, end-to-end compilation scheme is presented together with the scheduling algorithm, which was not, on the other hand, the focus of our work.

The structure of the paper is as follows. Section 2 describes abstract syntax of the Giotto programming language and shows some programs that will be used as running examples. Section 3 starts with assumptions on the underlying architecture, gives an overview of the code generation process in our approach (3.1) and then explains the Giotto compilation part (3.2). We first present semantics of general SCC programs in section 4, and discuss it both for distributed SCC programs compiled from Giotto (4.1) and for such programs that encode earliest deadline first scheduling strategy (5.2). The basis that enables composability, the typing mechanism, is introduced in section 5.1. Formal properties of the methodology are studied in remaining subsections of 5. We prove input-output equivalence of Giotto and distributed SCC programs if certain conditions are satisfied and present an efficient algorithm that verifies these conditions.

2 Giotto language

We briefly discuss Giotto syntax and refer to [1] for details on semantics. A simple example of a Giotto program is shown on Fig 1 (for now ignore the annotations in brackets). A Giotto program begins with port declarations. A port is any typed variable. A *valuation* of a set of ports is a function that maps each port to a value of the appropriate type. The set *Ports* of all port names is partitioned into the following five sets: a set *SensePorts* of *sensor* ports, a set *ActPorts* of *actuator* ports, a set *InPorts* of *task input* ports, a set *OutPorts* of *task output* ports, and a set *PrivPorts* of *task private* ports. The set of sensor ports includes integer typed port p_c, a discrete clock port.

```
sensor
  $s_1$ $[c_1, h_1]$;
  $s_2$ $[c_2, h_2]$;
  $s_3$ $[c_2, h_2]$;
actuator
  $a_1$ $[c_1, h_1]$;
  $a_2$ $[c_2, h_2]$;
  $a_3$ $[c_2, h_2]$;
output
  $o_1$ $[c_1, h_1]$;
  $o_2$ $[c_2, h_2]$;
  $o_3$ $[c_2, h_2]$;
task
  $t_1(x_1)$ output($o_1$);
  $t_2(x_2)$ output($o_2$);
  $t_3(x_3)$ output($o_3$);
driver
  $i_1(s_2, o_2)$ output($x_1$);
  $i_2(s_1, o_1)$ output($x_2$);
  $i_3(s_3)$ output($x_3$);
  $g_1(o_1, o_2)$ output($a_1$);
  $g_2(o_1, o_2)$ output($a_2$);
  $g_3(o_3)$ output($a_3$);
start $m_1$ {
 mode $m_1$() period 24 {
  actfreq 3 do $a_1(g_1)$;
  actfreq 3 do $a_2(g_2)$;
  actfreq 1 do $a_3(g_3)$;
  taskfreq 1 do $t_1(i_1)$ ;
  taskfreq 3 do $t_2(i_2)$
  taskfreq 1 do $t_3(i_3)$
 }
}
```

Fig. 1. Single mode Giotto program G_1

The sensor ports cannot be directly read by tasks, and the task ports cannot be directly written to the actuators. Instead, a sensor or actuator port p requires a device driver $dev[p]$. A task output port p requires the declaration of an initialization driver $init[p]$ and a copy driver $copy[p]$. Each task output port is double-buffered, that is, it is implemented by two copies, a *local* copy that is used by the task only, and a *global* copy that is accessible to the rest of the program that executes on the same host. The initialization driver initializes the local copy; the copy driver copies data from the local copy to the global copy. A task private port requires only an initialization driver. For simplicity reasons the drivers mentioned in this paragraph were not shown in the figure with example Giotto programs.

```
output
  $o_1$ [$c_1, h_1$];
  $o_2$ [$c_2, h_2$];
  $o_3$ [$c_3, h_3$];
task
  $t_1(x_1)$ output($o_1$);
  $t_2(x_2)$ output($o_2$);
  $t_3(x_3)$ output($o_3$);
driver
  $i_1()$ output($x_1$);
  $i_2(o_1)$ output($x_2$);
  $i_3(o_1)$ output($x_3$);
  $n_1(o_1, o_2)$;
  $n_2(o_1, o_3)$;
start $m_1$ {
 mode $m_1$() period 24 {
  exitfreq 3 do $m_2(n_1)$;
  taskfreq 1 do $t_1(i_1)$;
  taskfreq 3 do $t_2(i_2)$;
 }
 mode $m_2$() period 24 {
  exitfreq 2 do $m_1(n_2)$;
  taskfreq 1 do $t_1(i_1)$;
  taskfreq 2 do $t_3(i_3)$;
 }
}
```

Fig. 2. Multiple mode Giotto program G_2

The second part of a Giotto program are task declarations. A *Giotto task* t has a set $\mathit{In}[t] \subseteq \mathit{InPorts}$ of input ports, a set $\mathit{Out}[t] \subseteq \mathit{OutPorts}$ of output ports, a set $\mathit{Priv}[t] \subseteq \mathit{PrivPorts}$ of private ports, and a task function $\mathit{task}[t]$ from the input and private ports to the private and output ports. Let set Tasks be the set of all tasks t of the Giotto program.

The third part of a Giotto program are driver declarations. Giotto drivers transport data between ports and initiate mode changes. A *Giotto driver* d has a set $\mathit{In}[d] \subseteq \mathit{Ports}$ of source ports, an optional driver guard $\mathit{cond}[d]$, which is evaluated on the source ports and returns a boolean, a set $\mathit{Out}[d] \subseteq \mathit{Ports}$ of destination ports, and a driver function $\mathit{drv}[d]$ from the source to the destination ports. The driver guard is implemented as a branching condition of the E machine; the driver function, as an E machine driver. Let set Drvs be the set of all drivers of the Giotto program.

The final part of a Giotto program declares the set Modes of modes and the start mode $\mathit{start} \in \mathit{Modes}$. A *mode* m has a period $\pi[m] \in \mathbb{N}_{>0}$, a set $\mathit{ModePorts}[m] \subseteq \mathit{OutPorts}$ of mode ports, a set $\mathit{Invokes}[m]$ of task invocations, a set $\mathit{Updates}[m]$ of actuator updates, and a set $\mathit{Switches}[m]$ of mode switches. A *task invocation* $(\omega_{task}, t, d) \in \mathit{Invokes}[m]$ consists of a task frequency $\omega_{task} \in \mathbb{N}_{>0}$ relative to the mode period, a task t, and an input task driver d, which loads the task inputs. An *actuator update* $(\omega_{act}, d) \in \mathit{Updates}[m]$ consists of an actuator frequency $\omega_{act} \in \mathbb{N}_{>0}$, and an actuator driver d. A *mode switch* $(\omega_{switch}, m', d) \in \mathit{Switches}[m]$ consists of a mode-switch frequency $\omega_{switch} \in \mathbb{N}_{>0}$, a target mode $m' \in \mathit{Modes}$, and a mode driver d, which governs the mode change. Figure 2 shows a multiple-mode Giotto program. If driver d is an input task driver then its input ports can be both sensor and task output ports, $In[d] \subseteq SensePorts \cup OutPorts$. If driver d is an actuator or mode switch driver we require that its input ports be only output ports, $In[d] \subseteq OutPorts$.

For a mode m, the least common multiple of the task, actuator, and mode-switch frequencies of m is called the number of *units* of m, and is denoted $\omega_{max}[m]$. We use an integer $k \in \{0, \dots, \omega_{max}[m] - 1\}$ as the *unit counter* for a mode m. Given a mode $m \in Modes$ and a unit k with $0 \leq k < \omega_{max}[m]$, we need the following auxiliary operators (see explanations in [2]):

$$\begin{array}{ll}
\multicolumn{2}{l}{taskInvocations(m,k) :=} \\
\quad \multicolumn{2}{l}{\{(\omega_{task}, t, d) \in Invokes[m] \mid k \cdot \omega_{task}/\omega_{max}[m] \in \mathbb{N}_{>0}\}} \\
Tasks(m,k) := & \{t \mid (\cdot, t, \cdot) \in taskInvocations(m,k)\} \\
taskDrivers(m,k) := & \{d \mid (\cdot, \cdot, d) \in taskInvocations(m,k)\} \\
taskOutputPorts(m,k) := & \{p \mid t \in Tasks(m,k) \wedge p \in Out[t]\} \\
\multicolumn{2}{l}{taskSensorPorts(m,k) :=} \\
\quad \multicolumn{2}{l}{\{p \mid d \in taskDrivers(m,k) \wedge p \in In[d] \cap SensePorts\}} \\
\multicolumn{2}{l}{preemptedTaskPeriods(m,k) :=} \\
\quad \multicolumn{2}{l}{\{\omega_{max}[m]/\omega_{task} \mid (\omega_{task}, \cdot, \cdot) \in Invokes[m] \setminus taskInvocations(m,k)\}} \\
\multicolumn{2}{l}{actuatorDrivers(m,k) :=} \\
\quad \multicolumn{2}{l}{\{d \mid (\omega_{act}, d) \in Updates[m] \wedge k \cdot \omega_{act}/\omega_{max}[m] \in \mathbb{N}_{>0}\}} \\
\multicolumn{2}{l}{actuatorPorts(m,k) :=} \\
\quad \multicolumn{2}{l}{\{p \mid d \in actuatorDrivers(m,k) \wedge p \in Out[d]\}} \\
\multicolumn{2}{l}{modeSwitches(m,k) :=} \\
\quad \multicolumn{2}{l}{\{(m', d) \mid (\omega_{switch}, m', d) \in Switches[m] \wedge k \cdot \omega_{switch}/\omega_{max}[m] \in \mathbb{N}_{>0}\}} \\
\multicolumn{2}{l}{modeSensorPorts(m,k) :=} \\
\quad \multicolumn{2}{l}{\{p \mid (\cdot, d) \in modeSwitches(m,k) \wedge p \in In[d] \cap SensePorts\}}
\end{array}$$

3 Distributing Giotto Programs

We assume that nodes of the distributed platform are connected by a single shared bus and that all communication is performed according to a Time Division Multiple Access (TDMA) prototcol: in each time interval only one node is allowed to send data while all other nodes can listen for the data. In a distributed system S code may represent the TDMA schedule in the form of executable instructions. Here we consider a simple architecture where each node has only one processor for both computation and communication tasks, but distributed systems with dedicated communication processors can be handled with some modifications. We formally treat messages communicated over the network similar to tasks. Instead of worst-case execution time, a message has worst-case transmission time. In order to simplify notation we also use the same SCC instructions for messages. If μ is a message then E code instruction `schedule`(μ) releases the message for transmission. Similarly, while S code instruction `dispatch`(μ, v) is executing the processor is busy with sending the data over the network until the transmission is over or the timer v becomes true.

3.1 Code Generation Flow

In our model the global resource manager generates a Giotto program G to be implemented by a set C of contractors on a set H of hosts. Let Ports be the set of all ports declared in program G. The resource manager allocates each port p except the dicrete clock p_c, $p \in \mathit{Ports} \setminus \{p_c\}$, to a host $\bar{h}(p)$ and determines a contractor $\bar{c}(p)$ that will implement code for port p. We assume that all processors are synchronized so that all contractors on all hosts have reading access to p_c. On figures 1 and 2 the port annotation given in brackets denotes the contractor and the host to which the port is allocated to. If port p is a sensor port then contractor $\bar{c}(p)$ implements device driver for p. If p is an actuator port then, beside the device driver, $\bar{c}(p)$ also implements actuator driver. Task input, output and private ports are defined for each task separately, so any task input, output, or private port p of a task must be allocated to the same contractor on the same host. Also, all destination ports of a driver $d \in \mathit{Drvs}$ must be allocated to the same contractor and to the same host, which we denote $\bar{c}(d)$ and $\bar{h}(d)$ respectively.

For each contractor $c \in C$ and each host $h \in H$ the resource manager gives out

- an E program $\mathcal{E}_{c,h}$ that describes timing and control-flow of driver, task and message invocations for contractor c on host h and
- a type $T_{c,h}$ that specifies computation and transmission time instants on host h available for contractor c (see next sections for formal type definition).

Once a contractor c gets E program and $\mathcal{E}_{c,h}$ and type $T_{c,h}$ for host h it generates

- an S program $\mathcal{S}_{c,h}$ for host h,
- worst case execution (transmission) times $w_{c,h}$ for tasks (messages) - $w_{c,h} : \mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h} \rightarrow \mathbb{N}_{>0}$, and
- functionality code for tasks.

