Compositionality in Deterministic Real-Time Embedded Systems

Slobodan Matic

Electrical Engineering and Computer Sciences
University of California at Berkeley

Compositionality in Deterministic Real-Time Embedded Systems

by

Slobodan Matic

B.S. (University of Belgrade)

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Thomas A. Henzinger, Chair
Professor Edward A. Lee
Professor Raja Sengupta

Spring 2008

The dissertation of Slobodan Matic is approved.

Chair   Date

Date

Date

University of California, Berkeley

Spring 2008

Abstract

Compositionality in Deterministic Real-Time Embedded Systems

by

Slobodan Matic

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Thomas A. Henzinger, Chair

Many computing applications, especially those in safety critical embedded systems, require highly predictable timing properties. However, time is often not present in the prevailing computing and networking abstractions. In fact, most advances in computer architecture, software, and networking favor average-case performance over timing predictability. This thesis studies several methods for the design of concurrent and/or distributed embedded systems with precise timing guarantees. The focus is on flexible and compositional methods for programming and verification of the timing properties. The presented methods together with related formalisms cover two levels of design:

- Programming language/model level. We propose the distributed variant of Giotto, a coordination programming language with an explicit temporal semantics - the logical execution time (LET) semantics. The LET of a task is an interval of time that specifies the time instants at which task inputs and outputs become available (task release and termination instants). The LET of a task is always non-zero. This allows us to communicate values across the network without changing the timing information of the task, and without introducing nondeterminism. We show how this methodology supports distributed code

generation for distributed real-time systems. The method gives up some performance in favor of composability and predictability. We characterize the tradeoff by comparing the LET semantics with the semantics used in Simulink.

- Abstract task graph level. We study interface-based design and verification of applications represented with task graphs. We consider task sequence graphs with general event models, and cyclic graphs with periodic event models with jitter and phase. Here an interface of a component exposes time and resource constraints of the component. Together with interfaces we formally define interface composition operations and the refinement relation. For efficient and flexible composability checking two properties are important: incremental design and independent refinement. According to the incremental design property the composition of interfaces can be performed in any order, even if interfaces for some components are not known. The refinement relation is defined such that in a design we can always substitute a refined interface for an abstract one. We show that the framework supports independent refinement, i.e., the refinement relation is preserved under composition operations.

_____

Professor Thomas A. Henzinger
Dissertation Committee Chair

Contents

i

List of Figures

List of Tables

Acknowledgements

For every ailment under the sun,

There is a remedy, or there is none;

If there be one, try to find it;

If there be none, never mind it.

Mother Goose rhyme

My coming from troubled but revived Serbia, only few weeks before September 11th, was a departure from the safety of a familiar environment and the only way to become aware of one's own virtues and vices. One better drown learning to swim than spend the entire life sitting on the shore watching ships pass by. No doubt, the University of California at Berkeley and its EECS Department were offering much more than one can chew. Professors, students and researchers striving to live up to their potential while being occupied with intellectual feats and scholarly combats. And Ruth Gjerde, who listens to students attentively, nods with compassion, and navigates them through administrative mazes. Living in the International House Berkeley for three years reinforced my will to understand both other cultures and my own culture through the other's perspectives. Serbia is one of those places located on the cusps of a few cultural divides, and thus challenged after every major shift of power in the world. I am grateful to all the people, on all sides of all divides, who ever felt the need or even struggled to keep communication open while standing up for their own principles.

This thesis is the result of research under the guidance of Prof. Thomas Henzinger. There is no memory of working with him more vivid than the time spent in problem solving discussions. And these have all the potential to be fruitful. Beforehand, there is just enough pressure put upon a student to focus and come prepared. During a discussion, the exchange of arguments is conducted almost with a belief that the most important problem in the

world is being unraveled. Prof. Henzinger looks absorbed in the problem as if there were no other meetings waiting. And each time a step forward, no matter how small, is being taken. Afterwards, the student is encouraged even to go risky and remains full of hope, at least until the next meeting. Some argue that, in research, feeling confused is essential to progress. I always admired Prof. Henzinger's fortitude to feel lost at first, only to be able to regroup on the spot and reap the results a few moments later. And I can clearly see the marker in his hands, coming back and forth to the whiteboard, on and on, while he is thinking out loud and recollecting his thoughts before he starts to write. I am thankful to him for sharing his ideas, intuition, vision and work ethic.

When Prof. Henzinger decided to move to Switzerland, he generously offered his students either to come with him and work at the EPF Lausanne, or to stay at and get the degree from UC Berkeley, or to do something in between. I have never regretted choosing the third option and have only benefited from both environments and his continuing support. Incidentally, if it hadn't been for him, I would have never had a chance to come close to the ominous but so beguiling north face of Matterhorn. I cherish those paper deadline moments when Prof. Henzinger calls me in Berkeley at 1am Europe time and comments on everything from the definite article usage in English language to the cogent reformulation of propositions. And I also never forgot the positive appraisal he gave after my first EM-SOFT presentation, knowing what burden for me public speaking could be at that moment. He has this delicate sense to know when to show his laissez-faire attitude and when to push a student in a certain direction. A good advisor has to help students both accept the fact that they may not be the best and develop the belief that they can still do well and keep maturing if they work hard. I am not sure whether Prof. Henzinger agrees with the simple yet often elusive message that the verses of the rhyme given above convey, but working with him helped such a sentiment become closer to my mind.

All research for this thesis was done within the Center for Hybrid and Embedded Software Systems. Around Cory Hall, and the Donald O. Pederson Center in particular, I got

to know and interact with so many fascinating people, and I feel obliged to keep valuable memories of them all. Unfortunately, due to the inevitable limits of human interaction, for most of them I only scratched a bit of what they have to offer. Prof. Edward Lee used to advise me even before he kindly accepted to be my Berkeley co-advisor. Not only are his advanced material lectures smooth as silk, but he emanates a tremendous passion for knowledge and discovery. He seems so open to appreciate other people's work, integrate it with his own and already known research, and pass it to others. Towards research he almost comes as playful, clearly demonstrating how joy of research often means the joy of truly understanding ideas and concepts. I will remember Prof. Christoph Kirsch, a postdoc in my early days at Berkeley, for helping me define problems, putting up with my ideas, and initially serving as an interpreter between Prof. Henzinger and me.

