Composable Code Generation for Distributed Giotto [*]

Thomas A. Henzinger
EPFL and UC Berkeley
tah@epfl.ch

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

Slobodan Matic
University of California, Berkeley
matic@eecs.berkeley.edu

Abstract. We present a compositional approach to the implementation of hard real-time software running on a distributed platform. We explain how several code suppliers, coordinated by a system integrator, can independently generate different parts of the distributed software. The task structure, interaction, and timing is specified as a Giotto program. Each supplier is given a part of the Giotto program and a timing interface, from which the supplier generates task and scheduling code. The integrator then checks, individually for each supplier, in pseudo-polynomial time, if the supplied code meets its timing specification. If all checks succeed, then the supplied software parts are guaranteed to work together and implement the original Giotto program. The feasibility of the approach is demonstrated by a prototype implementation.

Categories and Subject Descriptors C.3 [*Special-purpose and Application-based Systems*]: Real-time and Embedded Systems; D.1.3 [*Software Techniques*]: Distributed Programming

General Terms Languages, Reliability

Keywords Real Time, Distributed Compilation

1 Introduction

The distributed implementation of hard real-time systems is a key challenge in modern control systems, especially in automobile (drive-by-wire) and aircraft (fly-by-wire) control. Much of the work in this area has been devoted to hardware-focused solutions, such as the time-triggered architecture [1], which guarantees hard real-time constraints across a distributed system by strict adherence to clock-synchronized networking protocols. The cost of such a solution is paid in terms of flexibility, and even recent efforts in the automotive industry (FlexRay, Autosar [2]) require that all component processes, their dependencies, and their timing profiles be known in advance. We suggest that the competing goals of *timely execution* and *composable design* can be achieved together by adopting a software solution that requires only basic hardware services such as clock synchronization and redundancy management. We have previously proposed the LET (*logical execution time*) paradigm, and the LET-based language Giotto, as a software model that guarantees predictable real-time execution and at the same time supports portable, composable code [3]. In this paper, we demonstrate how Giotto can be implemented on a distributed platform by distributed compilation with little global coordination. In this way, Giotto offers a framework for the compositional design of hard real-time systems.

Giotto. Giotto is a domain-specific language for control applications [3]. A Giotto program executes a periodic set of LET tasks, and the set of tasks, or their periods, may change whenever a Giotto mode switch occurs. Instead of just a deadline, a LET task has a *release* and a *termination time*: the release time specifies the exact time at which the task inputs are made available to the task; the termination time specifies when the task outputs become available to other tasks. The task must start running, may be preempted, and must complete execution during its LET, which is the time from release to termination. Thus the times when a LET task reads and writes data are decoupled from the task execution. LET avoids race conditions, and thus ensures the predictable, deterministic execution of a set of real-time tasks. LET tasks can be replaced and composed without modifying their behavior or timing. Since LET is an abstract programming model, the compiler must ensure that the generated code satisfies the LET assumption. This can be achieved by compiling Giotto into *schedule-carrying code* (SCC) [4] for a pair of virtual machines: the E (embedded) machine mediates between tasks and the physical environment [5]; the S (scheduling) machine mediates between tasks and the CPU [4]. E code specifies when sensors and task inputs are read, and when actuators and task outputs are written; S code specifies when a task is executed on the CPU. We have implemented the E and S machine as part of a high-performance microkernel for real-time systems [6] and used Giotto to implement the flight-control system for a model helicopter [7].

Distributed hard real-time code. A Giotto program specifies the functional and timing behavior of a dynamic set of tasks, for example, the tasks of an automotive control system. Such a system is typically executed by an on-board network with several hosts (CPUs). Moreover, such a system is typically put together from several parts, which correspond to different control problems, for example, fuel injection and anti-lock brake control. While the different software parts may interact, they are often developed by different *suppliers*: the brake supplier will deliver its own software, etc. Furthermore, to optimize the use of computational resources, there need not be a one-to-one correspondence between hosts and suppliers. The contracting company, or *integrator* (e.g., the car manufacturer), then faces the challenge of putting together and maintaining the entire system. Using today's methodologies, a simple modification in the software of a single supplier may induce a series of modifications in the whole system. For example, a change of timing attributes (e.g., task execution times) in one software component may cause the schedule of other components to change. We show how this problem can be avoided using Giotto.

Our approach. We view the Giotto program as the overall system specification (timing and task interaction). Each supplier is

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

LCTES'05, June 15–17, 2005, Chicago, Illinois, USA.

given a part of the Giotto program with the charge to implement the corresponding tasks. This information can be regarded as a *component specification*. So that all supplied software parts will fit together, each supplier also receives timing information in the form of a *timing interface*. The timing interface specifies the time slots that can be used by the supplier for computation on the hosts, and the time slots that can be used by the supplier for communication over the network. From a component specification and a timing interface each supplier produces code. The integrator then checks that the produced code complies to the timing interface and meets, on the given hardware, the release and termination times specified by the Giotto program. The first check is called *interface compliance*; the second, *time safety*. Both checks are local for each piece of supplied code and can be performed in pseudo-polynomial time. If all checks go through, the integrator is assured that all supplied software parts fit together and correctly implement the original Giotto program (note that correctness includes the satisfaction of all real-time constraints).

Why does this work? Essentially, we build a fully software-based instance of the time-triggered paradigm. Instead of having the hardware and network protocol enforce all timing interfaces, each timing interface is enforced separately by the compiler (during distributed code generation by the suppliers) and by program analysis (during code integration by the integrator). The LET assumption is crucial to this approach. The LET (release to termination) of a task is always non-zero. This allows us to communicate values across the network without changing the timing of a task, and without introducing nondeterminism, as long as the timing interface ensures that all values are available in time to meet all task release and termination times, and all sensor read and actuator update times. By contrast, the synchrony assumption used by other real-time languages [8] does not offer this flexibility, and hence an important approach to distributing synchronous programs is based on the Globally Asynchronous, Locally Synchronous paradigm [9].

What are the benefits? We obtain the benefits of the time-triggered paradigm in terms of real-time assurance, and at the same time achieve a high degree of flexibility. For example, a supplier may be replaced by another one, and as long as the code produced by the new supplier complies to its component specification and timing interface, it will work together properly with all other code in the system. Likewise, if new functionality is added to the system, say by adding a new supplier, as long as the new software passes the two checks (interface compliance and time safety), it will not change the behavior (neither functionality nor timing) of the original system in any way. This is because interface compliance succeeds only if the original set of timing interfaces can accommodate an additional timing interface with sufficient capacity, and time safety succeeds only if the original set of hosts can accommodate the new tasks. The advantage of our approach lies in the fact that the two checks can be performed automatically, and the system integrator need not rely exclusively on testing to see if the upgraded system behaves correctly.

Related work. Previously, Giotto had only been compiled for single-CPU systems [10]. The contribution of this paper is two-fold: we describe a methodology that supports (1) *distributed* real-time code generation for (2) *distributed* real-time systems. Multiple suppliers (1) can independently compile different parts of a Giotto program to run on a system of multiple CPUs (2). Because of the time-driven nature of our timing interfaces, (1) immediately enables (2) on clock-synchronized systems. Other approaches for (2), however, may not necessarily support (1); for example, synchronous reactive programs written in Lustre have been compiled globally for distributed real-time systems [11]. Aimed at (1) are scheduling techniques that address the problem of dividing tasks into groups, and scheduling tasks within groups [12, 13]: the challenge is to develop compositional schemes for resource partitioning such that each task group may be programmed as if it had dedicated access to the resource and may be tested for schedulability without global task knowledge. However, these techniques typically assume a single CPU and no interaction between tasks. In distributed real-time systems there are efforts [14] to define minimal but complete interfaces that link components together. In avionics software, where previously each control subsystem had its own dedicated resource, new solutions are proposed which offer a common computing platform for multiple functions; [15] presents requirements for the temporal partitioning of such a platform. The car manufacturers' and suppliers' perspectives on embedded software reuse are described in [16], which presents a general framework in which different software components can be classified according to their degree of reusability, albeit without considering real-time communication in detail.

Outline of the paper. In Section 2, we present a brief review of Giotto and introduce a running example that we will use throughout this paper. In Section 3, we discuss the algorithm that generates from a given Giotto program virtual machine code (SCC) for each host and each supplier. In Section 4, we introduce timing interfaces and show how they can be composed. Section 5 describes our prototype implementation of distributed Giotto. In Section 6, we analyze distributed SCC generated from Giotto, present pseudo-polynomial checks for interface compliance (w.r.t. a timing interface) and time safety (w.r.t. the worst-case execution times of tasks), and prove the distributed Giotto compiler correct.

2 The Giotto Language

We give a brief introduction to Giotto and refer to [3] for details. A simple example of a Giotto program G_A is shown in Fig. 1. For now ignore the distribution annotations given in the brackets to the right of the program. In this audio application a prerecorded PCM-format audio file is read, processed, analyzed, and reproduced by three real-time tasks. The $Generator$ task synthesizes the digital audio samples of the sound that resembles the plucking of a string. This is done according to the Karplus-Strong algorithm [17], where the period of the task determines the pitch of the generated sound. The $Mixer$ task merges the file samples with the synthesized samples amplifying the string pluck sound. The $Analyzer$ task computes a short-time Fourier series of the mix sound.