Provided with the worst case execution and transmission times the resource manager finally verifies each generated S program against the corresponding type and E program. This way the resource manager can verify composability and ensure that the entire distributed SCC program satisfies Giotto semantics of program G. Once a contractor modifies its S program on a host, to check if Giotto semantics is preserved, it is sufficient to check only if this program conforms to its type. The emphasis here is not to develop the algorithm for generating type given the application (e.g. Giotto program). This is a scheduling problem and should be addressed separately possibly through several iterations of the proposed code generation flow.

3.2 From Giotto to Distributed E Code

Let P be a collection of generated distributed SCC programs for all contractors and over all hosts. In later sections we formally define syntax and semantics for such a distributed SCC program. The set of ports Ports_P of P contains additional ports ($\mathit{Ports} \subseteq \mathit{Ports}_P$) needed to store the data communicated over the network. Namely, if according to the Giotto program G and port-to-host allocation a value of the port $p \in \mathit{SensePorts} \cup \mathit{OutPorts}$ is needed as input to a driver on a host h different than $\bar{h}(p)$, i.e. if a message with the value of p must be communicated over the network, then the host h must keep its own copy p_h of port p. Note that $p_{\bar{h}(p)}$ is simply p. We assume that completion of message transmission updates the port p_h with the transmitted value. For each $p \in \mathit{SensePorts} \cup \mathit{OutPorts}$ let $\mu(p)$ be the message with the port p value.

We will need the following three operators in order to present compilation strategy for E programs of distributed SCC:

$$\begin{array}{l} \mathit{sendToHosts}(m,p) := \\ \quad \{\bar{h}(d) \mid ((\cdot,\cdot,d) \in \mathit{Invokes}[m] \vee (\cdot,d) \in \mathit{Updates}[m]) \wedge p \in \mathit{In}[d]\} \\ \quad \cup \{h \mid (\cdot,\cdot,d) \in \mathit{Switches}[m] \wedge h \in H \wedge p \in \mathit{In}[d]\} \setminus \{\bar{h}(p)\} \\ \mathit{sendToHosts}(p) := \qquad \{h \mid m \in \mathit{Modes} \wedge h \in \mathit{sendToHosts}(m,p)\} \\ \mathit{sendOutputPorts}(t) := \{p \mid p \in \mathit{Out}[t] \wedge \mathit{sendToHosts}(p) \neq \emptyset\} \end{array}$$

For a given mode m and port p the set $\mathit{sendToHosts}(m,p)$ is a set of hosts on which a task input, actuator or mode switch driver d is executed in mode m such that p is an input port of d. The host $\bar{h}(p)$ to which port p is allocated to is not in $\mathit{sendToHosts}(m,p)$, therefore this set is actually the set of hosts to which messages with port p values must be sent in mode m. The set $\mathit{sendToHosts}(p)$ is the set of hosts to which port p values must be sent in at least one of modes in Modes. For a given task t the set $\mathit{sendOutputPorts}(t)$ is a set of task t output ports for which there are hosts that should receive the message with port value.

We can formally define a message in an SCC program similar to a Giotto task: a message has a set of input and output ports and a function from the input to the output ports. In particular, for a message $\mu(p)$ let message input ports $\mathit{In}[\mu(p)]$ be $\{p\}$, message outputs ports $\mathit{Out}[\mu(p)]$ be $\{p_h \mid h \in \mathit{sendToHosts}(p)\}$ and a message function $\mathit{task}[\mu(p)]$ be identity function from the message input port to output ports. Let set Msgs be the set of all messages of program P.

According to Giotto semantics task input (output) driver reads (writes) input (output) ports at the time instants defined by the beginning (end) of the task period. In the distributed SCC implementation a task output driver is still performed at the end of the task period in an E code instruction. However, a task input driver is executed in an S code instruction and it is delayed because its input might need to be sent over the network. In general, in each task period, transmission of sensor input ports preceeds task execution which preceeds transmission of task output ports. Let d be the task input driver for a task t allocated to host h. For all ports $p \in \mathit{In}[d] \cap \mathit{SensePorts}$ such that $\bar{h}(p) \neq h$ a message $\mu(p)$ is received at h. Completion of the message $\mu(p)$ writes on each host $h' \in \mathit{sendToHosts}(p)$ to the sensor port $p_{h'}$. The task t input driver reads p_h (and maybe other ports) and writes to the task t input ports. It succeeds all sensor port messages and preceeds task t execution. Completion of a task t writes to the local copy of the task t output ports. Start of a task output port message $\mu(p')$ for $p' \in \mathit{Out}[t]$ succeeds the task t completion. Completion of a task output port message $\mu(p')$ writes on each of the hosts in $h'' \in \mathit{sendToHosts}(p')$ to the task output port $p_{h''}$. Finally, at each $h'' \in \mathit{sendToHosts}(p') \cup \{h\}$ a task output driver copies local into global task output ports at the end of the task t period.

We assume that the transmission of a sensor port value is performed in a time interval of length ϵ after the time instant the sensor is read. The *latency* value ϵ must be determined at compile time and in order to simplify code generation we also assume that this value is constant for all ports. If a task reads a sensor port that needs to be transmitted then the task input driver is called exactly ϵ time instants after task is released. Otherwise, it is executed at the time task is released. Symetrically, the transmission of task output ports is performed in a time interval of length ϵ before the time instant the task is logically completed (its period expires). Again, latency time ϵ is constant for all tasks t and modes m. We also require that time ϵ be less than the mode unit time $\gamma[m] = \pi[m]/\omega_{max}[m]$ for each mode m. This directly implies that the task input driver is always called before its input ports could be updated with new values.

Algorithm 1 The distributed Giotto program compiler

```
1: $\forall p \in \mathit{OutPorts} \cup \mathit{PrivPorts}$: $\mathit{emit}(\alpha, \bar{h}(p), \mathtt{call}(\mathit{init}[p]))$;
2: $\forall p \in \mathit{OutPorts}.\forall h \in \mathit{sendToHosts}(p) : \mathit{emit}(\alpha, h, \mathtt{call}(\mathit{init}[p_h]))$
3: $\forall h \in H$: $\mathit{emit}(\alpha, h, \mathtt{jump}(\mathtt{E}_{\alpha,h}^{start,0,M}))$;
4: $\forall m \in \mathit{Modes}$: invoke Algorithm 2 for mode $m$;
```

Algorithm 1 generates E programs $\mathcal{E}_{c,h}$ for each contractor $c \in C$ and each host $h \in H$. It is similar to the algorithm presented in [2] but modified for the distributed setting. The E code compiler command $\mathit{emit}(c, h, \mathit{instr})$ generates E code instruction *instr* for a contractor c on a host h. The symbolic reference $\mathtt{E}_{c,h}^{m,\mathtt{k},\mathtt{x}}$ denotes address of an instruction of $\mathcal{E}_{c,h}$ which is executed in mode m at unit k. The symbol $\mathtt{x}$ is used for three different parts of E code and is explained below. We assume that there is a special contractor α in C and for each host h the algorithm generates E program $\mathcal{E}_{\alpha,h}$ for it. These E programs are known only to a resource manager and typically contain only task input and output drivers and mode switch drivers. These programs do not contain any information needed by a contractor to generate S code. We give to a contractor as few information about other contractors as is necessary in order to enhance composibility. Note also that resource manager E program is always put first in the trigger queue, so it always executes first.

The compiler begins generating E code by emitting `call` instructions to the initialization drivers of all task output and private ports. Then an absolute `jump` is emitted to the first instruction of the start mode. The instruction $\mathtt{jump}(a)$ is shorthand for $\mathtt{if}(\mathtt{true}, a)$. Since this instruction is unknown at this point, we use a symbolic reference $\mathtt{E}_{c,h}^{m,\mathtt{k},M}$. The symbolic reference will be linked to the first instruction of the E code that implements mode m at unit k. The Algorithm 2 is called to generate code for all modes and units. It generates three types of E code blocks for each unit k of a mode m. The duration of a unit is denoted by γ. The

Algorithm 2 The distributed Giotto mode compiler

```
$k := 0$; $\gamma := \pi[m]/\omega_{max}[m]$;
while $k < \omega_{max}[m]$ do
  $\forall c \in C \,.\, \forall h \in H$: link $\mathtt{E}_{c,h}^{m,\mathtt{k},M}$ to the address of the next free instruction cell on $(c,h)$;
  $\forall p \in taskOutputPorts(m,k) \,.\, \forall h \in sendToHosts(p) \cup \{\bar{h}(p)\}$: $emit(\alpha, h, \mathtt{call}(copy[p_h]))$;
  $\forall d \in actuatorDrivers(m,k)$: $emit(\bar{c}(d), \bar{h}(d), \mathtt{call}(drv[d]))$;
  $\forall p \in actuatorPorts(m,k)$: $emit(\bar{c}(p), \bar{h}(p), \mathtt{call}(dev[p])$;
  $\forall p \in modeSensorPorts(m,k)$: $emit(\bar{c}(p), \bar{h}(p), \mathtt{call}(dev[p]))$;
  $\forall c \in C \,.\, \forall h \in H \,.\, \forall (m',d) \in modeSwitches(m,k)$: $emit(c, h, \mathtt{if}(cond[d], \mathtt{E}_{c,h}^{m,\mathtt{k},m'}))$;
  $\forall h \in H$: $emit(c, h, \mathtt{jump}(\mathtt{E}_{c,h}^{m,\mathtt{k},T}))$;

  $\forall (m',d) \in modeSwitches(m,k)$:
    $\forall c \in C \,.\, \forall h \in H$: link $\mathtt{E}_{c,h}^{m,\mathtt{k},m'}$ to the address of the next free instruction cell on $(c,h)$;
      // compute the unit $k'$ to which to jump in the target mode $m'$ and
      // compute the time $\delta'$ before new tasks in the target mode $m'$ can be scheduled
    if $preemptedTaskPeriods(m,k) = \emptyset$ then
      // jump to the beginning of $m'$ if all tasks in mode $m$ are completed
      $\delta' := 0$; $k' := 0$;
    else
      // compute the hyperperiod $\pi$ of the preempted tasks in units of mode $m$
      $\pi := \mathtt{lcm}(preemptedTaskPeriods(m,k))$;
      // compute the time $\delta$ to finish the hyperperiod $\pi$
      $\delta := (\pi - k \bmod \pi) * \pi[m]/\omega_{max}[m]$;
      // compute the time $\delta'$ to wait for the unit $k'$ in the target mode $m'$ to begin
      $\delta' := \delta \bmod (\pi[m']/\omega_{max}[m'])$;
      // compute the closest unit $k'$ to the end of the mode period in $m'$ after $\delta'$
      $k' := (\omega_{max}[m'] - (\delta - \delta') * \omega_{max}[m']/\pi[m']) \bmod \omega_{max}[m']$;
    end if
    $\forall h \in H$:
      $emit(\alpha, h, \mathtt{call}(drv[d]))$;
      $\forall c \in C$:
        if $\delta' > 0$ then
          $emit(c, h, \mathtt{future}(timer[\delta'], \mathtt{E}_{c,h}^{m',\mathtt{k}',M}))$;
          $emit(c, h, \mathtt{return})$;
        else
          $emit(c, h, \mathtt{jump}(\mathtt{E}_{c,h}^{m',\mathtt{k}',T}))$;
        end if

  $\forall c \in C \,.\, \forall h \in H$: link $\mathtt{E}_{c,h}^{m,\mathtt{k},T}$ to the address of the next free instruction cell on $(c,h)$;
  $\forall p \in taskSensorPorts(m,k)$:
    $emit(\bar{c}(p), \bar{h}(p), \mathtt{call}(dev[p]))$;
    if $sendToHosts(p) \neq \emptyset$ then $emit(\bar{c}(p), \bar{h}(p), \mathtt{schedule}(\mu[p] \prec \epsilon))$;
    end if
  $\forall (\cdot, t, d) \in taskInvocations(m,k)$:
    if $In[d] \cap taskSensorPorts(m,k) \neq \emptyset$ then $\epsilon_1 = \epsilon$ else $\epsilon_1 = 0$;
    if $sendOutputPorts(t) \neq \emptyset$ then $\epsilon_2 = -\epsilon$ else $\epsilon_2 = 0$;
    $emit(\bar{c}(t), \bar{h}(t), \mathtt{schedule}(\epsilon_1 \prec task[t] \prec \epsilon_2))$;
    $\forall p \in sendOutputPorts(t)$ :
      $emit(\bar{c}(t), \bar{h}(t), \mathtt{schedule}(-\epsilon \prec \mu[p]))$;
  $\forall c \in C \,.\, \forall h \in H$: $emit(c, h, \mathtt{future}(timer[\gamma], \mathtt{E}_{c,h}^{m,(\mathtt{k}+1) \bmod \omega_{max}[m],M}))$;
  $\forall c \in C \,.\, \forall h \in H$: $emit(c, h, \mathtt{return})$;
  $k := k + 1$;
end while
```

first type of E code block (line 3), labeled $\mathtt{E}_{c,h}^{m,\mathtt{k},M}$ takes care of updating task output ports, updating actuators, reading sensors, and checking mode switches. The compiler generates `call` instructions to the appropriate drivers and an `if` instruction for each mode switch. The block is terminated by a `jump` instruction to a block that deals with task invocations. The `jump` is only reached if none of the mode switches is enabled. The second type of E code block (line 12) implements the mode change to a target mode m'. We use the symbolic reference $\mathtt{E}_{c,h}^{m,\mathtt{k},m'}$ to label this type of E code block. Upon a mode change, the mode driver is called and then control is transfered to the appropriate E code block of the target mode. The compiler computes the destination unit k' as close as possible to the end of the target mode's period [1]. The trigger $timer[\delta']$ is a time trigger with enabling time δ'; that is, it specifies the trigger predicate $p'_c = p_c + \delta'$, which evaluates to true after δ' units of time (say ms) elapse. The third type of E code blocks (line 38) handles the invocation of tasks and the future invocation of the E machine for the next unit. The label for these blocks is $\mathtt{E}_{c,h}^{m,\mathtt{k},T}$. The final `future` instruction makes the E machine wait for the duration γ and then execute the E code for the next unit.