With my office mates, Krishnendu Chatterjee and Arindam Chakrabarti, I shared much more than piles of papers, office commandments or courteous talk. That blissful Yosemite trip all with its $270°$ car rotation accident comes to mind. I find disheartening that, despite good will and intention, we did not get to work on a problem together, and do hope we somehow correct this in future. Krishnendu, so breezy yet engrossed, so "never do anything" yet weeks ahead of all deadlines, so self accomplished yet without a breath of arrogance. With Arindam I spent countless hours in social and economic discussions, often not knowing what stand Arindam would take in an argument and thus enjoying every second of it even more. I am thankful to him for disturbing my narrow perception of the role of the free market. Rational people may reasonably disagree on a matter, but despite that, it always seemed that both of us yearned to share those daily stories that had captivated us. I do realize, sadly, that one is offered only few lifetime opportunities to make such a meaningful connection with a friend.

I hope never to forget what, while staying in Berkeley, I learned to deem very precious: mid-afternoon fog rolling over the northern crest of Berkeley Hills, the Bridge and its symbolism, reflection of Alta Peak over Pear Lake in Sequoia National Park, small collection of

Giacometti's sculptures in San Francisco Museum of Modern Art, homeless old lady that I kept seeing in the middle of the night on the corner of Telegraph and Channing, Bay Area citizens' activism, KGO 810 news talk and Gene Burns' commentary, Aspan Dahmubed's April Fools' Day "love" letter, soft and distant sounds of Sunday 6pm Campanile concerts and swarms of students around Sather Gate at noon.

I revere my parents, Milena and Milivoje, for empowering me to accept immaterial wealth and values, and for embedding me with moral virtues of hard work and diligence. I salute them hoping they can hear the message that I am still on track of the dream they had for me (Slobodan - the one who is free, Serbian). My siblings, Radina and Ljubisa, are the foremost ones to follow example of and permanent sources of both benevolent critique and motivation. Living with all of them bestowed upon me a unique blend of soulful empathy and existential felicity. Throughout these years I constantly felt frustrated for not being able to explain them exactly the relevance of the research problems I was working on. Therefore, although I know they always stood by me, it somehow feels not right to say this thesis is devoted to them. I would rather now pledge my word that I shall devote more time to them in future. My work was, however, worth their sacrifice of being far away from me. That they should have no doubt about. No matter how (in)significant this work may be in the grand scheme of things, it means so much to me. I feel very fortunate.

Chapter 1

Introduction

1.1 Motivation

Embedded computing systems are nowadays common in most spheres of life. In fact, as popular definition suggests, everywhere where computation is subject to physical constraints. It is estimated that an average American comes into contact with about a hundred embedded computers per day, and that by the year 2010 software for embedded computers will account for 90% of all software being written. The design of such a system asks for a specified behavior to be mapped on a computing platform under certain constraints. In the application domains such as automotive, mechatronics, and multimedia, power consumption and size constraints are commonly not of the highest concern. Instead, *time related constraints* such as latency, deadline, throughput, response-time predictability are required by specification. For instance, in safety-critical systems the deterministic and timely response is often required for fault-tolerance. A common way to achieve it is through replica determinism that demands the redundant system nodes to take the same decision at about the same time.

Lee observes in [49] that time is not present in the prevailing computing and network-

ing abstractions used to design embedded systems. Unfortunately, this is true not only for hardware components, but also for software components such as operating systems, middleware concepts, and even design tools. In fact, the vast majority of dynamic or virtual features designed to increase the average-case performance have detrimental effects on *timing predictability*. Moreover, there are common examples in which a decrease of the average-case latency results with an increase of the worst-case latency. As Ziegenbein et al. note in [84] with respect to multiprocessor scheduling anomalies, in some cases the best-case performance needs to be considered simultaneously. In general, time predictability is defined as the difference between estimated upper and lower bound of an event time, or as the difference between estimated and measured worst-case times. In this thesis we consider systems in which timing predictability is far more important than average performance.

Theile and Wilhelm notice that most cases of low timing predictability fall in two categories, *unknown external interference* and *limited analyzability* [73]. In the former case, low predictability is caused by the limited knowledge or unavailability of the system information relevant for the implementation. For instance, inter-arrival event times or the underlying scheduling mechanism are unknown during the system design. In the latter case, system components and their parameters are substantially known, but such a knowledge is too complex to be analyzed effectively. For instance, many microprocessor architecture techniques like out-of-order execution or cache replacement strategies are too complicated for the time bounds to be estimated. Thus, there are at least two ways to attack the timing predictability problem in embedded systems: reduce the sensitivity to unknown system information (see, e.g., [31, 81]), and, use architectures or implementation techniques that can be analyzed more easily ([38, 20]). The elements of both strategies can be found in solutions for the problems we address in this thesis.

Both [49] and [73] analyze system architecture layers pertinent to time determinism. [73] lists four layers: hardware architecture (includes all design aspects below the instruction set), single task software development (code synthesis, intratask analysis and opti-

mization tools), task level (scheduling, shared resources, intertask synchronization), and distributed operation (distributed resources, end-to-end deadlines). In this study we focus on the methods addressing the two uppermost layers in this classification.

The step in the design process during which system performance, including power and timing, is validated against its specification is called *performance verification*. This step is often performed simultaneously with design-space exploration. According to a recent Embedded Market Forecasters [21] analysis, one quarter of embedded system designs missed the project schedule by at least 50%, about one third missed at least 50% of functional specifications, and more than 70% missed performance specifications by at least 30%. As discussed in the International Technology Roadmap for Semiconductors [66], the performance verification is already named as one of the top three issues in system-on-chip design.

The current industrial practice in performance verification is almost exclusively limited to (cycle-true) simulation and prototype testing. The examples include Seamless [55] in system-on-chip and AutoBox [18] in automotive domains. Even though simulation often seems viable for functional verification it is less so for the validation of performance, in particular, timing. Two major reasons are often quoted. The corner-case simulation vectors that result in worst-case performance are often not intuitive and, thus, very difficult to find, which makes the simulation procedure very time-consuming. Second, these methods require executable code, which can often be provided only in the later stages of design.

On the other hand, formal performance validation methods are yet to get significant attention from industry. The tools that do exist are typically only applied to individual component analysis, because system-level heterogeneity and complexity are still difficult to capture precisely in most formalisms. In principle, formal timing analysis should give both bounds for critical scenarios and the corner-case vectors that produce the scenario. Also, formal performance analysis seems suitable for design-space exploration since it runs considerably faster than simulation. Most of existing methods of formal timing analysis fall

3

in two categories, *code execution time* analysis and *resource sharing* analysis. The former methods are based on program path analysis and cycle-true processor modeling, and the latter on the research in scheduling for real-time operating systems. Notable industry-level examples are tools aiT [1] for C-code worst-case execution time analysis, and RapidRMA [74] for rate monotonic fixed priority scheduling.