A Giotto program begins with port declarations. A port is a typed variable. The set $Ports$ is partitioned into the following four sets: a set $SensePorts$ of sensor ports, a set $ActPorts$ of actuator ports, a set $InPorts$ of task input ports, and a set $OutPorts$ of task output ports. The sensor ports include the integer-typed port p_c, a discrete clock. In Fig. 1 the sensor port $AudioSampler$ represents a vector of audio file samples, the actuator port $MixPlayer$ a vector of final waveform samples, and the task output ports $Spectrum$, $MixSound$, and $StringSound$, respectively, represent vectors of Fourier coefficients, mix samples, and string samples. The Fig. 2 shows the data dependency graph for the tasks (rectangles with rounded corners), the sensor, and the actuator. Each sensor (resp. actuator) port p is read (resp. written) by a device driver $dev[p]$. Each task output port is double-buffered, i.e., 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 including other tasks. The copy driver $copy[p]$ copies data from the local copy to the global copy of the task output port p.

Giotto has two kinds of computational activities, tasks and drivers. Tasks are released and their execution take time, while drivers are executed in logically zero time. A Giotto task t has a set $In[t] \subseteq InPorts$ of input ports, a set $Out[t] \subseteq OutPorts$ of output ports, and a task function $task[t]$ from the input to the output ports. The task function represents the result of the computational

```
sensor
 AudioSampler uses dev[AudioSampler];                    [s1, h1]
actuator
 MixPlayer uses dev[MixPlayer];                          [s1, h1]
output
 Spectrum uses copy[Spectrum];                           [s1, h1]
 MixSound uses copy[MixSound];                           [s2, h2]
 StringSound uses copy[StringSound];                     [s3, h2]
task
 Analyzer(In1) output(Spectrum);
 Mixer(In2) output(MixSound);
 Generator(In3) output(StringSound);
driver
 InDrv1(MixSound) output(In1);
 InDrv2(AudioSampler, StringSound) output(In2);
 InDrv3() output(In3);
 ActDrv(MixSound) output(MixPlayer);
start m1 {
 mode m1() period 8 {
  actfreq 2 do MixPlayer(ActDrv);
  taskfreq 1 do Analyzer(InDrv1);
  taskfreq 2 do Mixer(InDrv2);
  taskfreq 1 do Generator(InDrv3); }
}
```

Figure 1. Audio mixer Giotto program G_A

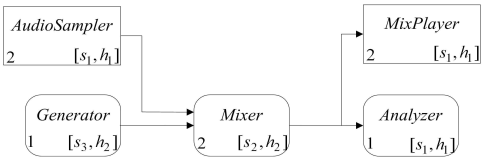

Figure 2. Data dependency graph for the program G_A

activity performed by the task. For example, the task $Mixer$ is defined with input port In_2, output port $MixSound$, and task function $task[Mixer]$. In addition to the device and copy drivers described above, drivers can be used to transport data between ports and to initiate mode changes. A Giotto driver d has a set $Src[d] \subseteq Ports$ of source ports, a set $Dst[d] \subseteq Ports$ of destination ports, a driver function $drv[d]$ from the source to the destination ports, and an optional boolean condition on the source ports to control mode switching. For instance, $AudioSampler$ and $StringSound$ are the source ports and In_2 is the destination port of the driver $InDrv_2$. Let $Tasks$ (resp. $Drvs$) be the set of tasks (resp. drivers).

A Giotto program is defined with a set of modes, each of which consists of a set of periodic tasks. In each mode the invocation of tasks is repeated after a fixed amount of time we call the mode period. The task set can change at transitions (switches) from one mode to another. Let $Modes$ be the set of modes, containing a start mode $start \in Modes$. A Giotto mode m has a period $\pi[m] \in \mathbb{N}_{>0}$, a set of task invocations, a set of actuator updates, and a set of mode switches. Each task invocation (ω_{task}, t, d) consists of a task frequency $\omega_{task} \in \mathbb{N}_{>0}$ relative to the mode period, a task t, and a task input driver d, which loads the task inputs. In our example there is only one mode m_1 with the period $\pi[m_1] = 8$ time units, in this case milliseconds. The audio file is discretized at the rate of $11Khz$, and 44 of its samples are read every $4ms$. The mix sound is also processed with the period of $4ms$, so the frequency of the $Mixer$ task is 2, and one of the three task invocations of mode m_1 is $(2, Mixer, InDrv_2)$. The LET character of the $Mixer$ task implies that, even if it completes earlier, its output $MixSound$ is made available through the $copy[MixSound]$ driver exactly at $4ms$. Each actuator update (ω_{act}, d) consists of an actuator frequency $\omega_{act} \in \mathbb{N}_{>0}$, and an actuator driver d. Each mode switch (ω_{switch}, m', d) consists of a switch frequency $\omega_{switch} \in \mathbb{N}_{>0}$, a target mode m', and a mode driver d which uses the boolean condition on its source ports to control the mode switch. For the single mode m_1 of the example, we have one actuator update $(2, ActDrv)$ and no mode switches. In the rest of the paper we will refer to the single-mode program in Fig. 1. However, if, for instance, we want to be able to switch to a mode m_2 in which task $Mixer$ is executed twice as fast, i.e. with ω_{task}=4, the program G_A should also contain code for m_2 shown in Fig. 3.

```
mode m2() period 8 {
 exitfreq 4 do m1(ModeDrv2);
 actfreq 4 do MixPlayer(ActDrv);
 taskfreq 1 do Analyzer(InDrv1);
 taskfreq 4 do Mixer(InDrv2);
 taskfreq 1 do Generator(InDrv3); }
```

Figure 3. Additional mode for the Giotto program G_A

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, denoted $\omega_{max}[m]$. The duration of a unit is $\gamma[m] = \pi[m]/\omega_{max}[m]$. For the compilation procedure we need the following sets which can, given a mode m and an integer unit $0 \le k < \omega_{max}[m]$, be directly determined from the Giotto program. The set $taskInvocations(m, k)$ contains all task invocations of mode m that are released at unit k, i.e., for which $k \cdot \gamma[m]$ is an integer multiple of $\pi[m]/\omega_{task}$. For instance, $\gamma[m_1] = 4$ and $taskInvocations(m_1, 1) = \{(2, Mixer, InDrv_2)\}$, because the $Mixer$ task is the only task that is released at unit 1 of m_1, at $4ms$. An output port is in the set $taskOutPorts(m, k)$ if in mode m it is updated at unit k, i.e., if it is an output port of a task in $taskInvocations(m, k)$. A sensor port is in the set $senPorts(m, k)$ if in mode m it is read at unit k, i.e., if it is a source port of an input driver of a task in $taskInvocations(m, k)$. The set $actDrivers(m, k)$ contains all actuator drivers of mode m that are invoked at unit k. Finally, an actuator port is in the set $actPorts(m, k)$ if in mode m it is updated at unit k, i.e., if it is a destination port of a driver in $actDrivers(m, k)$. For instance, $senPorts(m_1, 1) = \{AudioSampler\}$ and $actPorts(m_1, 1) = \{MixPlayer\}$.

E code, S code, and schedule-carrying code (SCC). In [4] we presented the execution of a Giotto program on a single processor through the interpretation of code compiled for two virtual machines, *embedded* and *scheduling* machine. The embedded machine [5] handles sensors, actuators, and all task requests. It runs *E code* that specifies the timing and control flow of Giotto tasks and drivers. The embedded machine has three non-control-flow instructions. A `call`(d) instruction immediately invokes a driver d. A `release`(t) instruction[1] releases a task t and proceeds to the next E code instruction. A `future`(ℓ, a) instruction marks the E code at the address a for execution after ℓ ms elapse. The positive integer ℓ specifies a time trigger, the simplest and only form of trigger that we consider in this paper. In order to handle multiple active triggers, the embedded machine maintains a trigger queue. The Giotto compiler generates a block of E code instructions for each unit of each program mode.

For example, in Fig. 4, the block of E code for unit 0 of mode m_1 is identified by the label $E(m_1, 0)$. It initiates the execution of the copy drivers that update the three task output ports, and the execution of the audio player device driver. Then the audio sampler device driver and the three task input drivers update the input ports of the three tasks that are released next. Note the order of driver `call` instructions: copy drivers are followed by device drivers, followed by task input drivers. Finally, a time trigger with address label $E(m_1, 1)$ is activated. So, after $4ms$ the embedded machine executes the block of E code starting at the address $E(m_1, 1)$. The last instruction of this block activates another $4ms$ trigger, now with address $E(m_1, 0)$. In this way the execution of each

[1] The `release` instruction corresponds to the `schedule` instruction in [5], but has been renamed for clarity.

```
E(m1, 0):
  call(copy[Spectrum])
  call(copy[MixSound])
  call(copy[StringSound])
  call(ActDrv)
  call(dev[MixPlayer])
  call(dev[AudioSampler])
  call(InDrv1)
  call(InDrv2)
  call(InDrv3)
  release(Analyzer)
  release(Mixer)
  release(Generator)
  future(4, E(m1, 1))
```

```
E(m1, 1):
  call(copy[MixSound])
  call(ActDrv)
  call(dev[MixPlayer])
  call(dev[AudioSampler])
  call(InDrv2)
  release(Mixer)
  future(4, E(m1, 0))
```

Figure 4. E code blocks for the program G_A

```
S(m1, 0):
  dispatch(Mixer, 4)
  dispatch(Generator, 4)
  dispatch(Analyzer, 4)
```

```
S(m1, 1):
  dispatch(Mixer, 4)
  dispatch(Generator, 4)
  dispatch(Analyzer, 4)
```

Figure 5. S code blocks for the program G_A

of the two blocks is repeated every $8ms$. Note that the task and driver functions are external to the embedded machine and must be implemented in some other language.