The `schedule` instructions in the algorithm (lines 46 and 48) are of the special form not present in the single processor SCC case. They indirectly contain precedence constraints needed for correct data transmission by explicitly specifying latency time ϵ. This number does not affect program execution itself, but a contractor needs it in order to construct a correct schedule, i.e. S program. The instruction $\mathtt{schedule}(\mu[p] \prec \epsilon)$ releases message $\mu[p]$ with a sensor port p value, but demands that the message transmission is finished in ϵ time. The instruction $\mathtt{schedule}(\epsilon \prec task[t] \prec -\epsilon)$ releases task t with the constraint that the task be dispatched after the time ϵ from the release time instant, and completed at the latest ϵ time before the task t logical completion time instant (period). Finally, the instruction $\mathtt{schedule}(-\epsilon \prec \mu[p])$ releases the message with task t output port p, with the constraint that the message be sent no earlier than ϵ time before the task t logical completion.

Note that the code generation scheme in algorithm 2 implies the order of execution: task output drivers are followed by actuator drivers, mode switch drivers, and task input drivers in that order. If the task output port $p \in OutPorts$ is an input port of an actuator, mode switch or task input driver that executes at a host h in a mode m then $h \in sendToHosts(p) \cup \{\bar{h}(p)\}$. The set of hosts that receives port p data does not depend on program mode. This means that a message with the port p value is sent to the host h even if the program executes in a mode in which p is not an input port to any driver at h.

Figures 3 and 5 show distributed E programs for the Giotto programs form figures 1 and 2. The code for different contractors on the same host is separated by a single horizontal line, the code for different hosts by two, and the code for different modes by three horizontal lines.

α, h_1:	$\mathtt{E}^{1,0,M}_{\alpha,1}$:	call(o_1)
		call(o_2)
		jump($\mathtt{E}^{1,0,T}_{\alpha,1}$)
			$\mathtt{E}^{1,1,M}_{\alpha,1}$:	call(o_2)
				jump($\mathtt{E}^{1,1,T}_{\alpha,1}$)
					$\mathtt{E}^{1,2,M}_{\alpha,1}$:	call(o_2)
						jump($\mathtt{E}^{1,2,T}_{\alpha,1}$)

	$\mathtt{E}^{1,0,T}_{\alpha,1}$:	future($8, \mathtt{E}^{1,1,M}_{\alpha,1}$)	$\mathtt{E}^{1,1,T}_{\alpha,1}$:	future($8, \mathtt{E}^{1,2,M}_{\alpha,1}$)	$\mathtt{E}^{1,2,T}_{\alpha,1}$:	future($8, \mathtt{E}^{1,0,M}_{\alpha,1}$)

c_1, h_1:	$\mathtt{E}^{1,0,M}_{1,1}$:	call(g_1)
		jump($\mathtt{E}^{1,0,T}_{1,1}$)
			$\mathtt{E}^{1,1,M}_{1,1}$:	call(g_1)
				jump($\mathtt{E}^{1,1,T}_{1,1}$)
					$\mathtt{E}^{1,2,M}_{1,1}$:	call(g_1)
						jump($\mathtt{E}^{1,2,T}_{1,1}$)

	$\mathtt{E}^{1,0,T}_{1,1}$:	call(s_1)
		schedule($\mu(s_1) \prec 2$)
		schedule($2 \prec t_1 \prec -2$)
		schedule($-2 \prec \mu(o_1)$)
		future($8, \mathtt{E}^{1,1,M}_{1,1}$)
			$\mathtt{E}^{1,1,T}_{1,1}$:	call(s_1)
				schedule($\mu(s_1) \prec 2$)
				future($8, \mathtt{E}^{1,2,M}_{1,1}$)
					$\mathtt{E}^{1,2,T}_{1,1}$:	call(s_1)
						schedule($\mu(s_1) \prec 2$)
						future($8, \mathtt{E}^{1,0,M}_{1,1}$)

α, h_2:	$\mathtt{E}^{1,0,M}_{\alpha,2}$:	call(o_1)
		call(o_2)
		call(o_3)
		jump($\mathtt{E}^{1,0,T}_{\alpha,2}$)
			$\mathtt{E}^{1,1,M}_{\alpha,2}$:	call(o_2)
				jump($\mathtt{E}^{1,1,T}_{\alpha,2}$)
					$\mathtt{E}^{1,2,M}_{\alpha,2}$:	call(o_2)
						jump($\mathtt{E}^{1,2,T}_{\alpha,2}$)

	$\mathtt{E}^{1,0,T}_{\alpha,2}$:	future($8, \mathtt{E}^{1,1,M}_{\alpha,2}$)	$\mathtt{E}^{1,1,T}_{\alpha,1}$:	future($8, \mathtt{E}^{1,2,M}_{\alpha,2}$)	$\mathtt{E}^{1,2,T}_{\alpha,2}$:	future($8, \mathtt{E}^{1,0,M}_{\alpha,2}$)

c_2, h_2:	$\mathtt{E}^{1,0,M}_{2,2}$:	call(g_2)
		call(g_3)
		jump($\mathtt{E}^{1,0,T}_{2,2}$)
			$\mathtt{E}^{1,1,M}_{2,2}$:	call(g_2)
				jump($\mathtt{E}^{1,1,T}_{2,2}$)
					$\mathtt{E}^{1,2,M}_{2,2}$:	call(g_2)
						jump($\mathtt{E}^{1,2,T}_{2,2}$)

	$\mathtt{E}^{1,0,T}_{2,2}$:	call(s_2)
		call(s_3)
		schedule($\mu(s_2) \prec 2$)
		schedule($2 \prec t_2 \prec -2$)
		schedule($-2 \prec \mu(o_2)$)
		schedule(t_3)
		future($8, \mathtt{E}^{1,1,M}_{2,2}$)
			$\mathtt{E}^{1,1,T}_{2,2}$:	call(s_2)
				schedule($\mu(s_2) \prec 2$)
				schedule($2 \prec t_2 \prec -2$)
				schedule($-2 \prec \mu(o_2)$)
				future($8, \mathtt{E}^{1,1,M}_{1,1}$)
					$\mathtt{E}^{1,2,T}_{2,2}$:	call(s_2)
						schedule($\mu(s_2) \prec 2$)
						schedule($2 \prec t_2 \prec -2$)
						schedule($-2 \prec \mu(o_2)$)
						future($8, \mathtt{E}^{1,0,M}_{2,2}$)

Fig. 3. Distributed E program for G_1

4 Semantics of Distributed SCC

In [3] we defined the abstract semantics of SCC which ignored all port values. Here we are interested in input-output behavior of the distributed SCC, so we present extended semantics by taking into account port values and data transmission over the network. We first describe a general form of a distributed SCC program not necessarily compiled from Giotto.

An *E program* $\mathcal{E} = (V, E, \kappa, \lambda)$ over a set of tasks T, a set of messages M and a set of drivers D consists of a control-flow directed graph (V, E), and two edge-labeling functions κ and λ. Each edge $e \in E$ is labeled with an instruction $\kappa(e)$ and an argument $\lambda(e)$ as follows:

- $\kappa(e) =$ `schedule` and $\lambda(e) \in T \cup M$. The execution of e releases the task or message $\lambda(e)$.
- $\kappa(e) =$ `call` and $\lambda(e) \in D$. The execution of e calls a driver $\lambda(e)$.

α, h_1:

$\mathtt{E}^{1,0,T}_{\alpha,1}$:	idle(2)	$\mathtt{E}^{1,1,T}_{\alpha,1}$:	idle(8)	$\mathtt{E}^{1,2,T}_{\alpha,1}$:	idle(8)
	call(i_1)				
	idle(8)				

c_1, h_1:

$\mathtt{E}^{1,0,T}_{1,1}$:	dispatch$(\mu(s_1), 1)$	$\mathtt{E}^{1,1,T}_{1,1}$:	dispatch$(\mu(s_1), 1)$	$\mathtt{E}^{1,2,T}_{1,1}$:	dispatch$(\mu(s_1), 1)$
	idle(2)		idle(1)		idle(1)
	dispatch$(t_1, 7)$		dispatch$(t_1, 7)$		dispatch$(t_1, 6)$
	idle(8)		idle(8)		idle(7)
					dispatch$(\mu(o_1), 8)$
					idle(8)

α, h_2:

$\mathtt{E}^{1,0,T}_{\alpha,2}$:	call(i_3)	$\mathtt{E}^{1,1,T}_{\alpha,1}$:	idle(2)	$\mathtt{E}^{1,2,T}_{\alpha,2}$:	idle(2)
	idle(2)		call(i_2)		call(i_2)
	call(i_2)		idle(8)		idle(8)
	idle(8)				

c_2, h_2:

$\mathtt{E}^{1,0,T}_{2,2}$:	idle(1)	$\mathtt{E}^{1,1,T}_{2,2}$:	idle(1)	$\mathtt{E}^{1,2,T}_{2,2}$:	idle(1)
	dispatch$(\mu(s_2), 2)$		dispatch$(t_3, 2)$		dispatch$(t_3, 2)$
	idle(2)		idle(2)		idle(2)
	dispatch$(t_2, 7)$		dispatch$(t_2, 7)$		dispatch$(t_2, 6)$
	dispatch$(t_3, 7)$		dispatch$(t_3, 7)$		dispatch$(t_3, 6)$
	idle(7)		idle(7)		idle(6)
	dispatch$(\mu(o_2), 8)$		dispatch$(\mu(o_2), 8)$		dispatch$(\mu(o_2), 7)$
	idle(8)		idle(8)		idle(8)

Fig. 4. Distributed S program for G_1

– $\kappa(e) = \mathtt{future}$ and $\lambda(e) \in \mathbb{N}_{>0} \times V$. The execution of e activates the trigger with the binding $\lambda(e) = (\delta, v)$, which means that after δ time units, E code will be executed starting from control location v.