Beside the fact that the applications are becoming more complex in the number, character and interaction of consisting components, what makes the timing determinism in embedded systems specific and less tractable? Up until recently, in traditional hardware design, system timing was guaranteed by hierarchical composition of individual hardware subsystems. This was possible since the component control was mostly single threaded. However, the reactive character of embedded software requires preemption and corresponding scheduling strategies. It is well-known (see, for instance, the work of Richter et al. [62]) that preemptive and time-driven scheduling introduces timing *dependencies* between components that are functionally independent. Recent system-on-chip designs use networks to connect multiple programmable processor cores with specialized hardware subsystems. *Optimization* of performance in such systems, e.g. optimization of power consumption, requires component specialization which increases system heterogeneity. *Heterogeneous* platforms, on the other hand, add to the problem complexity through combined effects of different scheduling strategies needed for different components. If distributed embedded systems are considered additional resource sharing is involved for communication, often with event buffering and bursts. All these factors render standard approaches to resource sharing ineffective.

Embedded systems are complex as a whole, but very often consist of smaller modules minimally interacting with each other. Such a structure makes embedded systems amenable to *component-based design*. This approach provides a means for decomposing a system into components, enabling the reduction of a complex design problem into multiple simpler design problems. Thus, in contrast to holistic methodologies, in this approach there is no

need to perform global analysis for every system configuration. It is often argued that, as the system complexity increases, component "cut-and-paste" methodologies are the only way to reach design productivity, because designers must work at higher levels of abstraction, reusing already designed and verified components.

Heterogeneous component-based frameworks (e.g. Ptolemy [48], Metropolis [25], UML [17]) are currently primarily used for functional verification and for performance analysis they need to be extended with analytical models. This is important because in component-based designs it is mostly the case that only local performance problems have the potential to impact other parts of the system. As noted in [62], functional problems are typically confined because of the practice to modularize functions such that most interdependencies are localized. However, there is no simple performance modularization, since performance interdependencies are introduced even between functionally independent processes.

This thesis studies several methods for the component-based design of systems with precise timing requirements. The focus is on formal flexible and scalable methods for programming and verification of timing properties. We address issues such as models of computation, design principles, qualitative properties of composition, and performance/predictability/efficiency tradeoffs. In the following sections we describe the problems with more details, briefly explain our solutions, and review the related work.

1.2 Composable Code Generation for Distributed Systems

According to Frischkorn in [22], by 2015 up to 40% of the costs of an automotive vehicle will be driven by electronics and software. About 60% of all development costs for a car electronic control unit will be related to software. In addition, while the number of pro-

cessors is expected to level in the range 60-70, the growth rate of software functions will be 300%. This is closely related to one of the goals of Autosar [3], a huge project in automotive industry: to decouple growth rate of number of functions from growth rate of number of electronic components. Similar trend exists in avionics software, where previously each control subsystem had its own dedicated resource, whereas new solutions increasingly offer a common computing platform for multiple functions.

Clearly, software integration issues will play the major role in design of such distributed control systems. Note that in standard design techniques, e.g. in those that use simulation for performance validation, the design errors often show up only during system integration. In addition, different software parts are often developed by different suppliers. Thus, the integrator should also have a freedom in choosing between in-house and externally developed components. In the Autosar project, the description of software components, their interfaces, resource needs and network topologies are based on Unified Modeling Language, a generalized specification language for object modeling. In this thesis we address similar issues by using a coordination language Giotto [31] extended with suitably defined component timing interfaces.

There exists a vast literature on synchronous-reactive design and a part of it targets distributed deterministic systems (see, for instance, the works of Benveniste and his collaborators [5, 6, 4]). This methodology is primarily successful at the specification level. In globally synchronous specifications, parallel components are assumed to execute at exactly the same points in time enforced by perfect clocks. Also, execution and communication are assumed to be timeless. Synchronous reactive programs written in Lustre have been compiled globally for distributed real-time systems [10], but this approach of Caspi et al. resolves underlying scheduling problems through integer linear programming, a method that is not compositional. Two often quoted problems that prevent synchronous specifications to scale well to distributed implementations are large variance in component computation or communication times and the difficulty of maintaining a global notion of time. To address

6

the former problem the clock typically has to run as slow as the slowest system component. In addition, in a distributed implementation, the synchronous communication lines between system components are often replaced with asynchronous ones and designers have to insure that such a composition does not change the semantics. The desynchronization procedure introduced by Benveniste et al. in [5] is a formal technique to replace the synchronous communication lines with unbounded buffers. The procedure has been applied from loosely time-triggered architectures [6] to traffic signal control systems [80].

On the other hand, the global timing in a properly implemented distributed synchronous system can be predictable and fault-tolerant, although such a design often becomes inefficient. In fact, most software architectures and communication protocols used in safety-critical distributed real-time systems are time-triggered, i.e., all actions are initiated by temporal events that follow a global statically computed schedule. As argued by Kopetz et al. in [45, 44], such conservative designs attempt to avoid nonfunctional dependencies between components, and thus, support independent verification of each component. The time-triggered approach is preferred for its compositionality, but it becomes inefficient with increasing system complexity, especially if the network traffic is irregular. According to Richter et al. [62] it results in larger buffer size requirements, smaller utilization factors and larger power consumption. This overhead often drives designers towards asynchronous or event-triggered solutions. Some recent solutions use global time synchronization but support event-triggered processing, where the schedule unfolds dynamically during runtime, depending on the occurrences of different events. For instance, to preserve discrete-event model semantics in such a setting Zhao et al. [81] develops theoretical concept of relevant dependency. However, distributed Giotto can be thought of as purely software-based time-triggered architecture. Consequently, in our research on flexibility in software integration the focus is not on performance parameters.

Thesis Work. We present a compositional approach to the implementation of hard real-time software running on a distributed platform. We explain how several code suppli-

ers, coordinated by a system integrator, can independently generate different parts of the distributed software. The purpose of our effort [34, 36, 35] is to provide the application programmer with a programming interface that hides most of the implementation details (e.g. scheduling, handling of shared resources), but provides useful services (e.g. component communication and synchronization).