The scheduling machine [4] determines when, and in what order, tasks released by the E code are executed (dispatched). It replaces the system task scheduler, since the code that it runs, *S code*, defines a schedule according to which, at run time, a simple dispatcher selects which task to execute. The scheduling machine also has three instructions, one of which is `call`(d) as for the embedded machine. A `dispatch`(t, ℓ) instruction resumes (or starts) the execution of a released task t until ℓ ms elapse, measured from the start instant of the current S code block. The integer ℓ specifies the simplest and the only form of *timeouts* that we consider in this paper. The task executes until either it completes or the timeout becomes true, whichever happens first, and after that the scheduling machine proceeds to the next instruction. An `idle`(ℓ) instruction causes the scheduling machine to idle until the timeout ℓ becomes true. Each block of E code is annotated with a block of S code which starts execution in a separate thread after the last instruction of the E code block. An important difference between E and S code is that each E code block executes instructions instantaneously, whereas each block of S code executes over time. We call the resulting code, consisting of both E and S code blocks, *schedule-carrying code* (SCC). The example S code in Fig. 5 contains a possible schedule for the Giotto program G_A. The block of S code at the label $S(m_1, 0)$ is interpreted after the block of E code at the label $E(m_1, 0)$. It starts with the execution of the $Mixer$ task followed by the other two tasks. The task executing at $4ms$ is suspended and resumed with the corresponding `dispatch` instruction in the $S(m_1, 1)$ block. We note that an S code instruction that dispatches a task not yet released is simply ignored. With the SCC code in Fig. 4 and 5 the $Mixer$ task is executed twice every $8ms$, and the tasks $Generator$ and $Analyzer$ once, exactly as specified by the Giotto program G_A.

3 Distributed Code Generation

In our distributed model the system *integrator* generates a Giotto program G to be implemented by a set S of *suppliers* on a set H of *hosts*. A supplier is an independent code developer. A host is a self-contained computational element with its own processor, memory, and communication interface. We assume that hosts are connected by a shared bus or a broadcast network. Hosts communicate by exchanging messages containing port values. For a port $p \in Ports$, let $\mu[p]$ be the message with the port p value.

The integrator assigns each task and each driver defined in G to a particular host and supplier. For a task $t \in Tasks$ let $\bar{h}(t)$ (resp. $\bar{s}(t)$) be the host (resp. supplier) which executes (resp. implements) task t. We similarly define $\bar{h}(d)$ and $\bar{s}(d)$ for a driver $d \in Drvs$. Let $Tasks_{s,h}$ (resp. $Drvs_{s,h}$) be the set of all tasks (resp. drivers) assigned to supplier s on host h. We require that a task and its input and copy drivers be assigned to the same supplier on the same host. Also, an actuator driver and the corresponding device driver must be assigned to the same supplier on the same host. With such an assignment the integrator also allocates each port of G to a particular host and supplier. If $p \in Ports$ is a sensor or an actuator port, then $\bar{s}(p) = \bar{s}(dev[p])$ and $\bar{h}(p) = \bar{h}(dev[p])$. If p is a task t input or output port, i.e., if $p \in In[t] \cup Out[t]$, then $\bar{s}(p) = \bar{s}(t)$ and $\bar{h}(p) = \bar{h}(t)$. Finally, each message $\mu[p]$ is associated with a supplier $\bar{s}(p)$ and host $\bar{h}(p)$, namely, the sending supplier and host. Let $Msgs_{s,h}$ be the set of all messages that are associated with supplier s on host h.

In the rest of the paper we assume that the example Giotto program G_A, a streaming audio application, is to be implemented by three suppliers on two hosts. In Fig. 1 each annotation given in brackets to the right of a port denotes the supplier and the host to which the port is allocated. The assignment for tasks is shown in Fig. 2. The audio file is read on host h_1, and every $4ms$ 44 of its samples are sent to host h_2 for processing. The $Mixer$ and $Generator$ tasks, implemented respectively by the suppliers s_2 and s_3, run on h_2. After receiving the samples from h_1, the task $Mixer$ merges them with the generated samples, and within the same $4ms$, the resulting $MixSound$ samples are sent back to host h_1. The final waveform is there reproduced and analyzed by the $Analyzer$ task implemented by supplier s_1. The sets of tasks, drivers, and messages that are associated, for instance, with s_2 on h_2 are $Tasks_{s_2,h_2} = \{Mixer\}$, $Drvs_{s_2,h_2} = \{InDrv_2, copy[MixSound]\}$, and $Msgs_{s_2,h_2} = \{\mu[MixSound]\}$.

For each supplier $s \in S$ and each host $h \in H$, the integrator gives out (see the next sections for formal definitions)

1. an E code module $\mathcal{E}_{s,h}$ that describes the timing and control flow of driver, task, and message invocations for supplier s on host h, and
2. a timing interface $T_{s,h}$ that specifies the computation and transmission time instants on host h that are available for supplier s.

Once a supplier s receives the E code module $\mathcal{E}_{s,h}$ and timing interface $T_{s,h}$ for host h it generates

1. an S code module $\mathcal{S}_{s,h}$ for host h,
2. functionality code for all tasks $Tasks_{s,h}$ and drivers $Drvs_{s,h}$ (sequential functions written in, e.g., native C code), and
3. worst-case execution (transmission) time estimates $w_{s,h}$ for the tasks in $Tasks_{s,h}$ (messages in $Msgs_{s,h}$).

Provided with the worst-case execution and transmission times the integrator then verifies each generated S code module against the corresponding timing interface and E code module. In this way the integrator can check the composability of all supplied S code modules and ensure that the resulting distributed SCC program satisfies the semantics (including the timing) of the original Giotto program G. Moreover, once a supplier modifies its S code module on a host it is sufficient to check whether the new module complies to its timing interface to preserve Giotto semantics.

Distributed Giotto compilation. Let P be the entire distributed SCC program. The set $Ports_P$ of distributed SCC ports contains additional ports ($Ports \subseteq Ports_P$) to store the data sent over the network. Namely, if according to the Giotto program G and port-to-host allocation a value of the port $p \in Ports$ is needed as input to a driver on a host h different from the originating host $\bar{h}(p)$, i.e., if a message with the value of p must be sent over the network, then the host h must keep its own copy p_h of port p. For a given port p, let the set $recHosts(p)$ be the set of hosts that need to receive

Algorithm 1 The distributed Giotto compiler (mode *m*)

```
    $k := 0$; $\gamma[m] := \pi[m]/\omega_{max}[m]$;
    **while** $k < \omega_{max}[m]$ **do**
      $\forall s \in S \,.\, \forall h \in H$: link $E_{s,h}(m, k)$ to next address of $\mathcal{E}_{s,h}$;
      $\forall p \in taskOutPorts(m, k).\forall h \in recHosts(p) \cup \{\bar{h}(p)\}.\forall s \in S$:
5:      $emit(s, h, \texttt{call}(copy[p_h]))$;
      $\forall d \in actDrivers(m, k)$:
        $emit(\bar{s}(d), \bar{h}(d), \texttt{call}(d))$;
      $\forall p \in actPorts(m, k)$:
        $emit(\bar{s}(p), \bar{h}(p), \texttt{call}(dev[p]))$;
10:   *Mode_Switch_Compilation_Algorithm* [10]
      $\forall p \in senPorts(m, k)$:
        $emit(\bar{s}(p), \bar{h}(p), \texttt{call}(dev[p]))$;
        **if** $recHosts(p) \neq \emptyset$ **then**
          $emit(\bar{s}(p), \bar{h}(p), \texttt{release}(\mu[p]; \epsilon))$;
15:   $\forall(\cdot, t, d) \in taskInvocations(m, k)$:
        $\epsilon_1 := 0$; $\epsilon_2 := 0$;
        **if** $Src[d] \cap senPorts(m, k) \neq \emptyset$ **then** $\epsilon_1 := \epsilon$;
        **if** $sendOutPorts(t) \neq \emptyset$ **then** $\epsilon_2 := \epsilon$;
        $emit(\bar{s}(t), \bar{h}(t), \texttt{release}(\epsilon_1; t; \epsilon_2))$;
20:     $\forall p \in sendOutPorts(t)$ :
          $emit(\bar{s}(t), \bar{h}(t), \texttt{release}(\epsilon; \mu[p]))$;
      $\forall s \in S \,.\, \forall h \in H$:
        $emit(s, h, \texttt{future}(\gamma[m], E_{s,h}(m, (k+1) \bmod \omega_{max}[m])))$;
      $\forall s \in S \,.\, \forall h \in H$: $emit(s, h, \texttt{return})$;
25:   $k := k + 1$;
    **end while**
```

messages with port p values during program execution in at least one mode, i.e., the set of hosts on which a task input, actuator, or mode switch driver d is executed in at least one mode such that p is a source port of d. The host $\bar{h}(p)$ to which the port p is allocated is not in $recHosts(p)$. For a given task t, let the set $sendOutPorts(t)$ be the set of task t output ports p for which there are hosts that must receive the message with the port p value (i.e., those with $recHosts(p) \neq \emptyset$).