An *S program* $\mathcal{S} = (V, E, \rho, \nu, \kappa, \lambda)$ over a set of tasks T, a set of messages M and a set of drivers D consists of a control-flow directed graph (V, E), two node-labeling functions ρ and ν, and two edge-labeling functions κ and λ. Each control location $u \in V$ is labeled by one of the following:

– $\rho(u) = \mathtt{dispatch}$ and $\nu(u) \in T \cup M$. The node u has a successor u' such that $\lambda(u, u') \in \mathbb{N}_{>0}$. If $\nu(u) \in T$ the execution of u dispatches the task $\nu(u)$. Control proceeds to u' if $\nu(u)$ completes or the first $\lambda(u, u')$ time units pass from the time at which the thread with this control location was created. If $\nu(u) \in M$ then the analogous explanantion holds for the transmission of the message $\nu(u)$.
– $\rho(u) = \mathtt{idle}$ and u has a successor u' such that $\lambda(u, u') \in \mathbb{N}_{>0}$. The execution of u idles the processor h until $\lambda(u, u') \in \mathbb{N}_{>0}$ time units pass from the time at which the thread was created.

m_1:

α, h_1:

$\mathtt{E}^{1,0,M}_{\alpha,1}$: call($o_1$)
call(o_2)
if($cond[n_1]$, $\mathtt{E}^{1,0,2}_{\alpha,1}$)
jump($\mathtt{E}^{1,0,T}_{\alpha,1}$)
$\mathtt{E}^{1,0,2}_{\alpha,1}$: call($n_1$)
jump($\mathtt{E}^{2,0,T}_{\alpha,1}$)
$\mathtt{E}^{1,0,T}_{\alpha,1}$: future($8$, $\mathtt{E}^{1,1,M}_{\alpha,1}$)

$\mathtt{E}^{1,1,M}_{\alpha,1}$: call($o_2$)
if($cond[n_1]$, $\mathtt{E}^{1,1,2}_{\alpha,1}$)
jump($\mathtt{E}^{1,1,T}_{\alpha,1}$)
$\mathtt{E}^{1,1,2}_{\alpha,1}$: call($n_1$)
future(4, $\mathtt{E}^{2,1,M}_{\alpha,1}$)
$\mathtt{E}^{1,1,T}_{\alpha,1}$: future($8$, $\mathtt{E}^{1,2,M}_{\alpha,1}$)

$\mathtt{E}^{1,2,M}_{\alpha,1}$: call($o_2$)
if($cond[n_1]$, $\mathtt{E}^{1,2,2}_{\alpha,1}$)
jump($\mathtt{E}^{1,2,T}_{\alpha,1}$)
$\mathtt{E}^{1,2,2}_{\alpha,1}$: call($n_1$)
future(8, $\mathtt{E}^{2,0,M}_{\alpha,1}$)
$\mathtt{E}^{1,2,T}_{\alpha,1}$: future($8$, $\mathtt{E}^{1,0,M}_{\alpha,1}$)

---

c_1, h_1:

$\mathtt{E}^{1,0,M}_{1,1}$: if($cond[n_1]$, $\mathtt{E}^{1,0,2}_{1,1}$)
jump($\mathtt{E}^{1,0,T}_{1,1}$)
$\mathtt{E}^{1,0,2}_{1,1}$: jump($\mathtt{E}^{2,0,T}_{1,1}$)
$\mathtt{E}^{1,0,T}_{1,1}$: schedule($t_1 \prec -2$)
schedule($-2 \prec \mu(o_1)$)
future(8, $\mathtt{E}^{1,1,M}_{1,1}$)

$\mathtt{E}^{1,1,M}_{1,1}$: if($cond[n_1]$, $\mathtt{E}^{1,1,2}_{1,1}$)
jump($\mathtt{E}^{1,1,T}_{1,1}$)
$\mathtt{E}^{1,1,2}_{1,1}$: future(4, $\mathtt{E}^{2,1,M}_{1,1}$)
$\mathtt{E}^{1,1,T}_{1,1}$: future(8, $\mathtt{E}^{1,2,M}_{1,1}$)

$\mathtt{E}^{1,2,M}_{1,1}$: if($cond[n_1]$, $\mathtt{E}^{1,2,2}_{1,1}$)
jump($\mathtt{E}^{1,2,T}_{1,1}$)
$\mathtt{E}^{1,2,2}_{1,1}$: future(8, $\mathtt{E}^{2,0,M}_{1,1}$)
$\mathtt{E}^{1,2,T}_{1,1}$: future(8, $\mathtt{E}^{1,0,M}_{1,1}$)

---

c_2, h_2:

$\mathtt{E}^{1,0,M}_{2,2}$: if($cond[n_1]$, $\mathtt{E}^{1,0,2}_{2,2}$)
jump($\mathtt{E}^{1,0,T}_{2,2}$)
$\mathtt{E}^{1,0,2}_{2,2}$: jump($\mathtt{E}^{2,0,T}_{2,2}$)
$\mathtt{E}^{1,0,T}_{2,2}$: schedule($t_2 \prec -2$)
schedule($-2 \prec \mu(o_2)$)
future(8, $\mathtt{E}^{1,1,M}_{2,2}$)

$\mathtt{E}^{1,1,M}_{2,2}$: if($cond[n_1]$, $\mathtt{E}^{1,1,2}_{2,2}$)
jump($\mathtt{E}^{1,1,T}_{2,2}$)
$\mathtt{E}^{1,1,2}_{2,2}$: future(4, $\mathtt{E}^{2,1,M}_{2,2}$)
$\mathtt{E}^{1,1,T}_{2,2}$: schedule($t_2 \prec -2$)
schedule($-2 \prec \mu(o_2)$)
future(8, $\mathtt{E}^{1,2,M}_{2,2}$)

$\mathtt{E}^{1,2,M}_{2,2}$: if($cond[n_1]$, $\mathtt{E}^{1,2,2}_{2,2}$)
jump($\mathtt{E}^{1,2,T}_{2,2}$)
$\mathtt{E}^{1,2,2}_{2,2}$: future(8, $\mathtt{E}^{2,0,M}_{2,2}$)
$\mathtt{E}^{1,2,T}_{2,2}$: schedule($t_2 \prec -2$)
schedule($-2 \prec \mu(o_2)$)
future(8, $\mathtt{E}^{1,0,M}_{2,2}$)

---

c_3, h_3:

$\mathtt{E}^{1,0,M}_{3,3}$: if($cond[n_1]$, $\mathtt{E}^{1,0,2}_{3,3}$)
jump($\mathtt{E}^{1,0,T}_{3,3}$)
$\mathtt{E}^{1,0,2}_{3,3}$: jump($\mathtt{E}^{2,0,T}_{3,3}$)
$\mathtt{E}^{1,0,T}_{3,3}$: future(8, $\mathtt{E}^{1,1,M}_{3,3}$)

$\mathtt{E}^{1,1,M}_{3,3}$: if($cond[n_1]$, $\mathtt{E}^{1,1,2}_{3,3}$)
jump($\mathtt{E}^{1,1,T}_{3,3}$)
$\mathtt{E}^{1,1,2}_{3,3}$: future(4, $\mathtt{E}^{2,1,M}_{3,3}$)
$\mathtt{E}^{1,1,T}_{3,3}$: future(8, $\mathtt{E}^{1,2,M}_{3,3}$)

$\mathtt{E}^{1,2,M}_{3,3}$: if($cond[n_1]$, $\mathtt{E}^{1,2,2}_{3,3}$)
jump($\mathtt{E}^{1,2,T}_{3,3}$)
$\mathtt{E}^{1,2,2}_{3,3}$: future(8, $\mathtt{E}^{2,0,M}_{3,3}$)
$\mathtt{E}^{1,2,T}_{3,3}$: future(8, $\mathtt{E}^{1,0,M}_{3,3}$)

---

m_2:

α, h_1:

$\mathtt{E}^{2,0,M}_{\alpha,1}$: call($o_1$)
call(o_3)
if($cond[n_2]$, $\mathtt{E}^{2,0,1}_{\alpha,1}$)
jump($\mathtt{E}^{2,0,T}_{\alpha,1}$)
$\mathtt{E}^{2,0,1}_{\alpha,1}$: call($n_2$)
jump($\mathtt{E}^{1,0,T}_{\alpha,1}$)
$\mathtt{E}^{2,0,T}_{\alpha,1}$: future($12$, $\mathtt{E}^{2,1,M}_{\alpha,1}$)

$\mathtt{E}^{2,1,M}_{\alpha,1}$: call($o_3$)
if($cond[n_2]$, $\mathtt{E}^{2,1,1}_{\alpha,1}$)
jump($\mathtt{E}^{2,1,T}_{\alpha,1}$)
$\mathtt{E}^{2,1,1}_{\alpha,1}$: call($n_2$)
future(4, $\mathtt{E}^{1,2,M}_{\alpha,1}$)
$\mathtt{E}^{2,1,T}_{\alpha,1}$: future($12$, $\mathtt{E}^{2,0,M}_{\alpha,1}$)

---

c_1, h_1:

$\mathtt{E}^{2,0,M}_{1,1}$: if($cond[n_2]$, $\mathtt{E}^{2,0,1}_{1,1}$)
jump($\mathtt{E}^{2,0,T}_{1,1}$)
$\mathtt{E}^{2,0,1}_{1,1}$: jump($\mathtt{E}^{1,0,T}_{1,1}$)
$\mathtt{E}^{2,0,T}_{1,1}$: schedule($t_1 \prec -2$)
schedule($-2 \prec \mu(o_1)$)
future(12, $\mathtt{E}^{2,1,M}_{1,1}$)

$\mathtt{E}^{2,1,M}_{1,1}$: if($cond[n_2]$, $\mathtt{E}^{2,1,1}_{1,1}$)
jump($\mathtt{E}^{2,1,T}_{1,1}$)
$\mathtt{E}^{2,1,1}_{1,1}$: future(4, $\mathtt{E}^{1,2,M}_{1,1}$)
$\mathtt{E}^{2,1,T}_{1,1}$: future(12, $\mathtt{E}^{2,0,M}_{1,1}$)

---

c_2, h_2:

$\mathtt{E}^{2,0,M}_{2,2}$: if($cond[n_2]$, $\mathtt{E}^{2,0,1}_{2,2}$)
jump($\mathtt{E}^{2,0,T}_{2,2}$)
$\mathtt{E}^{2,0,1}_{2,2}$: jump($\mathtt{E}^{1,0,T}_{2,2}$)
$\mathtt{E}^{2,0,T}_{2,2}$: future(12, $\mathtt{E}^{2,1,M}_{2,2}$)

$\mathtt{E}^{2,1,M}_{2,2}$: if($cond[n_2]$, $\mathtt{E}^{2,1,1}_{2,2}$)
jump($\mathtt{E}^{2,1,T}_{2,2}$)
$\mathtt{E}^{2,1,1}_{2,2}$: future(4, $\mathtt{E}^{1,2,M}_{2,2}$)
$\mathtt{E}^{2,1,T}_{2,2}$: future(12, $\mathtt{E}^{2,0,M}_{2,2}$)

---

c_3, h_3:

$\mathtt{E}^{2,0,M}_{3,3}$: if($cond[n_2]$, $\mathtt{E}^{2,0,1}_{3,3}$)
jump($\mathtt{E}^{2,0,T}_{3,3}$)
$\mathtt{E}^{2,0,1}_{3,3}$: jump($\mathtt{E}^{1,0,T}_{3,3}$)
$\mathtt{E}^{2,0,T}_{3,3}$: schedule($t_3 \prec -2$)
schedule($-2 \prec \mu(o_3)$)
future(12, $\mathtt{E}^{2,1,M}_{3,3}$)

$\mathtt{E}^{2,1,M}_{3,3}$: if($cond[n_2]$, $\mathtt{E}^{2,1,1}_{3,3}$)
jump($\mathtt{E}^{2,1,T}_{3,3}$)
$\mathtt{E}^{2,1,1}_{3,3}$: future(4, $\mathtt{E}^{1,2,M}_{3,3}$)
$\mathtt{E}^{2,1,T}_{3,3}$: schedule($t_3 \prec -2$)
schedule($-2 \prec \mu(o_3)$)
future(12, $\mathtt{E}^{2,0,M}_{3,3}$)

Fig. 5. Distributed E program for G_2

– $\rho(u) = \nabla$. If u has a successor u' then $\kappa(u, u') = \texttt{call}$ and $\lambda(u, u') \in D$. This indicates that control is at a transient instruction and the execution of (u, u') calls a driver $\lambda(u, u')$.

If we compile Giotto program into E and S programs we assume that each port is allocated to a particular contractor and host. Let the sets of tasks, messages, and drivers to be implemented by a contractor c on a host h be defined as $\mathit{Tasks}_{c,h} = \{t \in \mathit{Tasks} \mid \forall p \in \mathit{In}[t] \cup \mathit{Out}[t] \cup \mathit{Priv}[t] \;.\; \bar{c}(p) = c \wedge \bar{h}(p) = h\}$, $\mathit{Msgs}_{c,h} = \{\mu \in \mathit{Msgs} \mid \forall p \in \mathit{In}[\mu] \;.\; \bar{c}(p) = c \wedge \bar{h}(p) = h\}$, and $\mathit{Drvs}_{c,h} = \{d \in \mathit{Drvs} \mid \forall p \in \mathit{Out}[d] \;.\; \bar{c}(p) = c \wedge \bar{h}(p) = h\}$ respectively. For the rest of the paper, a node of a directed graph without incoming (outgoing) is called a source (sink) node of the graph.