The task structure, interaction, and timing is specified as a Giotto program. Giotto is an example of a methodology based on a restricted model that attempts to reduce the sensitivity of unknown system information such as task execution time. A Giotto program executes a periodic set of LET (Logical Execution Time) 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. Therefore, the times when a LET task reads and writes data are decoupled from the task execution. The LET of a task is always non-zero. This allows us to communicate values across the network without changing the timing information of the task, and without introducing nondeterminism. Thus, LET tasks can be replaced and composed without modifying their behavior or timing.

We demonstrate how Giotto can be implemented on a distributed platform by distributed compilation with little global coordination. Each supplier is given a part of the Giotto program and a timing interface, from which the supplier generates task and scheduling code. 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. The integrator then checks, individually for each supplier, in pseudo-polynomial time, if the supplied code complies to the timing interface and meets, on the given hardware, the release and termination times specified by the Giotto program. If all checks succeed, then the supplied software parts are guaranteed to work together and

implement the original Giotto program. We demonstrate the feasibility of the approach by a prototype implementation.

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. 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.

1.3   Component Resource Abstraction and Tradeoffs

A general methodology for temporal protection in traditional real-time systems research is the resource reservation framework studied, for instance, by Lipari [51] and Almeida [2]. The idea is that each task, or a component of tasks, is assigned a server that is reserved a fraction of the processor available bandwidth: if a task tries more than it has been assigned, it is slowed down. This way one can isolate the unpredictability of execution times of different tasks or streams of tasks from each other. In such solutions, a failing component cannot influence the behavior of other components in the system, since there is a temporal isolation between components.

Recently, these methods were extended to hierarchical scheduling systems which consist of real-time components arranged in a scheduling hierarchy [56, 57, 61, 65]. This is a form of "divide and conquer" technique, where resource partitioning is performed over multiple levels. Each component consists of a real-time task workload and a scheduling policy for the workload. A resource is allocated by a higher to a lower scheduling level

through a *scheduling interface*. The interface specifies the resource requirement from the lower level and the resource guarantee from the higher-level scheduler. A hierarchical scheduling framework should exhibit *separation* among levels, i.e., the interface should be minimal. Moreover, the main benefits of hierarchical scheduling arise if the framework is fully *compositional*, i.e., if properties established at the lower also hold at the higher level.

These methods demonstrate how to perform composition of components in a hierarchical scheduling framework, but do not address the problem of generating the timing properties of a component. Shin et al. [68] defines this problem as *abstracting* the collective real-time requirements of a component as a single real-time requirement. This single requirement should be a sufficient and necessary requirement for all the collective requirements of the component. Abstraction of the internal complexity of a task group into a single requirement is used to reduce scheduling difficulties in the hierarchical scheduling framework.

Early work in task group abstraction by Lipari [51] or Shin [68] considers the *periodic resource model* (T, C), a resource abstraction under which a component is guaranteed to get C units of the resource every T units of time. This research showed how to abstract a group of independent periodic tasks with EDF (Earliest Deadline First) or RM (Rate Monotonic) scheduling algorithms into a single periodic task characterized with a pair (T, C). The exact procedures were given in [51] for a component with RM scheduling, and in [68] for a component with EDF scheduling. The compositionality of the framework was demonstrated by combining multiple scheduling interfaces into a single higher-level interface. The work by Easwaran et al. [19] is specific because the component at the topmost level can select a value for period T that minimizes the resource demand of the system. The corresponding periodic resource model is exported to the operating system for scheduling and the chosen value for period is propagated to all the components in the system where resource capacities are given by the corresponding interfaces. The component model by Almeida et al. [2] refines the periodic resource model by including release jitter, deadlines

10

earlier than periods and synchronization blocking. In addition, this is one of the rare efforts to study trade-offs between complexity and tightness of abstraction.

In [59] Mok et al. introduce another resource partition model, the *bounded-delay resource model*. The bounded delay resource model (c, δ), guarantees fraction c of the resource with at most δ time units of delay. This model is suitable when different components aimed at the same resource have considerably different latency requirements. Later work [69] by Shin shows how to abstract a set of independent periodic tasks into a bounded-delay interface. They also show how to use the bounded-delay model together with the periodic model as scheduling interface models, i.e., they show how to abstract a set of periodic and bounded-delay tasks into a single periodic or bounded-delay task.

Thesis Work. We showed that the previous results ([68, 69]) can be extended for supporting interacting tasks with data dependencies. We assume that all applications that execute on the considered resources are specified in the conventional periodic *task* model with an underlying task precedence graph. We study the periodic *resource* model for hierarchical scheduling model in the presence of dataflow constraints between the tasks within a group (intragroup dependencies), and between tasks in different groups (intergroup dependencies) [53].

We consider two natural semantics for dataflow constraints, namely, RTW (Real-Time Workshop) semantics and LET (logical execution time) semantics. While RTW follows the semantics of real-time code generated from a Simulink environment ([71]), LET has been used in Giotto domain-specific language, as discussed above. The most important semantics difference between the two models is as follows. The RTW scheme transfers the output of a task as soon as the task completes execution. The LET scheme makes the output of a task available at the prespecified time, namely, at the relative deadline defined by the task period.

We show that while RTW semantics offers better end-to-end latency on the task group

level, LET semantics allows tighter resource bounds in the abstraction hierarchy and therefore provides better composability properties. This result holds both for intragroup and intergroup dependencies, as well as for shared and for distributed resources. In addition, for a suitable chosen composability metrics, we prove some bounds on the composability difference between the two models. Finally, we show that, in contrast to the RTW semantics, the LET semantics both exhibits separation between levels and is fully compositional.

1.4 Interface-based Formalisms for Real-time Components

Although the performance verification community has different techniques than the real-time systems community, the goals are often similar [83]: to achieve high productivity, designers must work at higher levels of abstraction, reusing already designed and verified components. The goal of abstraction is to be able to verify correctness using the abstract interface without implementation or prototype. Thus, in both communities, the validation task is decomposed into the *analysis* of individual processes for which formal analysis techniques are known and on the *composition* of the results in order to obtain system-level timing information. However, the methods they use differ in either the analysis, or the composition parts, or both.