According to Giotto semantics, each task t input (resp. copy) driver reads (resp. writes) input (resp. output) ports at the release (resp. termination) time instants defined by the beginning (resp. end) of the task t period. In the distributed SCC implementation each copy driver is still executed by an E code instruction at the end of the task period. However, each task input driver is executed by an S code instruction and it is delayed if its source ports need to be sent over the network first. In general, in each task period, the transmission of sensor ports precedes task execution, which precedes the transmission of task output ports. More precisely, let d be the task input driver for a task t assigned to host h. For all sensor ports $p \in Src[d]$ such that $\bar{h}(p) \neq h$, a message $\mu[p]$ is received at h. The completion of the message $\mu[p]$ transmission updates on each host $h' \in recHosts(p)$ (including h) the sensor port $p_{h'}$. The task t input driver reads p_h (and other ports), applies its function, and writes to the task t input ports. It succeeds all sensor port messages and precedes the task t execution. The completion of the task t writes to the local copy of the task t output ports. The dispatch of the task output port message $\mu[p']$ for $p' \in Out[t]$ succeeds the task t completion. The completion of the task output port message $\mu[p']$ writes on each of the hosts in $h'' \in recHosts(p')$ to the task output port $p'_{h''}$. Finally, at each $h'' \in recHosts(p') \cup \{h\}$, the $copy[p'_{h''}]$ driver copies local into global task output ports at the end of the task t period (i.e., at the termination time of the task).

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 for simplicity we also assume that this value is the same for all ports. If a task reads a sensor port that needs to be received, then the task input driver is called exactly ϵ time instants after the task is released. Otherwise, it is executed at the time the task is released. Symmetrically, the transmission of task output ports is performed in a time interval of length ϵ before the task is terminated (i.e., before its period expires). We require that the time ϵ be less than or equal to the mode unit time $\gamma[m] = \pi[m]/\omega_{max}[m]$ for each mode m. This implies that the task input driver is always called before its source ports are updated with values that are more recent than what is allowed by the LET semantics.

```
$E_{s_1,h_1}(m_1, 0)$:
 call($copy[MixSound_{h_1}]$)
 call($copy[Spectrum]$)
 call($drv[ActDrv]$)
 call($dev[MixPlayer]$)
 call($dev[AudioSampler]$)
 release($\mu[AudioSampler]; 1$)
 release($0; Analyzer; 0$)
 future($4, E_{s_1,h_1}(m_1, 1)$)

$E_{s_1,h_1}(m_1, 1)$:
 call($copy[MixSound_{h_1}]$)
 call($drv[ActDrv]$)
 call($dev[MixPlayer]$)
 call($dev[AudioSampler]$)
 release($\mu[AudioSampler]; 1$)
 future($4, E_{s_1,h_1}(m_1, 0)$)
```

```
$E_{s_2,h_2}(m_1, 0)$:
 call($copy[MixSound]$)
 call($copy[StringSound]$)
 release($1; Mixer; 1$)
 release($1; \mu[MixSound]$)
 future($4, E_{s_2,h_2}(m_1, 1)$)

$E_{s_2,h_2}(m_1, 1)$:
 call($copy[MixSound]$)
 release($1; Mixer; 1$)
 release($1; \mu[MixSound]$)
 future($4, E_{s_2,h_2}(m_1, 0)$)
```

```
$E_{s_3,h_2}(m_1, 0)$:
 call($copy[MixSound]$)
 call($copy[StringSound]$)
 release($0; Generator; 0$)
 future($4, E_{s_3,h_2}(m_1, 1)$)

$E_{s_3,h_2}(m_1, 1)$:
 call($copy[MixSound]$)
 future($4, E_{s_3,h_2}(m_1, 0)$)
```

Figure 6. E code modules for the program G_A compiled by Alg. 1

Given a Giotto program, Alg. 1 generates all E code modules $\mathcal{E}_{s,h}$ executing in mode m. This is done in parallel for each supplier $s \in S$ and each host $h \in H$. The while loop generates a block of E code for each unit k of mode m. The E code compiler command $emit(s, h, instr)$ generates the E code instruction $instr$ for supplier s on host h. The compiler first generates `call` instructions to the task output (copy) drivers, actuator drivers, and actuator device drivers. Line 10 refers to [10] for details on generating a block of E code instructions that addresses mode switching; this is orthogonal to the issues discussed in this paper. The last segment handles `call` instructions for sensor device drivers, the invocation of tasks and messages, and the future invocation of the embedded machine at the next unit. The `release` instructions in the algorithm (lines 14, 19 and 21) are of a special form not needed for single-processor SCC. They indirectly contain precedence constraints that are necessary for correct communication by explicitly specifying the latency time ϵ. This number does not affect the program execution itself, but a supplier needs it in order to construct a correct schedule, i.e., S code module. We treat messages sent over the network analogous to tasks. In particular, we use the same SCC instructions for messages. The instruction $\texttt{release}(\mu[p]; \epsilon)$ releases the message $\mu[p]$ with the sensor port p value, but demands that the message transmission be completed by time ϵ from the release. The instruction $\texttt{release}(\epsilon_1; t; \epsilon_2)$ releases the task t with the constraint that the task be dispatched no earlier than time ϵ_1 after the release, and completed at the latest ϵ_2 time before the task t termination time. The instruction $\texttt{release}(\epsilon; \mu[p])$ 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 termination. The final `future` instruction causes the embedded machine to wait for time $\gamma[m]$ and then execute the E code for the next unit.

Fig. 6 shows the E code modules compiled by Alg. 1 from the audio mixer Giotto program G_A. The code for different suppliers on the same host is separated by a single horizontal line, and the code for different hosts is separated by two lines. The latency is chosen to be $\epsilon = 1ms$. For instance, the command $\texttt{release}(\mu[AudioSampler]; 1)$ releases the message with the sensor port $AudioSampler$ value, but also specifies a constraint that the message must be sent before $1ms$ expires.

Note that the code generation scheme of Alg. 1 implies the order of execution: copy drivers are followed by actuator drivers, mode switch drivers, and task input drivers, in that order. However, E code blocks compiled for the same host and same unit of a mode are fully composable, i.e., they can be executed in any order. If a task output port p is a source port of an actuator, mode switch, or task input driver that executes at a host h in a mode m, then $h \in \mathit{recHosts}(p) \cup \{\bar{h}(p)\}$. The set of hosts that receive port p data does not depend on the 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 a source port to any driver on h. This is so because in a mode where p is used, p must have a correct value even in the first period of execution in the mode.

4 Timing Interfaces

As presented in Section 3, each supplier obtains for each host an E code module specifying the release times of the tasks (resp. messages) that it implements, and for which it has to determine the times of execution (resp. transmission). Since both computation and communication resources are shared, this information must be accompanied by a temporal specification that provides exclusive time windows for task execution (resp. message transmission). This specification, which we call timing interface, is also given to each supplier. A timing interface defines the available computation and communication time windows, but not when to perform a particular action within these windows. This gives flexibility to a supplier, especially if multiple tasks are assigned to a supplier on a host. It also enables timing modifications that are local to a supplier and host, if a modification in the corresponding E module (e.g., adding a task) is made. In the next sections we show that the timing interface contains all information necessary for correct distributed code generation.

Formally, a supplier $s \in S$ on host $h \in H$ receives for each mode $m \in \mathit{Modes}$ of the Giotto program G a *timing interface*, which is a pair of predicates $T^m_{s,h} = (D^m_{s,h}, X^m_{s,h})$. The predicates $D^m_{s,h}, X^m_{s,h} : \{0, ..., \pi[m] - 1\} \to \{0, 1\}$ are defined as follows:

- $D^m_{s,h}(\ell) = 1$ iff in mode m at time ℓ supplier s on host h may execute a task from $\mathit{Tasks}_{s,h}$;
- $X^m_{s,h}(\ell) = 1$ iff in mode m at time ℓ supplier s on host h may send a message from $\mathit{Msgs}_{s,h}$.

Let $T_{s,h} = \{T^m_{s,h} | m \in \mathit{Modes}\}$ and $T = \{T_{s,h} | s \in S, h \in H\}$.

Fig. 7 shows a graphical representation of a timing interface for the program G_A from Fig. 1. The computation slots are shaded light; for these time units the corresponding predicate D is equal to 1. Recall the E module $\mathcal{E}_{s_1,h_1}$ of Fig. 6, in particular the blocks labeled $E_{s_1,h_1}(m_1, 0)$ and $E_{s_1,h_1}(m_1, 1)$. The timing interface given to supplier s_1 on host h_1 can be interpreted as follows. The task $Analyzer$ may be executed at any time in the intervals (1,3) and (5,7) ms (modulo $8ms$, which is the period of the mode m_1). Furthermore, the $0ms$-sample of the $AudioSampler$ sensor value may be sent at any time in the interval (0,1) ms, and the $4ms$-sample of the same sensor may be sent in (4,5) ms.

We assume that all hosts are clock-synchronized, so that communication is performed according to the Time Division Multiple Access (TDMA) protocol: in each time slot only one node is allowed to send data while all other nodes can listen for data. We have defined timing interfaces considering a simple communication architecture, where each host has only one processor for both computation and communication tasks. A host with an additional dedicated communication processor, e.g., a node in the Time-Triggered Architecture [1], can be modeled as two hosts.