An *SCC program* $P_{c,h} = (\mathcal{E}_{c,h}, \mathcal{S}_{c,h}, \Phi_{c,h})$ for a contractor c and a host h consists of an E program $\mathcal{E}_{c,h}$ and an S program $\mathcal{S}_{c,h}$ over sets $\mathit{Tasks}_{c,h}$, $\mathit{Msgs}_{c,h}$ and $\mathit{Drvs}_{c,h}$, and an *annotation function* $\Phi_{c,h}$ that maps each sink of the control graph of $\mathcal{E}_{c,h}$ to a node in the control graph of $\mathcal{S}_{c,h}$. When the E code execution arrives at a sink v, this creates a new thread of S code which starts at control location $\Phi_{c,h}(v)$. If $V^{\mathcal{E}}_{c,h}$ and $V^{\mathcal{S}}_{c,h}$ are respectively sets of control locations for $\mathcal{E}_{c,h}$ and $\mathcal{S}_{c,h}$ then define the following two sets of control locations $V^{\mathcal{E}}_{h} = \{v \in V^{\mathcal{E}}_{c,h} \mid c \in C\}$ and $V^{\mathcal{S}}_{h} = \{u \in V^{\mathcal{S}}_{c,h} \mid c \in C\}$, and a function $\Phi_h : V^{\mathcal{E}}_{h} \to V^{\mathcal{S}}_{h}$ such that $\Phi_h(v) = \Phi_{c,h}(v)$ if $v \in V^{\mathcal{E}}_{c,h}$.

A *distributed SCC program* P over a set C of contractors and a set H of hosts is a function that assigns to each $c \in C$ and each $h \in H$ an SCC program $P_{c,h}$ for a contractor c and a host h. In the following paragraphs we define operational semantics for a distibuted SCC program.

A *state* $q = (r, v, s, \tau, \theta)$ of P over a set C of contractors and a set H of hosts consists of a valuation function r of ports in Ports_P, a program counter function v that assigns to each host $h \in H$ a control node $v_h \in V^{\mathcal{E}}_{h}$, a status function $s : \mathit{Tasks} \cup \mathit{Msgs} \to \mathbb{N}_0 \cup \{\bot\}$, a trigger function τ that assigns to each host $h \in H$ a queue $\tau_h \subseteq (\mathbb{N}_0 \times V^{\mathcal{E}}_{h})^*$ of trigger bindings, and a thread function θ that assigns to each host $h \in H$ a set θ_h of threads. Let s be the function such that for each task $t \in \mathit{Tasks}$, the status $s(t) \in \mathbb{N}_0$ indicates that t has been released and executed for $s(t) \geq 0$ time units; the status $s(t) = \bot$ indicates that t has been completed (or not yet released). For a message $\mu \in \mathit{Msgs}$, $s(\mu)$ is defined analogously for the message release and transmission. Each thread $(u, \delta) \in \theta_h$ consists of a program counter $u \in V^{\mathcal{S}}_{h}$ and a number $\delta \in \mathbb{N}_0$ of time units for which the thread has been executed. If u is a sink, then the thread (u, δ) has terminated and may be removed from θ_h.

The state q has a *transition* to the state $q' = (r', v', s', \tau', \theta')$ if one of the following:

Completion S transition The state q is *completion enabling*, that is, there exist a host $h \in H$ and a thread $(u, \delta) \in \theta_h$ such that $s(\nu(u)) = \bot$ and $\rho(u) = \texttt{dispatch}$. Let the successor of u be u'. Then $r' = r$ except that $r'(\mathit{Out}[\nu(u)]) = f[\nu(u)](r(\mathit{In}[\nu(u)]))$, $(v', s', \tau') = (v, s, \tau)$ and $\theta' = \theta$ except that $\theta'_h = \theta_h \backslash \{(u, \delta)\}) \cup \{(u', \delta)\}$.

Transient S transition The state q is not completion enabling but *transient enabling*, that is, there exist a host $h \in H$ and a thread $(u, \delta) \in \theta_h$, such that $\rho(u) = \nabla$, the successor u is u' and $\kappa(u, u') = \texttt{call}$. Then $r' = r$ except that $r'(\mathit{Out}[\lambda(u, u')]) = f[\lambda(u, u')](r(\mathit{In}[\lambda(u, u')]))$, $(v', s', \tau') = (v, s, \tau)$ and $\theta' = \theta$ except that $\theta'_h = \theta_h \backslash \{(u, \delta)\}) \cup \{(u', \delta)\}$.

E transition The state q is neither completion nor transient enabling but *E enabling*, that is, there exists a host $h \in H$ and either (1) v_h has no successor and $(0, \cdot) \in \tau_h$, or (2) v_h has a successor v'_h. If (1) let $(0, \bar{v})$ be the first such pair in τ_h. Then $p = p'$, $v' = v$ except that $v'_h = \bar{v}$, $s' = s$, $\tau' = \tau$ except that $\tau'_h = \tau_h \setminus \{(0, \bar{v})\}$ and $\theta' = \theta$. If (2) then one of the following:

- $\kappa(v_h, v'_h) = \texttt{call}$ and $r' = r$ except that $r'(\mathit{Out}[\lambda(v_h, v'_h)]) = f[\lambda(v_h, v'_h)](r(\mathit{In}[\lambda(v_h, v'_h)]))$, $s' = s$ and $\tau' = \tau$;
- $\kappa(v_h, v'_h) = \texttt{schedule}$ and $r' = r$, $s' = s$ except that $s'(\lambda(v_h, v'_h)) = 0$, $\tau' = \tau$;
- $\kappa(v_h, v'_h) = \texttt{future}$ and $r = r'$, $s' = s$ and $\tau' = \tau$ except that $\tau'_h = \tau_h \circ \{\lambda(v_h, v'_h)\}$.

In all three cases, if v'_h is a sink, then $\theta' = \theta$ except that $\theta'_h = \theta_h \cup \{(\Phi_h(v'_h), 0)\}$; if v'_h is not a sink, then $\theta' = \theta$.

Timeout S transition The state q is neither completion nor transient nor E enabling but *timeout enabling*, that is, there exist a host $h \in H$ and a thread $(u, \delta) \in \theta_h$ such that $\rho(u) \in \{\texttt{dispatch}, \texttt{idle}\}$, the successor of u is u', $\lambda(u, u') \in \mathbb{N}_0$ and $\lambda(u, u') \leq \delta$. Then $(r', v', s, \tau') = (r, v, s, \tau)$, $\theta = \theta'$ except that $\theta'_h = (\theta_h \backslash \{(u, \delta)\}) \cup \{(u', \delta)\}$.

Time transition The state q is neither completion nor transient nor E nor timeout enabling. Then $r'(p) = r(p)$ for all $p \in \mathit{Ports}_P \setminus \{p_c\}$ and $r'(p_c) = r(p_c) + 1$. For $\sigma = r(p_c)$ we call function $r_\sigma = r$ the *port valuation at time* σ. For this transition it also holds $v' = v$ and for each $h \in H$ we have:

- the queue τ'_h results from τ_h by replacing each trigger binding (δ, u) by $(\delta - 1, u)$,
- the thread set θ'_h results from θ_h by replacing each thread (u, δ) by $(u, \delta + 1)$,
- let $X_h = \{x \mid (u, \cdot) \in \theta_h, \rho(u) = \texttt{dispatch}, \nu(u) = x\}$ and let $\bar{x} \in X_h$ be a task or message to be executed on h; if $x \in \mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h}$ for some $c \in C$, then $s'(x) = s(x) + 1$ or $s'(x) = \perp$ if $x = \bar{x}$, and $s'(x) = s(x)$ if $x \neq \bar{x}$; in case $s'(x) = \perp$ we say that on the transition (q, q'), task or message x *completes* after execution time $s(x) + 1$.

Note the priorities implied by this definition: transient S code that is enabled by the completion of tasks has priority over E code, which has priority over all remaining S code. See [3] for details on the order of transitions.

A *trace* of the distributed SCC program P is a sequence $\psi = q_0, q_1, \ldots, q_n$ of states of P such that (1) $q_0 = (\hat{r}, \hat{v}, \hat{s}, \cdot, \hat{\theta})$, where $\hat{r}(p_c) = 0$, $\hat{s}(x) = \perp$ for all $x \in \mathit{Tasks} \cup \mathit{Msgs}$, $\hat{v}_h \in V_h$ and $\hat{\theta}_h = \emptyset$ for all $h \in H$, and (2) for all $i \in \mathbb{N}_0$, there is a transition from q_i to q_{i+1}.

Let $w_{c,h} : \mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h} \to \mathbb{N}_{>0}$ be the worst case execution (transmission) time function for the tasks and messages of contractor $c \in C$ on host $h \in H$.

This function is provided by the contractor c. Let $w : \mathit{Tasks} \cup \mathit{Msgs} \to \mathbb{N}_{>0}$ be a function defined with $w(x) = w_{c,h}(x)$ for each $x \in \mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h}$. The trace ψ of P is an *w-trace* if for each task or message $x \in \mathit{Tasks} \cup \mathit{Msgs}$ and $i \in \mathbb{N}_0$, if x completes on the transition (q_i, q_{i+1}) of ψ, then it completes with execution (transmission) time at most $w(x)$.

4.1 Giotto Generated Distributed SCC

The E programs compiled according to the scheme presented in section 3.2 have a special form. Let G be a Giotto program, *Modes* the set of modes of G, and M the size of *Modes*. We assume that for each input Giotto program M is bounded by a constant. Note that according to Algorithm 2 each instruction of the compiled E program is associated with a Giotto program mode and a unit of the mode. Let the sets of tasks, messages, and drivers to be implemented by a contractor c on a host h be $\mathit{Tasks}_{c,h}$, $\mathit{Msgs}_{c,h}$ and $\mathit{Drvs}_{c,h}$ respectively. Let $g_{c,h}$ be equal to $|\mathit{Tasks}_{c,h}| + |\mathit{Msgs}_{c,h}| + |\mathit{Drvs}_{c,h}|$, i.e. let $g_{c,h}$ represent the size of the part of program G allocated to contractor c and host h. The size g of the Giotto program G is bounded by $O(\sum_{c \in C, h \in H} g_{c,h})$.

The tuple $\mathcal{E}_{c,h} = (V^{\mathcal{E}}_{c,h}, E^{\mathcal{E}}_{c,h}, \kappa, \lambda, \eta)$ is *G-generated E program* if it satisfies the following properties:

- The tuple $(V^{\mathcal{E}}_{c,h}, E^{\mathcal{E}}_{c,h}, \kappa, \lambda)$ is an E program and $\eta : V^{\mathcal{E}}_{c,h} \to \mathit{Modes} \times \mathbb{N}_0$ is a node-labeling function such that if $v \in V^{\mathcal{E}}_{c,h}$ and $\eta(v) = (m, k)$ then $k \in \{0, ..., \omega_{max}[m]\}$.
- The control graph $(V^{\mathcal{E}}_{c,h}, E^{\mathcal{E}}_{c,h})$ is a directed acyclic graph with exactly one source node v such that $\eta(v) = (m, k)$ for every mode $m \in \mathit{Modes}$ and every unit $k \in \{0, ..., \omega_{max}[m]\}$.
- Each path from a source to a sink of the control graph node consists of
  1. a sequence of $O(g_{c,h})$ edges (v, v') with $\kappa(v, v') = \texttt{call}$, followed by
  2. a sequence of $O(g_{c,h})$ edges (v, v') with $\kappa(v, v') = \texttt{schedule}$, followed by
  3. a single edge (v, v') with $\kappa(v, v') = \texttt{future}$ and $\lambda(v, v') = (\cdot, \bar{v})$ where $\bar{v}$ is a source of $V^{\mathcal{E}}_{c,h}$. The source on the path and $\bar{v}$ may or may not be associated with the same mode.
- For each $m \in \mathit{Modes}$ and each $k \in \{0, ..., \omega_{max}[m]\}$ at most one node v with $\nu(v) = (m, k)$ may have more than one successor. In that case v has no more than M successors. On each path either all nodes v have the same mode-unit pair $\nu(v)$ or $\nu(v)$ switches to a new value along the path.

If all numbers in G, i.e. mode periods as well as task and actuator frequencies, are bounded by n, then $\omega_{max}[m]$ is also bounded by n for all $m \in \mathit{Modes}$. The number of sources of $(V^{\mathcal{E}}_{c,h}, E^{\mathcal{E}}_{c,h})$ is $O(M \cdot n)$, and the number of sinks is $O(M^2 \cdot n)$. Since we consider the number of modes to be fixed, we have $|V^{\mathcal{E}}_{c,h}| = O(g_{c,h} \cdot n)$.

An SCC program $P_{c,h} = (\mathcal{E}_{c,h}, \mathcal{S}_{c,h}, \Phi_{c,h})$ for a contractor c and a host h is a *G-generated SCC program* if the following is satisfied:

- The tuple $\mathcal{E}_{c,h} = (V^{\mathcal{E}}_{c,h}, E^{\mathcal{E}}_{c,h}, \kappa, \lambda, \eta)$ is G-generated E program.