A group of methods by Ernst and his group integrate local analysis with a global event-flow based analysis, typically using existing models and analysis techniques [40, 63, 29]. To avoid traps of simulation these methods do not consider each event individually, but abstract events to *event streams*. Activating events may be aperiodic by nature, e.g. alarms, or periodic with jitter, e.g. packets in a communication protocol. Even strictly periodic task activation can be seen as event-driven, since it is the result of the expiration of a timer. Event streams are represented by standard event models, and the corresponding

compositional analysis methodology is based on the event propagation models. In most cases, the analysis requires only a few simple properties of event streams, such as event period, maximum jitter, or event burst. In such a context, global schedulability can be seen as flow-analysis problem for event streams that can be solved iteratively using event stream propagation. In principle, based on event stream manipulation one can identify worst-case scenarios, potentially even buffer overflows and missed deadlines as a result of transient overload.

In [40] the local analysis techniques are composed on the system level by connecting their input and output event streams. For such a compositional approach, it is required that the output event models of one component be compatible with the input event models of the connected components. Incompatible event models may also need to be connected by the overall application and communication structure. For instance, an aperiodic event model is to be connected to a periodic one. To overcome this problem certain transformation functions are defined and applied in order to adapt event models. In general, the method allows local scheduling results from the real-time systems research to be used, which is a major advantage over holistic analysis approaches such as the one by Pop et al. [60]. Another difference to the holistic approach is that the formal event stream equations are much better structured with respect to the architecture. The SymTA/S tool is based on this approach [29]. It supports heterogeneous architectures, complex task dependencies and context aware analysis, and it determines system-level performance data such as end-to-end latencies, bus and processor utilization, and worst-case scheduling scenarios. Furthermore, SymTA/S combines optimization algorithms with system sensitivity analysis for rapid design space exploration.

A similar compositional performance analysis approach is based on the *real-time calculus* by Thiele and his group [75, 11, 76, 72]. This approach is geared towards performance analysis of embedded and network processors and uses the event model representation known from the network calculus theory developed by Boudec et al. [7]. The work [75]

by Wandeler et al. is the first research effort that formally combines the network calculus and interface design theories in the real-time context. Each component represents a task, so there is no abstraction of task groups into components. Also, the task model in [75] assumes independent tasks, so interface compatibility checking does not have to take into account dataflow constraints. Finally, they assume preemptive fixed-priority scheduling, where each component (task) is specified with a certain priority. The research in [76] extends this work for other scheduling algorithms such as EDF and polling servers, whereas [72] moves from static interfaces by introducing formalism that can adapt system guarantees according to the system environment.

This approach is not limited to a particular task set characterization (e.g. periodic task set) or to a particular resource model (e.g. bounded-delay model). In contrast, they use network calculus notions of upper and lower bound event arrival curves for event streams, and service curves for resource modeling. This generality comes with a price. Since the event stream models are not the standard ones, new scheduling analysis procedures for the local components have to be developed. So, the existing work in real-time system research cannot be reused. Furthermore, the complexity of the equations makes the approach less intuitive than some simple local techniques such as rate monotonic analysis. As often advocated, a system-level analysis, especially a compositional or hierarchical one, should be comprehensible to be successful.

Thesis Work. In the area of interface-based timing verification we present an assume-guarantee interface algebra for real-time components [37]. This approach is based on interface theory methodology [13, 14] by de Alfaro et al. In general, the input/output behavior of a system component is captured by an automaton. Two interfaces are compatible if there is a way to use them together such that their input expectations are met. Thus, the interface automaton of a composition is constructed by pruning all violating states from the product of the component automata. In particular, the timed interfaces theory [15] can be applied when timing of inputs and outputs are important. A timed interface is specified as a timed

game between two players, representing the inputs and outputs of the component. However, in the results presented here, and since we always abstract events into a suitable event stream that can be represented by a simple predicate, the form of the interface is stateless, and thus the composition is much simpler. Our approach is also similar to the real-time interfaces approach of Wandeler et al. [75]. However, for the components we do not use general event and resource models, but only those for which effective resource abstraction results can be derived. Thus, we can extend and reuse some of the theory discussed in Sec. 1.3. The objective is to enable automatic, efficient, and flexible composition of such real-time interfaces.

In the first problem we address in this area, a component implements a set of *task sequences* that share a resource. The arrival rate function bounds the number of task (sequence) requests in a given interval of time. We show how to abstract such a task group using the bounded-delay or periodic resource models. Then we consider such a task group as a part, i.e., a component, of a larger real-time system specified with a set of task sequences that define task precedence constraints.

Due to the task dependencies between different components, the interface cannot just contain resource constraints, but also dataflow propagation constraints. A component interface consists of an arrival rate function and a latency for each task sequence, and a capacity function for each shared resource. A capacity function defines a fraction of processing power that is reserved for the component, or, more generally, a resource partition model such as bounded-delay model. The interface specifies that the component guarantees certain task latencies depending on assumptions about task arrival rates and allocated resource capacities. Together with interfaces we formally define interface composition operations, and the compatibility and refinement relation. Interface compatibility can be checked on partial designs, even when some component interfaces are yet unknown. In this case interface composition computes as new assumptions the weakest constraints on the unknown components that are necessary to satisfy the specified guarantees.

For efficient and flexible composability checking two properties are important: incremental design and independent refinement. According to the *incremental design property* the composition of interfaces can be performed in any order, i.e., it is associative, even if interfaces for some components are not known. Note that resource abstraction procedures described in Sec. 1.3 and most of other approaches described in Sec. 1.4 are not associative. However, we prove that our interface algebra satisfies the incremental design property. The refinement relation is defined such that in a design we can always substitute a refined interface for an abstract one. We show that the framework supports *independent refinement*, i.e., the refinement relation is preserved under composition operations. Our algebra thus formalizes an interface-based design methodology that supports both the incremental addition of new components and the independent stepwise refinement of existing components.

Little previous work exists that considers compositional performance analysis in the presence of complex task dependencies that include cycles. However, this is an important problem in practice since nonfunctional dependency cycles are often introduced by communication sharing as noted by Richter et al. [62]. The research presented by Yen et al. in [78] or Goddard et al. [24, 23] are notable examples, but these are holistic methods in real-time tradition, that do not allow compositional analysis. Zhou et al. [82] studies causality interfaces for general dataflow model, but the approach is targeted towards deadlock detection, and does not include real-time properties. In the SymTA/S tool limited set of cyclic graphs is allowed [63] in the models. For instance, a cycle can have only one external input. The cycles are analyzed by iterative propagation of event streams until the event stream parameters converge or until a process misses a deadline or exceeds a buffer bound. The iteration process terminates because the event timing uncertainty grows monotonically with every iteration, but that typically ends in uncompatibility error rather than in a fixed-point solution that satisfies interface constraints.