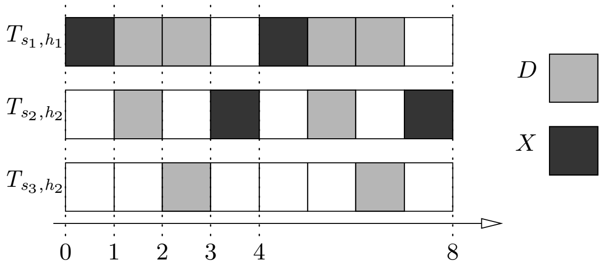

Figure 7. Timing interface for the program G_A

We next define *interface feasibility*, a property needed for the composition of SCC modules. First, we require that the timing interface windows for the same resource but different suppliers must be disjoint, i.e., at every time instant on each host at most one supplier may execute a task, and at most one of the suppliers may send a message. Second, when a host is supposed to receive data, no task execution is allowed. In particular, for sensor port data this is true in the latency time window (ϵ-window) after the data is read, and for task output port data, in the ϵ-window before the task termination time. Both properties are satisfied for the interface shown in Fig. 7.

Formally, a timing interface $T = (D, X)$ is *feasible* for a Giotto program G if the following two conditions are satisfied:

- (*Resource Sharing*) For all modes $m \in \mathit{Modes}$, suppliers $s_1, s_2 \in S$ (with $s_1 \neq s_2$), hosts $h_1, h_2 \in H$ (with $h_1 \neq h_2$), and times $\ell \in \{0, ..., \pi[m] - 1\}$,
    - at most one of $D^m_{s_1,h_1}(\ell)$, $D^m_{s_2,h_1}(\ell)$, $X^m_{s_1,h_1}(\ell)$, and $X^m_{s_2,h_1}(\ell)$ is equal to 1, and
    - at most one of $X^m_{s_1,h_1}(\ell)$, $X^m_{s_2,h_1}(\ell)$, $X^m_{s_1,h_2}(\ell)$, and $X^m_{s_2,h_2}(\ell)$ is equal to 1.
- (*Data Reception*) For all modes $m \in \mathit{Modes}$, units $k \in \{0, ..., \omega_{max}[m] - 1\}$, ports $p \in \mathit{SensePorts} \cup \mathit{OutPorts}$, and times $\ell \in \mathbb{N}_0$, if either
    - $p \in \mathit{senPorts}(m, k)$ and $k \cdot \gamma[m] \leq \ell < k \cdot \gamma[m] + \epsilon$, or
    - $p \in \mathit{taskOutPorts}(m, k+1)$ and $(k+1) \cdot \gamma[m] - \epsilon \leq \ell < (k+1) \cdot \gamma[m]$,

  and if $X^m_{\bar{s}(p),\bar{h}(p)}(\ell) = 1$, then $D^m_{s,h}(\ell) = 0$ for each supplier $s \in S$ and host $h \in \mathit{recHosts}(p)$.

Given a Giotto program and a set of timing interfaces, one for each supplier, host, and mode, the feasibility conditions can be checked independently for each interface.

Earliest-deadline-first S code. Provided with the pattern of task and message releases in an E code module $\mathcal{E}_{s,h}$, and available time windows in a timing interface $T_{s,h}$, the supplier s generates the schedule for host h, i.e., the order and timing of tasks and messages on h, and encodes it as an S code module $\mathcal{S}_{s,h}$. We briefly explain a potential generation scheme for $\mathcal{S}_{s,h}$. Even with the timing constraints imposed by $T_{s,h}$, it can be shown that the Earliest Deadline First (EDF) strategy is an optimal strategy with respect to schedule feasibility, i.e., if tasks and messages are schedulable in $T_{s,h}$ time windows by some strategy, then they are also schedulable by the EDF strategy. The release and deadline times of tasks and messages to be implemented by a supplier s on a host h in mode m are implicitly contained in the E code module $\mathcal{E}_{s,h}$. So, the supplier s can always check the EDF strategy and, if feasible, generate the S code module $\mathcal{S}_{s,h}$ according to the following scheme.

Let, for instance, an interval $[\ell_1, \ell_2) \subseteq [0, \pi[m])$, with integer bounds $\ell_1, \ell_2 \in \mathbb{N}_0$, be a computation window of the timing interface $T^m_{s,h}$, i.e., for all $\ell \in [\ell_1, \ell_2)$ be $D^m_{s,h}(\ell) = 1$. Let

$S_{s_1,h_1}(m_1,0)$: $\mathtt{call}(InDrv_1)$ $\mathtt{dispatch}(\mu[MixPlayer],1)$ $\mathtt{idle}(1)$ $\mathtt{dispatch}(Analyzer,3)$	$S_{s_1,h_1}(m_1,1)$: $\mathtt{dispatch}(\mu[MixPlayer],1)$ $\mathtt{idle}(1)$ $\mathtt{dispatch}(Analyzer,3)$
$S_{s_2,h_2}(m_1,0)$: $\mathtt{idle}(1)$ $\mathtt{call}(InDrv_2)$ $\mathtt{dispatch}(Mixer,2)$ $\mathtt{idle}(3)$ $\mathtt{dispatch}(\mu[MixSound],4)$	$S_{s_2,h_2}(m_1,1)$: $\mathtt{idle}(1)$ $\mathtt{call}(InDrv_2)$ $\mathtt{dispatch}(Mixer,2)$ $\mathtt{idle}(3)$ $\mathtt{dispatch}(\mu[MixSound],4)$
$S_{s_3,h_2}(m_1,0)$: $\mathtt{call}(InDrv_3)$ $\mathtt{idle}(2)$ $\mathtt{dispatch}(Generator,3)$	$S_{s_3,h_2}(m_1,1)$: $\mathtt{idle}(2)$ $\mathtt{dispatch}(Generator,3)$

Figure 8. S code modules for the program G_A

$t_1, t_2, ..., t_{|Tasks_{s,h}|}$ be the EDF permutation of tasks $Tasks_{s,h}$ at unit k of mode m (the task t_1 has the earliest deadline). The EDF S code module $\mathcal{S}_{s,h}$ has the following sequence of instructions:

$$\mathtt{idle}(\ell_1 - k \cdot \gamma[m])$$
$$\mathtt{dispatch}(t_1, \ell_2 - k \cdot \gamma[m])$$
$$\mathtt{dispatch}(t_2, \ell_2 - k \cdot \gamma[m])$$
$$...$$
$$\mathtt{dispatch}(t_{|Tasks_{s,h}|}, \ell_2 - k \cdot \gamma[m])$$

The entire EDF S code module consists of such code segments for each computation or communication slot of the timing interface. Fig. 8 shows all EDF S code modules for the Giotto program G_A which are generated using the timing interface of Fig. 7. Note that these modules also contain invocations of task input drivers.

5 Implementation

Our test system consists of several off-the-shelf PC hosts with 200MHz Pentium Pro processors and 128MB RAM. All hosts are equipped with standard 100Mbit Ethernet network cards and are locally connected. The underlying operating system is RTLinux, where standard Linux runs under the control of a real-time kernel as the lowest priority task [18]. In contrast to Linux' fair time-sharing scheduling, RTLinux uses a simple priority-based preemptive scheduler, thus permitting real-time functions to operate in a predictable and low-latency environment. In our tests the maximum scheduling latency was about 30μs.

Real-time communication is attained through a special network driver [19] that precludes the standard Ethernet CSMA/CD protocol by establishing a TDMA-based time-triggered protocol, where each node has exclusive access to the network within its scheduled time slot. A software-based synchronization of the hosts is carried out by controlling the period of a thread that performs send and receive network operations. The control algorithm uses the arrival times of incoming data packets. The communication cycle is shown in Fig. 9. For the purposes of synchronization, one of the hosts is designated as master and all others as clients. In each cycle the master sends a sync packet with the id of the client that is supposed to respond by sending a resync packet in the next slot. The subsequent slots are reserved for each of the hosts to send actual data packets. If T_0 is the duration of a single slot, and N is the number of hosts operating under the time-triggered protocol, then the cycle repeats after time $T_0 \cdot (N + 2)$.

In general, the protocol latency, i.e., the time between the send call of the network driver and the arrival of the data packet, depends on the time instant at which the call is made. However, the driver provides a function that synchronizes the sending thread with the network schedule, i.e., the driver resumes the thread when it reaches the exclusive time slot to send a message. This mechanism enables the precise timing in the interpretation of the SCC instructions (including message dispatch) with respect to the global time. The distributed SCC virtual machine is built as a dynamically loadable RTLinux kernel module. For the code of each supplier the machine maintains a context data structure similar to the non-distributed implementation described in [6]. To implement distributed SCC correctly we make use of special RTLinux calls that suspend and resume task threads.

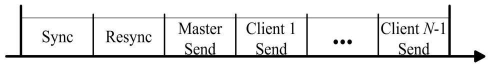

Figure 9. Cycle of the communication protocol [19]

To test the virtual machine we implemented the audio application G_A through the distributed SCC program shown in Fig. 6 and 8. Note that in Fig. 8 each `dispatch` instruction with a task (resp. message) as an argument executes in computation (resp. communication) slots shown in Fig. 7. In this setup each time slot lasts $T_0 = 1ms$, and an entire communication cycle lasts $4ms$ (N=2). The maximum bandwidth available to each host in such a configuration is 2.86$Mbit/s$. The tests show that the sound card is fed continuously with samples. The audio reproduced back at h_1 plays without any noticeable interruption or other sound defects. The estimated overhead of the network driver synchronization thread is 25μs. The overhead of the virtual machine, i.e., the time it takes to go through the machine event loop with two trigger and thread instances, is less than 12μs (divided roughly equally between E and S parts). Since the machine is invoked at 1kHz, the system overhead is about 3.7%. The actuator jitter is less than 2μs, since in Giotto a task output is written at the task termination time. In these measurements we used the Pentium time stamp counter, the most precise PC clock.