– The control graph $(V^{\mathcal{S}}_{c,h}, E^{\mathcal{S}}_{c,h})$ of the S program $\mathcal{S}_{c,h} = (V^{\mathcal{S}}_{c,h}, E^{\mathcal{S}}_{c,h}, \cdot, \cdot, \cdot, \cdot)$ consists of chains of total length $O(g_{c,h} \cdot n)$.
– The function $\Phi_{c,h}$ maps a sink node $v' \in V^{\mathcal{E}}_{c,h}$ to a source node $\Phi_{c,h}(v') \in V^{\mathcal{S}}_{c,h}$ such that if $(v, v') \in E^{\mathcal{E}}_{c,h}$, $\kappa(v, v') =$ `future` and $\lambda(v, v') = (\sigma, \cdot)$ then the chain in $(V^{\mathcal{S}}_{c,h}, E^{\mathcal{S}}_{c,h})$ that starts from node $\Phi_{c,h}(v')$ does not contain numbers, i.e. clock timeouts in `dispatch` and `idle` instructions, larger than σ.

According to the last condition, if the next E code instruction is executed after σ time units, then the chain of S code instructions describes the schedule for at most next σ time units. Note that if G is a single-mode program then both $(V^{\mathcal{E}}_{c,h}, E^{\mathcal{E}}_{c,h})$ and $(V^{\mathcal{S}}_{c,h}, E^{\mathcal{S}}_{c,h})$ consist of chains of size $O(g_{c,h})$.

A distributed SCC program P over a set C of contractors and a set H of hosts is a *G-generated distributed SCC program* if it consists of G-generated SCC programs $P_{c,h}$. For the initial state $q_0 = (\hat{r}, \hat{v}, \hat{s}, \hat{\tau}, \hat{\theta})$ of a trace of P we require that $\hat{v}_h$ is a source node $v_\alpha \in V^{\mathcal{E}}_{\alpha,h}$ with $\eta(v_\alpha) = (\mathit{start}, 0)$, and that $\hat{\tau}_h = \{(0, v_c) \mid c \in C \setminus \{\alpha\},\ v_c \text{ is a source in } V^{\mathcal{E}}_{c,h},\ \eta(v_c) = (\mathit{start}, 0)\}$.

5 Composing Distributed SCC

5.1 Types

Beside G-generated E program $\mathcal{E}_{c,h}$, a contractor $c \in C$ on host $h \in H$ receives for each mode mode $m \in \mathit{Modes}$ of the Giotto program G a *type*, a pair of predicates $T^m_{c,h} = (D^m_{c,h}, X^m_{c,h})$. The predicates $D^m_{c,h}, X^m_{c,h} : \{0, ..., \pi[m] - 1\} \to \{0, 1\}$ are defined as follows:

– $D^m_{c,h}(\sigma) = 1$ iff in mode m at time σ the contractor c at the host h may dispatch a task from $\mathit{Tasks}_{c,h}$,
– $X^m_{c,h}(\sigma) = 1$ iff in mode m at time σ the contractor c at the host h may send (dispatch) a message from $\mathit{Msgs}_{c,h}$.

Let $T_{c,h} = \{T^m_{c,h} \mid m \in \mathit{Modes}\}$ and let $T = \{T_{c,h} \mid c \in C, h \in H\}$. The type T is *feasible* for G if the following conditions are satisfied:

– (*Resource Sharing*) For all $m \in \mathit{Modes}$, $c_1, c_2 \in C$ $(c_1 \neq c_2)$, $h_1, h_2 \in H$ $(h_1 \neq h_2)$, and $\sigma \in \{0, ..., \pi[m] - 1\}$
RS1: at most one of $D^m_{c_1,h_1}(\sigma)$, $D^m_{c_2,h_1}(\sigma)$, $X^m_{c_1,h_1}(\sigma)$ and $X^m_{c_2,h_1}(\sigma)$ is equal to 1 [on each host at most one contractor may either dispatch or send at a time],
RS2: at most one of $X^m_{c_1,h_1}(\sigma)$, $X^m_{c_2,h_1}(\sigma)$, $X^m_{c_1,h_2}(\sigma)$ and $X^m_{c_2,h_2}(\sigma)$ is equal to 1 [at most one contractor on one host may send over the network at a time].
– (*Data Reception*) For all $m \in \mathit{Modes}$, all $k \in \{0, ..., \omega_{max}[m] - 1\}$, all $p \in \mathit{SensePorts} \cup \mathit{OutPorts}$ and all $\sigma \in \mathbb{N}_0$ if either
DR1: $p \in \mathit{taskSensorPorts}(m, k)$ and $k\gamma[m] \leq \sigma < k\gamma[m] + \epsilon$ [a sensor port data must be sent in the ϵ time window after it is read], or

DR2: $p \in \mathit{taskOutputPorts}(m, k+1)$ and $(k+1)\gamma[m] - \epsilon \leq \sigma < (k+1)\gamma[m]$ [a task output port data must be sent in the ϵ time window before it is written]

if $X^m_{\bar{c}(p),\bar{h}(p)}(\sigma) = 1$ then $D^m_{c,h}(\sigma) = X^m_{c,h}(\sigma) = 0$ for each $c \in C$ and $h \in \mathit{sendToHosts}(p)$ [when a host is supposed to receive data no dispatch or send is allowed].

Example feasible types for programs G_1 and G_2 are shown on figures 6 a) and 7.

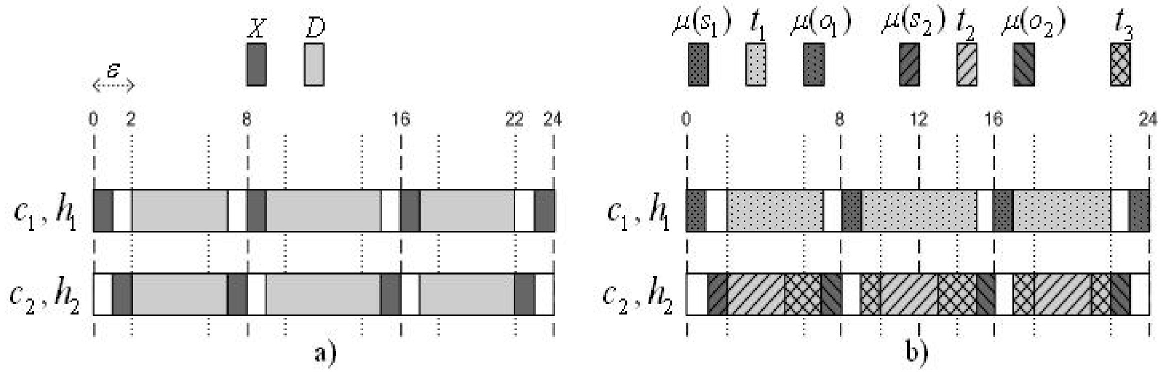

Fig. 6. a) Feasible type and b) EDF schedule for G_1

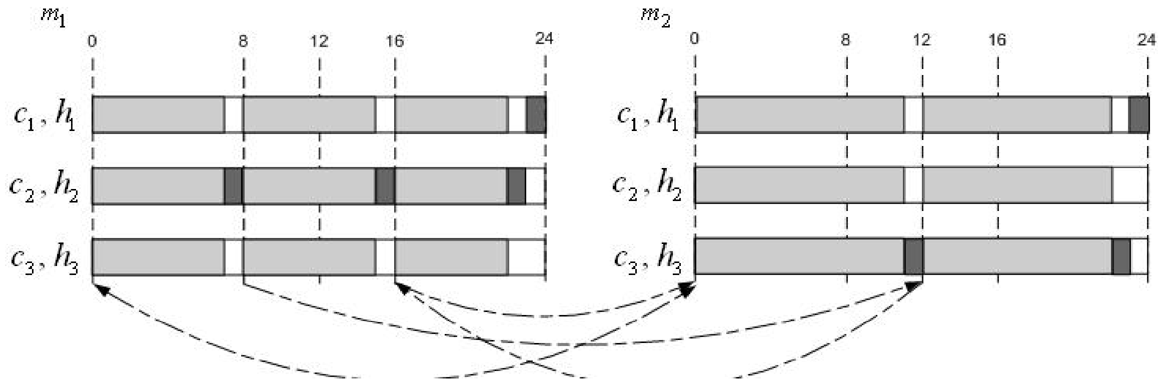

Fig. 7. Feasible type for multi-mode Giotto program G_2

5.2 Typed Earliest Deadline First S Programs

In the rest of the paper we assume that the type $T^m_{c,h} = (D^m_{c,h}, X^m_{c,h})$ is given as a sequence $J^m_{c,h} = \{[s, s')_j\}$ of $l^m_{c,h} \in \mathbb{N}_0$ nonintersecting and increasing intervals $[s, s')_j \subseteq [0, \pi[m])$ with integer bounds $s, s' \in \mathbb{N}_0$. For all $j \in \{1, ..., l^m_{c,h}\}$ and all integer $\sigma \in [s, s')_j$ it is either $D^m_{c,h}(\sigma) = 1$ or $X^m_{c,h}(\sigma) = 1$. For all integer $\sigma \in [0, \pi[m])$, $D^m_{c,h}(\sigma) = 1$ or $X^m_{c,h}(\sigma) = 1$ if and only if there exists $j \in \{1, ..., l^m_{c,h}\}$ such that $\sigma \in [s, s')_j$.

For each sink node $v \in V^{\mathcal{E}}_{c,h}$, i.e. each `future` instruction of $\mathcal{E}_{c,h}$, we construct a chain of the control graph of $\mathcal{S}_{c,h}$, i.e. we generate a block $\mathcal{S}^v_{c,h}$ of S code instructions `dispatch` and `idle`. Let $e \in E^{\mathcal{E}}_{c,h}$ be the incoming edge of v, $\kappa(e) =$ `future`, $\lambda(e) = (\sigma, \cdot)$ and $\eta(v) = (m, k)$ for some mode $m \in \mathit{Modes}$, integer time instant $\sigma \in [0, \pi[m])$ and unit $k \in \{0, ..., \omega_{max}[m]\}$. Let $J^v_{c,h}$ be the intersection of $J^m_{c,h}$ with the interval $[k\gamma[m], k\gamma[m]+\sigma)$, i.e. $J^v_{c,h} = \{[\mathtt{max}\{k\gamma[m], s\}, \mathtt{min}\{k\gamma[m]+\sigma, s'\}) \mid (s, s') \in J^m_{c,h}\}$.

Even if only time intervals from $J^m_{c,h}$ are avialable for task execution (message transmission) it can easily be shown by the standard interchange argument that the Earliest Deadline First (EDF) strategy is the optimal strategy with respect to schedule feasibility. So, contractors can always check EDF strategy and, if feasible, generate the S program $\mathcal{S}_{c,h}$ according to the following scheme. The release and deadline times of tasks and messages to be implemented by a contractor c on a host h in mode m are implicitly contained in the E program $\mathcal{E}_{c,h}$.

Let $t^{m,k}_{c,h}$ be the EDF permutation of tasks from $\mathit{Tasks}_{c,h}$ at unit k of mode m, i.e. let $t^{m,k}_{c,h}$ be the function that maps a number $i \in \{1, ..., |\mathit{Tasks}_{c,h}|\}$ to the task from $\mathit{Tasks}_{c,h}$ with the i-th earliest deadline at the time instant $k\gamma[m]$. We similarly define the function $\mu^{m,k}_{c,h}$, the earliest deadline function for messages at the unit k of mode m. The S code block of $\mathcal{S}^v_{c,h}$ contains for each interval $[s, s') \in J^v_{c,h}$ (in the order given by $J^v_{c,h}$) one of the two following sequences of instructions. It contains the instructions in the left column if $D^m_{c,h}(s) = 1$ and the instructions in the right column if $X^m_{c,h}(s) = 1$:

```
idle($s - k\gamma[m]$)                                          idle($s - k\gamma[m]$)
dispatch($t^{m,k}_{c,h}(1), s' - k\gamma[m]$)                   dispatch($\mu^{m,k}_{c,h}(1), s' - k\gamma[m]$)
dispatch($t^{m,k}_{c,h}(2), s' - k\gamma[m]$)                   dispatch($\mu^{m,k}_{c,h}(2), s' - k\gamma[m]$)
...                                                             ...
dispatch($t^{m,k}_{c,h}(|\mathit{Tasks}_{c,h}|), s' - k\gamma[m]$)   dispatch($\mu^{m,k}_{c,h}(|\mathit{Msgs}_{c,h}|), s' - k\gamma[m]$)
```

Note that this form of S code exploits the fact that an instruction that dispatches a task or a message which is not released is simply ignored. The Figure 4 shows S program for Giotto program G_1 which is generated according to the EDF scheme described above using type shown on Figure 6 a). The schedule represented by such S program is shown on Figure 6 b).