In the final part of the thesis we study interface-based verification of general task graphs, arbitrary directed graphs where each node represents a task, and each edge rep-

resents the data flow between tasks [54]. Since input degree of a node can be greater than one, a task may execute only after data is available on all input edges (AND type of task triggering). The graph is allowed to have cycles, i.e., we allow for cyclic functional dependencies between tasks. We assume that the primary inputs of a task graph are specified with event arrival curves that bound the number of task executions. In particular, we concentrate on periodic event models with jitter and burst. In order to avoid iteration problems of [63] one has to specify also phase information between events in different event streams. Therefore, our objective is to define the form of interface and interface operations that would enable flexible interface-based design similar to the case of task sequences. In this case, three operations are needed for construction of composite task graphs, the composition, connection, and join operations. Finally, we study requirements that enable the incremental design and independent refinement properties discussed above.

1.5 Thesis Organization and Contributions

We now present the organization of the thesis and the main results of each chapter.

- In Chapter 2 we present a compositional approach for the implementation of hard real-time software running on a distributed platform. We explain how several code suppliers, coordinated by a system integrator, can generate parts of the distributed software in a distributed manner. We present the algorithm that generates the necessary Giotto code and timing interface for each host and each supplier. We also 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 formally prove the distributed Giotto compiler correct. The feasibility of the approach is demonstrated by a prototype implementation. A preliminary version of this Chapter appeared in [35].

- In Chapter 3 we first study the abstraction of a task group that executes on a single resource and with precedence constraints among tasks within the group (intragroup task precedences). We show the tightness difference in favor of the LET semantics. For the case of a task group distributed over several resources we characterize how large the gap in the tightness of abstractions between the two schemes, RTW and LET, can be. In the context of higher levels of the hierarchical scheduling framework, we allow for the task precedences among different task groups (intergroup task precedences). The LET semantics again results in tighter and simpler abstractions. In addition, and contrary to the RTW semantics, we show that the LET semantics enables a compositional framework with separation between levels. The results of this Chapter were published in [53].

- In Chapter 4 we first study real-time components consisting of task sequences [37]. We give procedure to obtain resource partition parameters for a group of aperiodic tasks given with arrival rates and deadlines. The right form of the interface and corresponding algebra are presented and discussed. We formally prove that the framework satisfies incremental design and independent refinement properties. The approach is then extended for the case of task graph components that include task cycles [54]. Due to the different event model a different interface algebra has to be defined. The two properties adapted to allow cycles are shown to hold even for this interface-based design methodology.

- Chapter 5 concludes the thesis and gives some pointers for future research. The relevant problems and concluding remarks are given at the end of each chapter.

Chapter 2

Composable Code Generation for Distributed Giotto

2.1 Introduction

In this chapter 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 base our work on the LET (*logical execution time*) paradigm, and the LET-based language Giotto, previously proposed as a software model that guarantees predictable real-time execution and at the same time supports portable, composable code [31]. The chapter demonstrates 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 is a domain-specific language for control applications [31]. 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) [35] for a pair of virtual machines: the E (embedded) machine mediates between tasks and the physical environment [32]; the S (scheduling) machine mediates between tasks and the CPU [35]. 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 [42], and used Giotto to successfully implement flight control systems for model helicopters [30].

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.

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

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 [43], 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 [70, 3]) require that all component processes, their dependencies, and their timing profiles be known in advance. 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 sepa-

rately 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 [26] does not offer this flexibility, and hence an important approach to distributing synchronous programs is based on the Globally Asynchronous, Locally Synchronous paradigm [4].

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.

Previously, Giotto had only been compiled for single-CPU systems [33]. The contribution of this chapter 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 [10]. Aimed at (1) are scheduling techniques that address the problem of dividing tasks into groups, and scheduling tasks within groups [58, 68]: 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 [46] 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; [64] presents requirements for the temporal partitioning of such a platform. The car manufacturers' and suppliers' perspectives on embedded software reuse are described in [28], 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 Chapter. In Sec. 2.2 we present a brief review of Giotto and introduce a running example that we will use throughout this chapter. In Sec. 2.3 we discuss the algorithm that generates from a given Giotto program virtual machine code (SCC) for each host and each supplier. In Sec. 2.4 we introduce timing interfaces and show how they can be composed. Sec. 2.5 describes our prototype implementation of distributed Giotto. In Sec. 2.6 we give the formal semantics of distributed SCC, 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.

2.2 Giotto Language

We give a brief introduction to Giotto and refer to [31] for details. A simple example of a Giotto program G_A is shown in Fig. 2.1. 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 [41], 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. 2.1 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. 2.2 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

```
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 2.1. Audio mixer Giotto program G_A

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 activity performed by the task. For example, the task $Mixer$ is defined with input port In_2, output port $MixSound$, and task function $task[Mixer]$. Let $Tasks$ be the set of tasks. 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
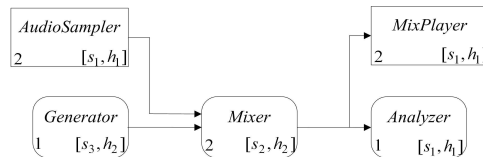
Figure 2.2. Data dependency graph for the program G_A

25

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 $Drvs$ be the set of 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 $m' \in Modes$, 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 chapter we will refer to the single-mode program in Fig. 2.1. 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. 2.3.

```
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 2.3. 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, and is 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 that can, given a mode $m \in Modes$ and an integer unit $0 \le k < \omega_{max}[m]$, be directly determined from the Gitto 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, i.e., at time $4ms$. 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 a task 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. In [31] 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 [32] 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

27

`release`(t) instruction 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 $\ell\ ms$ elapse. The positive integer ℓ specifies a time trigger, the simplest and only form of trigger that we consider in this chapter. 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. 2.4, 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 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.