6 Compositional SCC Analysis

We first characterize the control-flow graphs of the distributed SCC program that is compiled from a Giotto program G according to the scheme presented in Section 3. The distributed SCC program is then represented as a set of state-transition systems, one for each supplier and host, which are used to verify the correctness of this implementation of G.

6.1 Giotto-Generated Distributed SCC

We start by describing E and S code modules separately, and then define the entire distributed SCC program. Let G be a Giotto program with M modes. Let $g_{s,h}$ be equal to $|Tasks_{s,h}|+|Msgs_{s,h}|+|Drvs_{s,h}|$, i.e., $g_{s,h}$ represents the size of the program part which is allocated to supplier s on host h. Let a node of a directed graph without predecessor (resp. successor) be called a source (resp. sink) node of the graph. A *G-generated E module* $\mathcal{E}_{s,h}$ consists of a directed acyclic control-flow graph $(V^{\mathcal{E}}_{s,h}, E^{\mathcal{E}}_{s,h})$, two edge-labeling functions κ and λ, and a node-labeling function η. Each edge $e \in E^{\mathcal{E}}_{s,h}$ is labeled with an instruction $\kappa(e)$ and an argument $\lambda(e)$, and each node $v \in V^{\mathcal{E}}_{s,h}$ is labeled with a pair $\eta(v) = (m, k)$ such that m is a mode and k is a unit of m, i.e., $k \in \{0, ..., \omega_{max}[m]\}$. The graph $(V^{\mathcal{E}}_{s,h}, E^{\mathcal{E}}_{s,h})$ has the following properties:

- Each path from a source to a sink consists of
  - a sequence of $O(g_{s,h})$ edges e, each with a $\kappa(e) = \mathtt{call}$ instruction that calls a driver $\lambda(e)$ from $Drvs_{s,h}$, followed by
  - a sequence of $O(g_{s,h})$ edges e, each with a $\kappa(e) = \mathtt{release}$ instruction that releases a task or message $\lambda(e)$ from $Tasks_{s,h} \cup Msgs_{s,h}$, and followed by
  - a single edge e with a $\kappa(e) = \mathtt{future}$ instruction and an argument $\lambda(e) = (\delta, v')$ that marks a source v' of $V^{\mathcal{E}}_{s,h}$ for execution after $\delta \in \mathbb{N}_{>0}$ units of time.

- For each mode $m \in Modes$ and each unit $k \in \{0, ..., \omega_{max}[m]\}$ there exists
  - exactly one source node v such that $\eta(v) = (m, k)$, and
  - at most one node v such that $\eta(v) = (m, k)$ and v has more than one successor; such a node v has less than M successors.

Let all numbers in G, i.e., mode periods as well as task and actuator frequencies and $\omega_{max}[m]$, be bounded by n. For instance, for the Giotto program G_A, the largest constant n is equal to 8. The number of sources of $(V^{\mathcal{E}}_{s,h}, E^{\mathcal{E}}_{s,h})$ is $O(M \cdot n)$, and the number of sinks is $O(M^2 \cdot n)$. Assuming, for simplicity, that the number M of modes is bounded, the size of $V^{\mathcal{E}}_{s,h}$ is $O(g_{s,h} \cdot n)$.

A *G-generated S module* $\mathcal{S}_{s,h}$ consists of a directed control-flow graph $(V^{\mathcal{S}}_{s,h}, E^{\mathcal{S}}_{s,h})$, two node-labeling functions ρ and ν, and an edge-labeling function λ. We require that the graph $(V^{\mathcal{S}}_{s,h}, E^{\mathcal{S}}_{s,h})$ consists of chains of total length $O(g_{s,h} \cdot n)$. Each control location $u \in V$ is labeled by one of the following:

- $\rho(u) = \texttt{dispatch}$, $\nu(u) \in Tasks_{s,h} \cup Msgs_{s,h}$, and node u has a successor u' such that $\lambda(u, u') \in \mathbb{N}_{>0}$. If $\nu(u) \in Tasks_{s,h}$, then 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 $\nu(u) \in Msgs_{s,h}$, then the analogous explanation holds for the transmission of the message $\nu(u)$.
- $\rho(u) = \texttt{idle}$ 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 of thread creation.
- $\rho(u) = \texttt{call}$ and u has a successor u' such that $\lambda(u, u') \in Drvs_{s,h}$. The execution of (u, u') calls driver $\lambda(u, u')$.
- $\rho(u) = \triangledown$ and u has no successor indicates thread termination.

A *G-generated SCC module* $P_{s,h}$ for a supplier s and a host h consists of a G-generated E module $\mathcal{E}_{s,h}$, a G-generated S module $\mathcal{S}_{s,h}$, and an *annotation function* $\Phi_{s,h}$ that maps each sink of the control graph of $\mathcal{E}_{s,h}$ to a node in the control graph of $\mathcal{S}_{s,h}$. When the E code execution arrives at a sink v, this creates a new thread of S code which starts at control location $\Phi_{s,h}(v)$. Let $V^{\mathcal{E}}_h$ be the union of node sets $V^{\mathcal{E}}_{s,h}$ over all suppliers $s \in S$, i.e., the set of all E code control locations on host h. Each function $\Phi_{s,h}$ maps a sink node $v' \in V^{\mathcal{E}}_{s,h}$ to a source node $\Phi_{s,h}(v') \in V^{\mathcal{S}}_{s,h}$ such that if $(v, v') \in E^{\mathcal{E}}_{s,h}$ and $\kappa(v, v') = \texttt{future}$ and $\lambda(v, v') = (\ell, \cdot)$, then the chain in $(V^{\mathcal{S}}_{s,h}, E^{\mathcal{S}}_{s,h})$ that starts from the node $\Phi_{s,h}(v')$ does not contain numbers, i.e., clock timeouts in `dispatch` and `idle` instructions, larger than ℓ. According to the last condition, if the next E code instruction is executed after ℓ time units, then the chain of S code instructions describes the schedule for at most the next ℓ time units. Note that if G is a single-mode program, then both $(V^{\mathcal{E}}_{s,h}, E^{\mathcal{E}}_{s,h})$ and $(V^{\mathcal{S}}_{s,h}, E^{\mathcal{S}}_{s,h})$ consist of chains of size $O(g_{s,h})$. Lastly, a *G-generated distributed SCC program P* over a set S of suppliers and a set H of hosts is a function that assigns to each $s \in S$ and each $h \in H$ a G-generated SCC module $P_{s,h}$ for a supplier s and a host h.

Transition-system semantics. A *state* of a G-generated distributed SCC program P consists of a port valuation function r that maps each port in $Ports_P$ to a value of the appropriate type, 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 $c : Tasks \cup Msgs \to \mathbb{N}_0 \cup \{\bot\}$, 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 future invocations, and a thread function θ that assigns to each host $h \in H$ a set θ_h of threads. 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. Let c be the function such that for each task $t \in Tasks$, the status $c(t) \in \mathbb{N}_0$ indicates that t has been released and executed for $c(t) \geq 0$ time units; the status $c(t) = \bot$ indicates that t has been completed (or not yet released). For a message $\mu \in Msgs$, $c(\mu)$ is defined analogously for the message release and transmission.

The appendix presents the semantics of a distributed SCC program P by defining a transition system on the space of states of P. Each transition represents either the execution of an E or S code instruction on one of the hosts, or a time step. A series of *E transitions* corresponding to a block of E code instructions are taken when a trigger becomes true. A *completion S transition* is taken when a task or message completes; a *timeout S transition*, when a timeout on a `dispatch` or `idle` instruction becomes true; and a *transient S transition*, when an S code `call` instruction is executed.

For a given initial state q_0, a *trace* of the distributed SCC program P is an infinite sequence $q_0, q_1, \ldots$ of states of P such that for all $i \in \mathbb{N}_0$, there exists a transition from q_i to q_{i+1}. Let $w_{s,h} : Tasks_{s,h} \cup Msgs_{s,h} \to \mathbb{N}_{>0}$ be the worst-case execution or transmission time (wcet) function for the tasks and messages of supplier $s \in S$ on host $h \in H$, and let w be the set of such functions for all suppliers and all hosts. A trace of P is an *w-trace* if for each supplier $s \in S$, host $h \in H$, and each invocation of a task or message $x \in Tasks_{s,h} \cup Msgs_{s,h}$, the invocation x completes execution (transmission) within time $w_{s,h}(x)$.

6.2 Interface Compliance and Time Safety

For the compositional analysis of a distributed SCC program we need the following two properties. Let G be a (multi-mode) Giotto program, let $T_{s,h}$ be a timing interface for a supplier s and a host h, let $P_{s,h}$ be the G-generated SCC module, and let $w_{s,h}$ be a wcet function. The module $P_{s,h}$ *interface-complies* with $T_{s,h}$ if all `dispatch` instructions of $P_{s,h}$ execute in time intervals provided by $T_{s,h}$. In our example each SCC module $P_{s,h}$ defined by the E and S code blocks in Fig. 6 and 8 interface-complies with the timing interface $T_{s,h}$ shown in Fig. 7, because the S code in Fig. 8 was generated as EDF S code with respect to this interface.