The size of both sequences shown above is bounded by $O(g_{c,h})$. The number of such sequences in mode m is bounded by the sum of the number of units $\omega_{max}[m]$ and the number of intervals $l^m_{c,h}$. If all numbers in G are bounded by n, then for each mode m both $\omega_{max}[m]$ and $l^m_{c,h}$ are $O(n)$. For a given mode unit there are $O(M^2)$ sink nodes in $(V^{\mathcal{E}}_{c,h}, E^{\mathcal{E}}_{c,h})$, but since we consider the number of modes to be bounded, the size of the S program $\mathcal{S}_{c,h}$ is $|V^{\mathcal{S}}_{c,h}| = O(g_{c,h} \cdot n)$. Therefore, it is easy to check that $\mathcal{S}_{c,h}$ generated according to the above scheme and $\mathcal{E}_{c,h}$ satisfy the requirements for G-generated SCC program $P_{c,h} = (\mathcal{E}_{c,h}, \mathcal{S}_{c,h}, \Phi_{c,h})$.

5.3 Distributed SCC implementing Giotto

Let G be a multiple mode Giotto program, let $P = \{P_{c,h} \mid c \in C, h \in H\}$ be a G-generated distributed SCC program and let $T = \{T_{c,h} \mid c \in C, h \in H\}$ be a feasible type for G. The following properties of G-generated SCC programs $P_{c,h}$ will be used to state composability result:

– *Type compliance* of $P_{c,h}$ with $T_{c,h}$ is satisfied if all `dispatch` instructions of $P_{c,h}$ execute in time intervals defined in $T_{c,h}$.
  Formally, a state $q = (\cdot, v, \cdot, \cdot, \theta)$ of P violates *type compliance with* $T_{c,h} = (D_{c,h}, X_{c,h})$ if there exists a thread $(u, \delta) \in \theta_h$ such that $\rho(u) = \texttt{dispatch}$, $\eta(v_h) = (m, k)$ and either
  - $\nu(u) \in \mathit{Tasks}_{c,h}$ and $D^m_{c,h}(k\gamma[m] + \delta) = 0$, or
  - $\nu(u) = \mathit{Msgs}_{c,h}$ and $X^m_{c,h}(k\gamma[m] + \delta) = 0$.

  We say that $(P_{c,h}, w_{c,h})$ *type-complies* with $T_{c,h}$ if for all $w_{c,h}$-traces ψ of $P = \{P_{c,h}\}$ no state of ψ violates type compliance with $T_{c,h}$.
– *Time safety* property of $P_{c,h}$ requires that
  1. no driver reads from output ports of a task or message from $\mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h}$ before it completes execution (transmission), and
  2. no driver writes to input ports of a task or message from $\mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h}$ after it starts with execution (transmission).

  Formally, a state $q = (\cdot, v, s, \cdot, \theta)$ of P violates *time safety on* (c, h) if there exists a task or message $x \in \mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h}$ such that either
  - v_h has a successor v'_h with $\kappa(v_h, v'_h) = \texttt{call}$ and $\lambda(v_h, v'_h) = d$ [E code driver], or
  - there exists a thread $(u, \cdot) \in \theta_h$ with $\rho(u) = \nabla$, u has a successor u', $\kappa(u, u') = \texttt{call}$ and $\lambda(u, u') = d$ [S code driver],

  and one of the following
  TS1: $\mathit{In}[d] \cap \mathit{Out}[x] \neq \emptyset$ and $s(x) \neq \perp$, or
  TS2: $\mathit{Out}[d] \cap \mathit{In}[x] \neq \emptyset$ and $s(x) \neq 0$,
  We say that $(P_{c,h}, w_{c,h})$ is *time-safe* if for all $w_{c,h}$-traces ψ of $P = \{P_{c,h}\}$ no state of ψ violates time safety on (c, h).

Finally, we define type compliance and time safety for entire G-generated distibuted SCC program P:

– We say that (P, w) type-complies to T if $(P_{c,h}, w_{c,h})$ type-complies with $T_{c,h}$ for each $c \in C$ and each $h \in H$.
– We say that (P, w) is time-safe if $(P_{c,h}, w_{c,h})$ is time-safe for each $c \in C$ and each $h \in H$.

The compositional nature of these definitions shows that if, for some $c \in C$ and $h \in H$, only $P_{c,h}$ is modified then for the compliance with the type T and time safety of P it is sufficient to check if $(P_{c,h}, w_{c,h})$ type-complies and data-complies with $T_{c,h}$ and if it is time-safe.

Let r^G_σ and r^P_σ be port valuation functions at time $\sigma \in \mathbb{N}_0$ for G and P respectively. A trace of P and a trace of G are *input-compatible* (*output-compatible*) if they have the same sensor (actuator) port values at the same times, i.e. if $r^G_\sigma(p) = r^P_\sigma(p)$ for each sensor port $p \in \mathit{SensePorts}$ (each actuator port $p \in \mathit{ActPorts}$) and each time instant $\sigma \in \mathbb{N}_0$. The G-generated distributed SCC program (P,w) *implements* the Giotto program G if for any w-trace of P and any trace of G, input-compatibility implies output-compatibility.

Theorem 1 *Let G be a Giotto program and (P,w) be the distributed SCC program G-generated according to Algorithm 2. If T is a feasible type for G and (P,w) type-complies to T and is time-safe, then (P,w) implements G.*

Proof. Note first that the *RS1* resource sharing property of T and type compliance property of (P,w) ensure that for each state of (P,w) and each host $h \in H$ there exists at most one thread $(u, \cdot)$ in θ_h such that $\rho(u) = \texttt{dispatch}$. Also, the *RS2* resource sharing property of T and type compliance property of (P,w) ensure that for each state of (P,w) there exists at most one thread $(u, \cdot)$ in $\bigcup_{h \in H} \theta_h$ such that $\rho(u) = \texttt{dispatch}$ and $\nu(u) \in \mathit{Msgs}$. So, if T is feasible and (P,w) type-complies to T then there are no resource sharing conflicts.

We prove the input-output equivalence of the two programs under the type compliance and time safety assumptions. We first show that traces of G and P match on task output port values.

Lemma 1 *If $p \in \mathit{OutPorts}$, $h \in \mathit{sendToHosts}(p) \cup \{\bar{h}(p)\}$, then $r^G_\sigma(p) = r^P_\sigma(p_h)$ for any time $\sigma \in \mathbb{N}_0$.*

Proof (Lemma). We use induction on time σ. For time $\sigma = 0$ the statement holds because according to lines 1 and 2 of the Algorithm 1 the initialization driver $\mathit{init}[p]$ is called on $\bar{h}(p)$ and $\mathit{init}[p_h]$ is called on all $h \in \mathit{sendToHosts}(p)$ (E transitions with `call` instructions). They set p and p_h to initial $r^G_0(p)$ value.

Since $p \in \mathit{OutPorts}$ there exists a task t such that $p \in \mathit{Out}[t]$ and $t \in \mathit{Tasks}_{c,\bar{h}(p)}$ for some $c \in C$. In the code generated by the Algorithm 2 (line 4) the global copy $r^P(p_h)$ of the task output port p on host h is updated only by the invocation of the driver $\mathit{copy}[p_h]$ (`call` E transition) if $t \in \mathit{taskOutputPorts}(m,k)$ for a mode m and a unit k, i.e. when task t logically completes. Note that according to the Giotto semantics $r^G(p)$ is also updated only if $t \in \mathit{taskOutputPorts}(m,k)$, so we only have to prove that $r^P(p_h)$ is modified with a correct value.

Let σ be any time instant at which `call`($\mathit{copy}[p_h]$) instruction is executed, i.e. for which $t \in \mathit{taskOutputPorts}(m,k)$ for some mode $m \in \mathit{Modes}$ and unit k of m. Assume that lemma holds for all integers less than σ.

1. $h = \bar{h}(p)$:
   Let σ' be the last time instant task t was released before σ. Let the mode

and the unit of the corresponding `schedule` E transition be m' and k' respectively, $t \in \mathit{taskOutputPorts}(m', k')$. Let d be the task t input driver, i.e. $(\cdot, t, d) \in \mathit{Invokes}[m']$, and let p' be an input port of d, $p' \in \mathit{Out}[d]$. By the definition of the *sendToHosts* operator we have $h \in \mathit{sendToHosts}(p') \cup \{\bar{h}(p')\}$.

- If $p' \in \mathit{OutPorts}$ by induction hypothesis we also have $r^G_{\sigma'}(p') = r^P_{\sigma'}(p'_h)$.
- If $p' \in \mathit{SensePorts}$ and $\bar{h}(p') = h$ the port p' is updated on the host h at time σ' by execution of $\mathit{dev}[p']$ driver (line 40, `call` E transition) and by input-compatibility assumption we have $r^G_{\sigma'}(p') = r^P_{\sigma'}(p'_h) = r^P_{\sigma'}(p')$.
- Let $p' \in \mathit{SensePorts}$ and $h \in \mathit{sendToHosts}(p')$. According to the Algorithm 2 (line 40) and input-compatibility the driver $\mathit{dev}[p']$ is invoked at the unit k' on the host $\bar{h}(p')$ and the message $\mu(p')$ with the port p' value $r^G_{\sigma'}(p')$ is released (line 41, `schedule` E transition). If the program (P,w) is time-safe, then $(P_{c,h},w)$ is also time-safe. Therefore, the message transmission completes before time $\sigma' + \epsilon$ because at this time instant driver d is called and *TS1* should not hold. By assumption, the *DR1* data reception property is satisfied throughout the message transmission, so the message completion S transition correctly updates the port p'_h.

So, for all $p' \in \mathit{Out}[d]$ we have $r^G_{\sigma'}(p') = r^P_{\sigma'+\epsilon}(p')$. We assume that time ϵ is less than a time step $\gamma[m']$ so the message transmission is completed before any potential mode switch from mode m'. If the time safety property is satisfied the task t is dispatched after $\sigma' + \epsilon$ (*TS2* does not hold), but completed (completion S transition) by time $\sigma - \epsilon$ (*TS1* does not hold) at which the local copy of p is updated. So, $r^G_\sigma(p) = r^P_\sigma(p)$ for all $p \in \mathit{Out}[t]$. Since $h = \bar{h}(p)$ we have $r^G_\sigma(p) = r^P_\sigma(p_h)$.

2. $h \in \mathit{sendToHosts}(p)$:
By the similar argument as above it can be proved that $r^G_\sigma(p) = r^P_\sigma(p)$. According to the Algorithm 2 (line 48) on the host $\bar{h}(p)$ the message with the port p value $r^G_\sigma(p)$ is released (`schedule` E transition). Again, time safety and *DR2* data reseption properties ensure that the message is transmitted to the host h after the task t completes but before time σ. Since $h \in \mathit{sendToHosts}(p)$ the driver $\mathit{copy}[p_h]$ is invoked on the host h at time σ and we have $r^G_\sigma(p) = r^P_\sigma(p_h)$.

So, if the programs G and (P,w) are input-compatible the lemma above holds. To prove the output-compatibility of the two programs consider a port $p \in \mathit{ActPorts}$ and let $h = \bar{h}(p)$. The code in P generated by the Algorithm 2 updates p in mode m at unit k only if $p \in \mathit{actuatorPorts}(m, k)$ (line 5). The same is true for the execution of the Giotto program G. Let d be an actuator driver such that $p \in \mathit{Out}[d]$. Since each driver input port $p' \in \mathit{In}[d]$ is also in the set of task output ports *OutPorts* and since by the definition of the *sendToHosts* operator $h \in \mathit{sendToHosts}(p') \cup \{\bar{h}(p')\}$ by the lemma we have $r^G_\sigma(p') = r^P_\sigma(p'_h)$. After applying driver function $\mathit{drv}[d]$ on $\mathit{Out}[d]$, which updates p on h, we have $r^G_\sigma(p) = r^P_\sigma(p_h) = r^P_\sigma(p)$.

5.4 Checking Type Compliance and Time Safety

The paper [3] discusses time safety checking only for single mode Giotto programs without communication. These results are here generalized both for distributed and multiple-mode setting. For single mode Giotto program we give an efficent algorithm that checks if $P_{c,h}$ complies to a given type and if it is time-safe. For multiple-mode programs we give only a sufficient condition that can efficiently be checked.