$E(m_1, 0)$:
  `call`$(copy[Spectrum])$
  `call`$(copy[MixSound])$
  `call`$(copy[StringSound])$
  `call`$(ActDrv)$
  `call`$(dev[MixPlayer])$
  `call`$(dev[AudioSampler])$
  `call`$(InDrv_1)$
  `call`$(InDrv_2)$
  `call`$(InDrv_3)$
  `release`$(Analyzer)$
  `release`$(Mixer)$
  `release`$(Generator)$
  `future`$(4, E(m_1, 1))$

$E(m_1, 1)$:
  `call`$(copy[MixSound])$
  `call`$(ActDrv)$
  `call`$(dev[MixPlayer])$
  `call`$(dev[AudioSampler])$
  `call`$(InDrv_2)$
  `release`$(Mixer)$
  `future`$(4, E(m_1, 0))$

Figure 2.4. E code blocks for the program G_A

```
S(m₁, 0):                      S(m₁, 1):
  dispatch(Mixer, 4)             dispatch(Mixer, 4)
  dispatch(Generator, 4)         dispatch(Generator, 4)
  dispatch(Analyzer, 4)          dispatch(Analyzer, 4)
```

Figure 2.5. S code blocks for the program G_A

The scheduling machine [31] 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 $\ell\ ms$ 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 chapter. 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. 2.5 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. 2.4 and 2.5 the *Mixer* task is executed twice every $8ms$, and the tasks *Generator* and *Analyzer* once, exactly as specified by the Giotto program G_A.

2.3 Composable Design with Giotto

Distributed Code Generation Flow. 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 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 chapter 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. 2.1 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. 2.2. 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 wcet's 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

31

the original Giotto program G. Moreover, once a supplier modifies its S code module on a host, to check if Giotto semantics is preserved, it is sufficient to check only if the new module complies to its timing interface.

From Giotto to Distributed E Code. Let P be entire distributed SCC program. The set $Ports_P$ of distributed SCC ports contains additional ports ($Ports \subseteq Ports_P$) needed 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 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 at the end of the task period by an E code instruction. 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 preceeds task execution, which preceeds the transmission of task output ports.

More precisely, let d be the task input driver for a task t asigned 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

32

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 preceeds 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. Symetrically, 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 input ports can be updated with new values.

Given a Giotto program, Algorithm 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 [33] for details on generating a block of E code instructions that addresses mode switching; this is orthogonal to the issues discussed in this chapter. The last segment handles `call` instructions for sensor device drivers,

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* [33]

  $\forall p \in senPorts(m, k):$

    $emit(\bar{s}(p), \bar{h}(p), \texttt{call}(dev[p]));$

    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

---

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 similar to tasks. So, in order to simplify notation we also use the same SCC instructions for messages. The instruction `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 `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 `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. 2.6 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 `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 the task output port $p \in OutPorts$

$E_{s_1,h_1}(m_1, 0)$:
    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 2.6. E code modules for the program G_A compiled by Alg. 1

is a source port of an actuator, mode switch, or task input driver that executes at a host h in a mode m, then $h \in recHosts(p) \cup \{\bar{h}(p)\}$. 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 corect value even in the first period of execution in the mode.

2.4   Timing Interfaces

As presented in Section 2.3, 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 $m \in Modes$ of the Giotto program G a *timing interface*, a pair of predicates $T_{s,h}^m = (D_{s,h}^m, X_{s,h}^m)$. The predicates $D_{s,h}^m, X_{s,h}^m : \{0, ..., \pi[m] - 1\} \rightarrow \{0, 1\}$ are defined as follows:

- $D_{s,h}^m(\ell) = 1$ iff in mode m at time ℓ supplier s on host h may execute a task from $Tasks_{s,h}$;

- $X_{s,h}^m(\ell) = 1$ iff in mode m at time ℓ supplier s on host h may send a message from $Msgs_{s,h}$.

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

Fig. 2.7 shows a graphical representation of a timing interface for the program G_A from Fig. 2.1. 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. 2.6, 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.
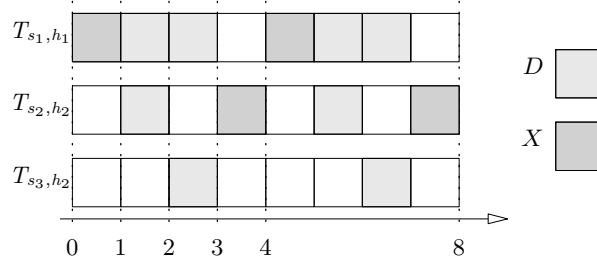
Figure 2.7. Timing interface for the program G_A

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 interface 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 [43], can be modeled as two hosts.

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. 2.7.

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 $m \in Modes$, 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

38

– 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 $m \in Modes$, units $k \in \{0, ..., \omega_{max}[m] - 1\}$, ports $p \in SensePorts \cup OutPorts$, and times $\ell \in \mathbb{N}_0$, if either

  – $p \in senPorts(m, k)$ and $k \cdot \gamma[m] \leq \ell < k \cdot \gamma[m] + \epsilon$, or

  – $p \in taskOutPorts(m, k+1)$ 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 $h \in recHosts(p)$.

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., 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 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., let for all $\ell \in [\ell_1, \ell_2)$ be $D^m_{s,h}(\ell) = 1$. Let $t_1, t_2, ..., t_{|Tasks_{s,h}|}$ 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 $\mathcal{S}_{s,h}$ contains the following sequence of instructions:

```
idle(ℓ₁ − kγ[m])
dispatch(t₁, ℓ₂ − kγ[m])
dispatch(t₂, ℓ₂ − kγ[m])
...
dispatch(t_{|Tasks_{s,h}|}, ℓ₂ − kγ[m])
```

The entire EDF S code module consists of such code segments for each computation or communication slot of the timing interface. The Fig. 2.8 shows EDF S code modules for Giotto program G_A generated using timing interface shown in Fig. 2.7.

```
S_{s₁,h₁}(m₁, 0):                      S_{s₁,h₁}(m₁, 1):
  call(InDrv₁)                           dispatch(μ[MixPlayer], 1)
  dispatch(μ[MixPlayer], 1)              idle(1)
  idle(1)                                dispatch(Analyzer, 3)
  dispatch(Analyzer, 3)

S_{s₂,h₂}(m₁, 0):                      S_{s₂,h₂}(m₁, 1):
  idle(1)                                idle(1)
  call(InDrv₂)                           call(InDrv₂)
  dispatch(Mixer, 2)                     dispatch(Mixer, 2)
  idle(3)                                idle(3)
  dispatch(μ[MixSound], 4)               dispatch(μ[MixSound], 4)

S_{s₃,h₂}(m₁, 0):                      S_{s₃,h₂}(m₁, 1):
  call(InDrv₃)                           idle(2)
  idle(2)                                dispatch(Generator, 3)
  dispatch(Generator, 3)
```

Figure 2.8. S code modules for the program G_A

2.5   Implementation

Our test system consists of several off-the-shelf PC hosts with 200Mhz PentiumPro 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 [79]. 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 $30\mu s$.