The module $P_{s,h}$ is *time-safe* if (1) no driver reads from output ports of a task (resp. message) assigned to supplier s on host h before it completes execution (resp. transmission), and (2) no driver writes to input ports of a task (resp. message) after it starts execution (resp. transmission). This requirement ensures that all task release and termination times of the original Giotto program are maintained [10]. Let, for instance, the worst-case execution (resp. transmission) times of all tasks (resp. messages) be $1ms$. Each SCC module $P_{s,h}$ defined by the E and S code blocks in Fig. 6 and 8 is time-safe. For example, in P_{s_2,h_2}, the input ports of the task *Mixer* are written at time $1ms$ ($InDrv_2$ driver), its output ports are read at $4ms$ (*copy*[*MixSound*] driver), and the task starts execution at $1ms$, but completes before $2ms$.

We now give the formal definitions of interface compliance and time safety as safety properties, so that it becomes clear how to check them. A state of a distributed SCC program P with a program counter function v and thread function θ violates *interface compliance with* $T_{s,h} = (D_{s,h}, X_{s,h})$ if there exists a thread $(u, \delta) \in \theta_h$ such that $\rho(u) = \texttt{dispatch}$, $\eta(v_h) = (m, k)$, and either (1) $\nu(u) \in Tasks_{s,h}$ and $D^m_{s,h}(k \cdot \gamma[m] + \delta) = 0$, or (2) $\nu(u) = Msgs_{s,h}$ and $X^m_{s,h}(k \cdot \gamma[m] + \delta) = 0$. We say that $(P_{s,h}, w_{s,h})$ *interface-complies* with $T_{s,h}$ if for all $w_{s,h}$-traces ψ of $\{P_{s,h}\}$, no state of ψ violates interface compliance with $T_{s,h}$.

A state of a distributed SCC program P with a program counter function v, status function c, and thread function θ violates *time safety on* (s, h) if there exists a task or message $x \in Tasks_{s,h} \cup Msgs_{s,h}$ such that either (a) 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 (b) there exists a

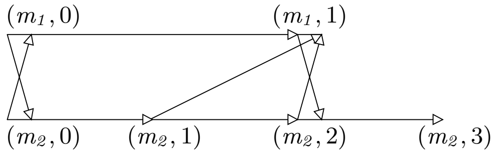

Figure 10. Graph related to $\mathcal{P}_{s,h}$ for G_A with additional mode m_2

thread $(u, \cdot) \in \theta_h$ with $\rho(u) = \texttt{call}$, u has a successor u', and $\lambda(u, u') = d$ (S code driver), and one of the following: (1) $Src[d] \cap Out[x] \neq \emptyset$ and $c(x) \neq \perp$, or (2) $Dst[d] \cap In[x] \neq \emptyset$ and $c(x) \neq 0$. We say that $(P_{s,h}, w_{s,h})$ is *time-safe* if for all $w_{s,h}$-traces ψ of $\{P_{s,h}\}$, no state of ψ violates time safety on (s, h).

Checking interface compliance and time safety. The paper [4] discusses time safety checking for single-mode, single-CPU Giotto programs. These results are here generalized to both the distributed and multi-mode settings. For distributed single-mode programs G we give pseudo-polynomial algorithms for checking the interface compliance and time safety of each G-generated SCC module. For distributed multi-mode programs the checks are sufficient. For details and proofs the reader is referred to [20]. Let a G-generated SCC module be given as a G-generated E module $\mathcal{E}_{s,h}$, a G-generated S module $\mathcal{S}_{s,h}$, and an annotation function $\Phi_{s,h}$. We first construct a directed graph $\mathcal{P}_{s,h}$ by connecting the control graphs of $\mathcal{E}_{s,h}$ and $\mathcal{S}_{s,h}$ through edges from each sink of $V^{\mathcal{E}}_{s,h}$ (resp. $V^{\mathcal{S}}_{s,h}$) to a source of $V^{\mathcal{S}}_{s,h}$ (resp. $V^{\mathcal{E}}_{s,h}$) determined by the map $\Phi_{s,h}$ and control flow of $\mathcal{E}_{s,h}$. It can be shown that each graph $\mathcal{P}_{s,h}$ is acyclic even if G is a multi-mode program [20]. For instance, consider the Giotto program G_A with the original mode m_1 and the additional mode m_2 given in Fig. 3, in which the *Mixer* task is invoked every $2ms$. Fig. 10 shows a graph in which each edge abstracts a chain of $O(g_{s,h})$ edges of the graph $\mathcal{P}_{s,h}$.

We next construct a state-transition graph by annotating each node of the graph $\mathcal{P}_{s,h}$ with a particular state of the SCC module $P_{s,h}$. The graph $\mathcal{P}_{s,h}$ is acyclic, so the nodes can be sorted and processed in topological order. Each source node of $\mathcal{P}_{s,h}$ (for each mode there is exactly one such node) is annotated with the state in which the trigger queue and thread set are empty and the status function maps each $x \in Tasks_{s,h} \cup Msgs_{s,h}$ to $\perp$ (recall that $c(x) = \perp$ means that x has not yet been released). For the other nodes of $\mathcal{P}_{s,h}$ we proceed by transforming the state of their immediate predecessors. We do so by performing one or more transition steps defined by the semantics of SCC programs (App. A). Task execution-time nondeterminism in time transition steps is eliminated by assuming that each task (or message) x completes exactly after the time given by the wcet $w_{s,h}(x)$. If a node v has more than one predecessor v', then the status function value at node v, for each $x \in Tasks_{s,h} \cup Msgs_{s,h}$, is the least value among the status function values for x at all predecessors v'. So, for the nodes with more than one incoming edge, we compute the task execution time pointwise and conservatively.

Checking the states of the graph $\mathcal{P}_{s,h}$ offers a sufficient condition for time safety and interface compliance of all executions of the distributed SCC module $P_{s,h}$. If no state of the graph $\mathcal{P}_{s,h}$ violates time safety and interface compliance, then the G-generated SCC module $(P_{s,h}, w_{s,h})$ interface-complies with $T_{s,h}$ and is time-safe. If this is not the case then, for a general multi-mode Giotto program G, we cannot conclude that SCC module $(P_{s,h}, w_{s,h})$ does not interface-comply with $T_{s,h}$ (or is not time-safe). This is because in the state construction of $\mathcal{P}_{s,h}$ different incoming edges of a node may impose conservative approximations on different tasks. Also, there may be unreachable modes [10]. However, if G is a single-mode program, then the state-transition graph $\mathcal{P}_{s,h}$ is a chain. So, if $\mathcal{P}_{s,h}$ does not interface-comply or is not time-safe at some state q, then the trace along the chain up to q is a counterexample. The size of $\mathcal{P}_{s,h}$ is $O(g_{s,h} \cdot n)$, because both $(V^{\mathcal{E}}_{s,h}, E^{\mathcal{E}}_{s,h})$ and $(V^{\mathcal{S}}_{s,h}, E^{\mathcal{S}}_{s,h})$ are of the same size. Constructing the transition graph $\mathcal{P}_{s,h}$, annotating it with states, and checking its states can be done in $O(g_{s,h} \cdot n)$ time. Therefore, we have the following theorem.

THEOREM 1. *Let G be a single-mode[2] Giotto program with all numbers bounded by n. Let $g_{s,h}$ and $T_{s,h}$ be the size of the part of G and the timing interface assigned to supplier s on host h. Let $P_{s,h}$ and $w_{s,h}$ be the G-generated SCC module and wcet function for supplier s on host h. It can be checked in time $O(g_{s,h} \cdot n)$ whether $(P_{s,h}, w_{s,h})$ interface-complies with $T_{s,h}$ and is time-safe.*

6.3 Distributed Code Generation Correctness

We show that LET semantics of a Giotto program is preserved by the distributed SCC program generated according to Alg. 1 if each SCC module satisfies interface compliance and time safety. If an SCC program preserves the LET semantics of a Giotto program we say that it implements the Giotto program.

Let G be a Giotto program, let $T = \{T_{s,h} \mid s \in S \text{ and } h \in H\}$ be a feasible interface for G, let $P = \{P_{s,h} \mid s \in S \text{ and } h \in H\}$ be a G-generated distributed SCC program, and let $w = \{w_{s,h} \mid s \in S \text{ and } h \in H\}$ be a wcet function for P. Let r^G_ℓ and r^P_ℓ be the port valuation functions at time $\ell \in \mathbb{N}_0$ for G and P [3]. A trace of P and a trace of G are *input-compatible* (resp. *output-compatible*) if they have the same sensor (resp. actuator) port values at the same times, i.e., $r^G_\ell(p) = r^P_\ell(p)$ for each $p \in SensePorts$ (resp. $p \in ActPorts$) and each time instant $\ell \in \mathbb{N}_0$. The pair (P,w) *implements* the Giotto program G if for every w-trace of P and every trace of G, input-compatibility implies output-compatibility (i.e., if, for all sensor inputs, they produce the same actuator outputs at the same times). The pair (P,w) interface-complies to T if for each supplier $s \in S$ and host $h \in H$, the G-generated SCC module $(P_{s,h},w_{s,h})$ interface-complies with $T_{s,h}$. We say that (P,w) is time-safe if $(P_{s,h},w_{s,h})$ is time-safe for each $s \in S$ and $h \in H$.