Let $T = \{T_{c,h} \mid c \in C, h \in H\}$ be a feasible type for a Giotto program G with M modes and $P = \{P_{c,h} \mid c \in C, h \in H\}$ be distributed SCC program G-generated by the Algorithm 2. Let w be worst case execution function. Finally, let $g_{c,h}$ be the size of the part of G allocated to contractor c and host h and let all numbers in G be bounded by n.

Given a G-generated SCC program $P_{c,h} = (\mathcal{E}_{c,h}, \mathcal{S}_{c,h}, \Phi_{c,h})$, with $\mathcal{E}_{c,h} = (V^{\mathcal{E}}_{c,h}, E^{\mathcal{E}}_{c,h}, \kappa, \lambda, \eta)$ and $\mathcal{S}_{c,h} = (V^{\mathcal{S}}_{c,h}, E^{\mathcal{S}}_{c,h}, \cdot, \cdot, \cdot, \cdot)$ we first define a directed graph $\mathcal{P}_{c,h} = (V^{P}_{c,h}, E^{P}_{c,h})$ with $V^{P}_{c,h} = V^{\mathcal{E}}_{c,h} \cup V^{\mathcal{S}}_{c,h}$ and $E^{P}_{c,h} = E^{\mathcal{E}}_{c,h} \cup E^{\mathcal{S}}_{c,h} \cup E^{\mathcal{E},\mathcal{S}}_{c,h} \cup E^{\mathcal{S},\mathcal{E}}_{c,h}$, where $E^{\mathcal{E},\mathcal{S}}_{c,h}$ contains edges connecting a sink of $V^{\mathcal{E}}_{c,h}$ with a source of $V^{\mathcal{S}}_{c,h}$ and $E^{\mathcal{S},\mathcal{E}}_{c,h}$ edges connecting a sink of $V^{\mathcal{S}}_{c,h}$ with a source of $V^{\mathcal{E}}_{c,h}$. In particular, if $(v_1', v_1) \in E^{\mathcal{E}}_{c,h}$ such that $\kappa(v_1', v_1) = \texttt{future}$ and $\lambda(v_1', v_1) = (\cdot, v_2)$ then v_1 is a sink and v_2 is a source of $V^{\mathcal{E}}_{c,h}$. Let u_1 be a source of $V^{\mathcal{S}}_{c,h}$ such that $u_1 = \Phi(v_1')$ and let $u_2 \in V^{\mathcal{S}}_{c,h}$ be the sink node of the chain for which u_1 is the source. If $\nu(v_2) = (\cdot, k)$ such that $k \neq 0$ then $(v_1, u_1) \in E^{\mathcal{E},\mathcal{S}}_{c,h}$ and $(u_2, v_2) \in E^{\mathcal{S},\mathcal{E}}_{c,h}$. The Figure 8 is related to graphs $\mathcal{P}_{c,h}$ for the multiple-mode Giotto program G_2. It shows graph $\mathcal{P}$ in which each edge abstracts a chain of $O(g_{c,h})$ edges found in each $\mathcal{P}_{c,h}$ graph.

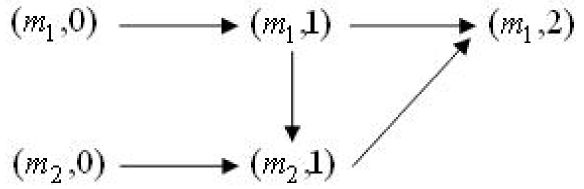
$$\begin{tikzcd}
(m_1,0) \arrow[r] & (m_1,1) \arrow[r] \arrow[d] & (m_1,2) \\
(m_2,0) \arrow[r] & (m_2,1) \arrow[ur] &
\end{tikzcd}$$

Fig. 8. Abstracted transition graph $\mathcal{P}$ for G_2

Lemma 2 *The graph $\mathcal{P}_{c,h}$ is an acyclic graph.*

Proof. As discussed with respect to the Algorithm 2 (line 25) and as defined by the Giotto semantics in [1], for a mode switch the compiler computes the unit of the destination mode as close as possible to the end of the mode's period. This means that the time until the end cannot increase when mode switch is performed. Since there can be no multiple switches at the same time instant, i.e.

in each visited mode time has to progress for some nonzero time, this actually means that time until the end of target mode's period has to decrease. Therefore, if there was a mode switch from mode m at unit k_1 and at some later instant the program performs another mode switch now to the mode m at unit k_2 then $k_1 < k_2$. Note also that in constructing $E^P_{c,h}$ we ignore mode switches with unit zero of target mode. This is because at such mode switch there will be no active task that already executed for some time and further behavior is as if the program started its execution at that time instant. The last two conclusions together show that $\mathcal{P}_{c,h}$ is an acyclic directed graph.

We next construct a state transition graph by annotating each node v of the graph $\mathcal{P}_{c,h}$ with a particular state $q_v = (r, v, s, \tau, \theta)$ of the SCC program $P_{c,h}$. The port valuation component r is completely ignored. The graph $\mathcal{P}_{c,h}$ is acyclic, so its nodes can be sorted and processed in topological order. Each source node v of $\mathcal{P}_{c,h}$, and for each mode there is exactly one such node, is annotated with state $(\cdot, v, s_0, \emptyset, \emptyset)$, where s_0 is a function that maps each $x \in \mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h}$ to 0. For other nodes of $\mathcal{P}_{c,h}$ we proceed by transforming the state of its immediate predecessors. We do so by performing one or more transition steps defined by semantics of an SCC program. Task execution time nondeterminism in time transition steps is eliminated by assuming that each task (or message) $x \in \mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h}$ completes exactly after time given by the worst-case execution time $w_{c,h}(x)$. From the properties of G-generated SCC program it can easily be shown that if a node $v \in \mathcal{P}_{c,h}$ has more than one predecessor then each predecessor v' is annotated with a state $q_{v'} = (\cdot, v', s_{v'}, \emptyset, \emptyset)$. Let the relation "$\leq$" on set $\mathbb{N}_0 \cup \perp$ be defined by extending the order relation on $\mathbb{N}_0$ with $a \leq \perp$ for all $a \in \mathbb{N}_0$ (recall that $s(x) = \perp$ means that task x completed execution). Then $q_v = (\cdot, v, s_v, \emptyset, \emptyset)$ where, for each $x \in \mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h}$, $s_v(x)$ is the least element (with respect to relation $\leq$) of the set $\{s_{v'}(x) \mid (v', v) \in E^{\mathcal{P}}_{c,h}\}$. So, for the nodes with more than one incoming edge we compute the amount of time tasks executed pointwise and conservatively.

We say that the state transition graph $\mathcal{P}_{c,h}$ type-complies with $T_{c,h}$ (is time-safe) if none of its states violates type-compliance with $T_{c,h}$ (time-safety). We now give the sufficent condition for a distributed SCC program to be time-safe and type-compliant.

Lemma 3 *If state transition graph $\mathcal{P}_{c,h}$ type-complies with $T_{c,h}$ and if it is time-safe then the G-generated SCC program $(P_{c,h}, w_{c,h})$ type-complies with $T_{c,h}$ and is time-safe.*

Proof. Let $q = (\cdot, v, s, \tau, \{(u, \delta)\})$ be an arbitrary state of an $w_{c,h}$-trace ψ of $P_{c,h}$. There exists a node in $\mathcal{P}_{c,h}$ with a state $q' = (\cdot, v, s', \tau, \{(u', \delta)\})$, i.e. q and q' match except on task status functions s and s' and S program control nodes u and u'. We show that $s'(x) \leq s(x)$ for each $x \in \mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h}$. Note that on the trace ψ tasks and messages may complete in time less than the one given by $w_{c,h}$. Assume first that the trace ψ executes in a single mode. If along ψ we have $s'(x) = s(x)$ for some state $q = (\cdot, v, s, \tau, \{(u, \delta)\})$ then $s'(x)$ is incremented in a time transition if $\rho(u) = \texttt{dispatch}$ and $\nu(u) = x$. But in that case $s(x)$ is also incremented or it becomes $\perp$ (in which case x executed for less than $w_{c,h}(x)$). Since $a \leq \perp$ for any $a \in \mathbb{N}_0$ we still have $s'(x) \leq s(x)$ after transition is taken. We can conclude the same for the case when trace ψ consists of states from different modes by noting that for mode switches in the graph $\mathcal{P}_{c,h}$ the status function s is determined by taking the least value on all incoming edges of a node. From the fact that $s'(x) \leq s(x)$ we directly have that if q' is time-safe then q is also time-safe. This fact also shows that if $s(x)$ is incremented by a time transition at a state q then there must be some $x' \in \mathit{Tasks}_{c,h} \cup \mathit{Msgs}_{c,h}$ such that $s(x')$ is incremented by the time transition at a state q'. So, if q' type-complies with $T_{c,h}$ then q also type-complies with $T_{c,h}$.

The size of $\mathcal{P}_{c,h} = (V^P_{c,h}, E^P_{c,h})$ is $O(g_{c,h} \cdot n)$ because, as we recall from the section 4.1, both $(V^{\mathcal{E}}_{c,h}, E^{\mathcal{E}}_{c,h})$ and $(V^{\mathcal{S}}_{c,h}, E^{\mathcal{S}}_{c,h})$ are of the same size. If node $v \in \mathcal{P}_{c,h}$ has only one predecessor then computing its state takes constant time. Otherwise, if v has l predecessors then it takes $O(g_{c,h} \cdot l)$ time. However, since there are $O(M^2 \cdot n) = O(n)$ predecessors for all such nodes v, we can compute their states in total time $O(g_{c,h} \cdot n)$. Therefore, constructing transition graph $\mathcal{P}_{c,h}$ and annotating it with states can be done in $O(g_{c,h} \cdot n)$ time. Finally, each state of $\mathcal{P}_{c,h}$ can be checked for type compliance and time safety in constant time (we assume that a driver can access at most a constant number of tasks). Therefore, we have the following lemma:

Lemma 4 *The state transition graph $\mathcal{P}_{c,h}$ can be constructed and checked if it type-complies with $T_{c,h}$ and if it is time-safe in time $O(g_{c,h} \cdot n)$.*

The Lemma 3 gives only a sufficient condition. If state transition graph $\mathcal{P}_{c,h}$ does not type-comply with $T_{c,h}$ (is not time-safe) then, for general Giotto program G, we can not conclude that SCC program $(P_{c,h}, w_{c,h})$ does not type-comply with $T_{c,h}$ (is not time-safe). This is so because in the state construction of $\mathcal{P}_{c,h}$ different incoming edges of a node may impose conservative approximation on different tasks. However, for some special cases this condition is also a necessary one. If G is a single mode Giotto program then the state transition graph $\mathcal{P}_{c,h}$ for such a program consists of disconnected chains. So, if $\mathcal{P}_{c,h}$ does not type-comply or is not time-safe at some state q then the trace along the chain up to q is a counterexample. From the previous lemmas we directly have the following theorem:

Theorem 2 *Let G be a single-mode Giotto program and let g be the total size of G. It can be checked in time $O(g_{c,h} \cdot n)$ if $(P_{c,h}, w_{c,h})$ type-complies with $T_{c,h}$ and if it is time-safe. It can be checked in time $O(g \cdot n)$ if (P, w) implements G. If only $(P_{c,h}, w_{c,h})$ is modified then it can be checked in time $O(g_{c,h} \cdot n)$ if (P, w) still implements G.*

References

1. T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A Time-Triggered Language for Embedded Programming In *Proc. IEEE*, Vol. 91, pp. 84-99, 2003.

2. T.A. Henzinger, C.M. Kirsch, R. Majumdar, and S. Matic. Time safety checking for embedded programs. In *Embedded Software*, LNCS 2491, pp. 76-90. Springer, 2002.
3. T. A. Henzinger, C. M. Kirsch, and S. Matic. Schedule Carrying Code In *Embedded Software*, LNCS 2855, pp. 241-256. Springer, 2003.
4. T. A. Henzinger and C. M. Kirsch. The Embedded Machine: Predictable, Portable Real-time Code In *Proc. PLDI*, pp. 315-326, ACM Press, 2002.
5. H. Kopetz and N. Suri. Compositional Design of RT Systems: A Conceptual Basis for Specification of Linking Interfaces In *Proc. ISORC*, pp. 51-60, IEEE, 2003.
6. P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, P. Niebert and T. Le Sergent. From Simulink1 to SCADE/Lustre to TTA: a layered approach for distributed embedded applications In *Proc. LCTES*, pp. 153-162, ACM Press, 2003.