THEOREM 2. *Let G be a Giotto program, let T be a feasible timing interface for G, let P be the distributed SCC program G-generated according to Alg. 1, and let w be a wcet function. If (P,w) interface-complies to T and is time-safe, then (P,w) implements G.*

For the proof of this theorem we refer to [20]. Instead we give informal explanation why interface feasibility, interface compliance, and time safety ensure correctness of the implementation. If interface feasibility is violated, e.g., the time windows on a host are not disjoint, even if each supplier produces interface-compliant and time-safe code, the host may be overloaded and miss deadlines defined by the LET semantics. A similar outcome is possible if the interface is feasible, and each supplier on each host generates an SCC module that is individually time-safe, but it ignores the interface. Lastly, if a module does not satisfy one of the time-safety conditions, e.g., a time slot in the interface is not sufficiently large, then a task or message invocation may result in incorrect output. The compositional nature of interface compliance and time safety of (P,w) ensures that if, for some supplier s and host h, one module $P_{s,h}$ is modified, then for P to implement G it suffices to check if $(P_{s,h},w_{s,h})$ interface-complies with $T_{s,h}$ and if it is time-safe. Combining Theorems 1 and 2, we have the following.

COROLLARY 1. *Let G be a single-mode[2] Giotto program of size g with all numbers bounded by n. It can be checked in time $O(g \cdot n)$ if (P, w) implements G. Moreover, if $(P_{s,h},w_{s,h})$ is modified for a single supplier s and host h, then it can be checked in time $O(g_{s,h} \cdot n)$ if (P, w) still implements G.*

[2] For multi-mode Giotto the pseudo-polynomial check is only sufficient but not necessary.

Note that $(P_{s,h}, w_{s,h})$ can be modified either by modifying $\mathcal{E}_{s,h}$ (i.e., modifying task invocation and/or environment interaction), $\mathcal{S}_{s,h}$ (schedule), or $w_{s,h}$ (wcet). Suppose that in the audio example the integrator wants to assign additional functionality to supplier s_3 on host h_2, say, mix with another synthesized sound with a pitch twice as high. Supplier s_3 implements a new task $Generator_2$ (of two times higher frequency) with input driver $InDrv_4$, and modifies the S module $\mathcal{S}_{s_3,h_2}$ as shown below. Then, for correctness of the entire program P, only the modified module P_{s_3,h_2} needs to be checked for interface compliance and time safety.

```
$S_{s_3,h_2}(m_1, 0)$:
 call($InDrv_3$)
 call($InDrv_4$)
 idle(2)
 dispatch($Generator_2$, 3)
 dispatch($Generator$, 3)
```

```
$S_{s_3,h_2}(m_1, 1)$:
 call($InDrv_4$)
 idle(2)
 dispatch($Generator_2$, 3)
 dispatch($Generator$, 3)
```

7 Conclusion

We introduced timing interfaces and showed how they can be used to distribute the code generation for Giotto programs and distributed target platforms. The integration of the individually compiled components is performed by individually checking the interface compliance and time safety of each component. Our approach guarantees global timing requirements without solving a global scheduling problem: as part of the continuing effort of the Giotto project to trade performance for predictability and composability, the burden is shifted to the generation of timing interfaces. There are related efforts [12, 13, 21], how they can be optimized for different criteria is a topic for future research.

References

[1] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.

[2] http://www.flexray-group.com; http://www.autosar.org.

[3] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. In *Proc. IEEE* 91, pp. 84–99, 2003.

[4] T.A. Henzinger, C.M. Kirsch, and S. Matic. Schedule-carrying code. In *Proc. EMSOFT*, LNCS 2855, pp. 241–256, Springer, 2003.

[5] T.A. Henzinger and C.M. Kirsch. The Embedded Machine: predictable, portable real-time code. In *Proc. PLDI*, pp. 315–326, ACM, 2002.

[6] C.M. Kirsch, M.A.A. Sanvido, and T.A. Henzinger. A programmable microkernel for real-time systems. In *Proc. VEE*, ACM, 2005.

[7] C.M. Kirsch, M.A.A. Sanvido, T.A. Henzinger, and W. Pree. A Giotto-based helicopter control system. In *Proc. EMSOFT*, LNCS 2491, pp. 46–60, Springer, 2002.

[8] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[9] A. Benveniste, L.P. Carloni, P. Caspi, and A.L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In *Proc. EMSOFT*, LNCS 2855, pp.35–50, Springer, 2003.

[10] T.A. Henzinger, C.M. Kirsch, R. Majumdar, and S. Matic. Time-safety checking for embedded programs. In *Proc. EMSOFT*, LNCS 2491, pp. 76–90, Springer, 2002.

[11] P. Caspi, et al. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proc. LCTES*, pp. 153-162, ACM, 2003.

[12] A. Mok and X. Feng. Real-time virtual resource: a timely abstraction for embedded systems. In *Proc. EMSOFT*, LNCS 2491, pp. 182–196, Springer, 2002.

[13] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. RTSS*, pp. 2–13, IEEE, 2003.

[14] H. Kopetz and N. Suri. Compositional design of real-time systems: a conceptual basis for the specification of linking interfaces. In *Proc. ISORC*, pp. 51–60, 2003.

[15] J. Rushby. Partitioning in avionics architectures: requirements, mechanisms, and assurance. In *NASA Contractor Report 209347*, SRI International, 1999.

[16] B. Hardung, T. Koelzow, and A. Krueger. Reuse of software in distributed embedded automotive systems. In *Proc. EMSOFT*, pp. 203–210, ACM, 2004.

[17] K. Karplus and A. Strong. Digital synthesis of plucked-string and drum timbres. in *Computer Music Journal* 7, pp. 43–55, 1983.

[18] V.Yodaiken. RTLinux Manifesto. In *Proc. LinuxExpo*, 1999.

[19] S. Lankes, A. Jabs, and M. Reke. A time-triggered Ethernet protocol for real-time CORBA. In *Proc. ISORC*, pp. 215–222, 2002.

[20] T.A. Henzinger and S. Matic. *Distributed Schedule-Carrying Code*. Tech. Rep. UCB/CSD-04-1360, 2004.

[21] S. Shigero, M. Takashi, and H. Kei. On the schedulability conditions on partial time slots. In *Proc. RTCSA*, pp. 166–173, IEEE, 1999.

Appendix A. Formal Distributed SCC Semantics

In [4] we give an operational semantics of schedule-carrying code by defining a state-transition system in which all port values are abstracted away. Here we are interested in the input-output behavior of *distributed* SCC, so we extend the formalism by taking into account port values and the distributed nature of code. We present the interleaving semantics for SCC modules of all suppliers on all hosts. To use the same notation for messages as for tasks, let the message input ports $In[\mu[p]]$ formally be $\{p\}$, let the message output ports $Out[\mu[p]]$ be $\{p_h \mid h \in recHosts(p)\}$, and let the message function $task[\mu[p]]$ be the identity function from the message input to output ports. A state $q = (r, v, c, \tau, \theta)$ has a *transition* to a state $q' = (r', v', c', \tau', \theta')$ if one of the following is true:

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 $c(\nu(u)) = \bot$ and $\rho(u) = \texttt{dispatch}$. Let the successor of u be u'. Then $r' = r$ except that $r'(Out[\nu(u)]) = task[\nu(u)](r(In[\nu(u)]))$, $(v', c', \tau') = (v, c, \tau)$, 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 $\rho(u) = \texttt{call}$. Let the successor of u be u'. Then $r' = r$ except that $r'(Dst[\lambda(u, u')]) = drv[\lambda(u, u')](r(Src[\lambda(u, u')]))$, $(v', c', \tau') = (v, c, \tau)$, 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. In case (1) let $(0, \bar{v})$ be the first such pair in τ_h. Then $p = p'$, $v' = v$ except that $v'_h = \bar{v}$, $c' = c$, $\tau' = \tau$ except that $\tau'_h = \tau_h \setminus \{(0, \bar{v})\}$, and $\theta' = \theta$. In case (2) one of the following: (a) $\kappa(v_h, v'_h) = \texttt{call}$ and $r' = r$ except that $r'(Dst[\lambda(v_h, v'_h)]) = drv[\lambda(v_h, v'_h)](r(Src[\lambda(v_h, v'_h)]))$, $c' = c$, and $\tau' = \tau$; (b) $\kappa(v_h, v'_h) = \texttt{release}$ and $r' = r$, $c' = c$ except that $c'(\lambda(v_h, v'_h)) = 0$, $\tau' = \tau$; or (c) $\kappa(v_h, v'_h) = \texttt{future}$ and $r = r'$, $c' = c$, 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 $(r', v', c, \tau') = (r, v, c, \tau)$, and $\theta = \theta'$ 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 $p \in Ports_P \setminus \{p_c\}$, and $r'(p_c) = r(p_c) + 1$. For $\ell = r(p_c)$, we call the function $r_\ell = r$ the *port valuation at time* ℓ. For each $h \in H$, let $X_h = \{x \mid (u, \cdot) \in \theta_h, \rho(u) = \texttt{dispatch}, \nu(u) = x\}$, and let $\bar{x}_h \in X_h$ be the task or message to be executed on h. Then $v' = v$; 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 $(u, \delta + 1)$; if $x \in Tasks_{s,h} \cup Msgs_{s,h}$ for some $s \in S$, then $c'(x) = c(x) + 1$ or $c'(x) = \bot$ if $x = \bar{x}_h$, and $c'(x) = c(x)$ if $x \neq \bar{x}_h$. In case $c'(x) = \bot$ we say that on the transition (q, q'), the task or message x *completes* after execution time $c(x) + 1$.