Real-time communication is attained through a special network driver [47] 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. 2.9. 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 time of a single slot, and N is the number of hosts operating under the time-triggered protocol, then the cycle repeats after time $T_0 \cdot (N + 2)$.

Sync	Resync	Master Send	Client 1 Send	...	Client N-1 Send
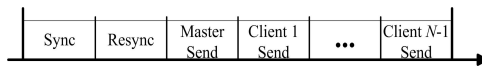
Figure 2.9. Cycle of the communication protocol [19]

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 [42]. To implement distributed SCC correctly we make use of special RTLinux calls that suspend and resume task threads.

41

To test the virtual machine we implemented the audio application G_A through the distributed SCC program shown in Fig. 2.6 and 2.8. Note that in Fig. 2.8 each `dispatch` instruction with a task (resp. messsage) as an argument executes in computation (resp. communication) slots shown in Fig. 2.7. In this setup each time slot lasts $T_0 = 1ms$, and an entire communication cycle lasts $4ms$ (N=2). In this configuration the maximum bandwidth available to each host is $2.86Mbit/s$. The tests show that the sound card is fed continuously with samples. The audio reproduced back at h_1 plays without any noticable interruption or other sound defects.

The estimated overhead of the network driver synchronization thread is $25\mu s$. 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 $12\mu s$ (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 $2\mu s$, 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.

2.6 Compositional SCC Analysis

We first characterize distributed SCC program compiled from a Giotto program G according to the scheme presented in Section 2.3. The program is represented as a state transition system that is then used to verify correctness of such an implementation of G.

2.6.1 Giotto-Generated Distributed SCC

We start by describing E and S code modules separately, and then define entire distributed SCC program. 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 con-

stant. Let $g_{s,h}$ be equal to $|Tasks_{s,h}| + |Msgs_{s,h}| + |Drvs_{s,h}|$, i.e., let $g_{s,h}$ represent the size of the part of program G 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 from $Modes$ and k is a unit od mode m, i.e. $k \in \{0, ..., \omega_{max}[m]\}$. The graph $(V^{\mathcal{E}}_{s,h}, E^{\mathcal{E}}_{s,h})$ has following properties:

- Each path from a source to a sink consists of

  - a sequence of $O(g_{s,h})$ edges e, each with $\kappa(e) = $ `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 $\kappa(e) = $ `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 $\kappa(e) = $ `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 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, 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 $O(M^2 \cdot n)$. Since we consider the number of modes to be fixed, we have that 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 control-flow directed graph $(V_{s,h}^{\mathcal{S}}, E_{s,h}^{\mathcal{S}})$, two node-labeling functions ρ and ν, and an edge-labeling function λ. We require that the graph $(V_{s,h}^{\mathcal{S}}, E_{s,h}^{\mathcal{S}})$ 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:

- $\rho(u) = \texttt{dispatch}$, $\nu(u) \in \mathit{Tasks}_{s,h} \cup \mathit{Msgs}_{s,h}$ and node u has a successor u' such that $\lambda(u, u') \in \mathbb{N}_{>0}$. If $\nu(u) \in \mathit{Tasks}_{s,h}$ 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 $\nu(u) \in \mathit{Msgs}_{s,h}$ 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 $\lambda(u, u') \in \mathit{Drvs}_{s,h}$. 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 $V_h^{\mathcal{E}}$ be the union of node sets $V_{s,h}^{\mathcal{E}}$ 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 $v' \in V_{s,h}^{\mathcal{E}}$ to a source node $\Phi_{s,h}(v') \in V_{s,h}^{\mathcal{S}}$ such that if $(v, v') \in E_{s,h}^{\mathcal{E}}$, $\kappa(v, v') = \texttt{future}$ and $\lambda(v, v') = (\ell, \cdot)$ then the chain in $(V_{s,h}^{\mathcal{S}}, E_{s,h}^{\mathcal{S}})$ that starts from node $\Phi_{s,h}(v')$ does not contain numbers, i.e., clock timeouts in $\texttt{dispatch}$ and $\texttt{idle}$ 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 $(V_{s,h}^{\mathcal{E}}, E_{s,h}^{\mathcal{E}})$ and $(V_{s,h}^{\mathcal{S}}, E_{s,h}^{\mathcal{S}})$ 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.

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 $v_h \in V_h^{\mathcal{E}}$, 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 $\tau_h \subseteq (\mathbb{N}_0 \times V_h^{\mathcal{E}})^*$ 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 $u \in V_h^{\mathcal{S}}$ 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.

Section 2.6.2 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 `dispatch` or `idle` instruction becomes true; and a *transient S transition* when an S code `call` instruction is executed. The transition rules impose an order on transitions of different type. For instance, if an E transition and a timeout S transition occur at the same time, then an enabled trigger must be processed before any expired timeout is

45

handled, because the E code may release tasks that require immediate dispatching service from the S code.

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 $x \in Tasks_{s,h} \cup Msgs_{s,h}$, x completes with execution (transmission) time at most $w_{s,h}(x)$.

2.6.2 Formal Distributed SCC Semantics

In [34] 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 message output ports $Out[\mu[p]]$ be $\{p_h \mid h \in recHosts(p)\}$ and let a message function $task[\mu[p]]$ be identity function from the message input to output ports.

The state $q = (r, v, c, \tau, \theta)$ has a *transition* to the state $q' = (r', v', c', \tau', \theta')$ 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 $c(\nu(u)) = \bot$ 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 $\theta'_h = (\theta_h \backslash \{(u, \delta)\}) \cup \{(u', \delta)\}$.

46

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) = $ call, and the successor u is 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. If (1) let $(0, \bar{v})$ be the first such pair in τ_h. Then $p = p'$, $v' = v$ except that $v'_h = \bar{v}$, $c' = c$, $\tau' = \tau$ except that $\tau'_h = \tau_h \backslash \{(0, \bar{v})\}$ and $\theta' = \theta$. If (2) then one of the following: (a) $\kappa(v_h, v'_h) = $ 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) = $ release and $r' = r$, $c' = c$ except that $c'(\lambda(v_h, v'_h)) = 0$, $\tau' = \tau$; (c) $\kappa(v_h, v'_h) = $ 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 $\rho(u) \in \{$dispatch, idle$\}$, the successor of u is u', $\lambda(u, u') \in \mathbb{N}_0$ and $\lambda(u, u') \leq \delta$. Then $(r', v', c, \tau') = (r, v, c, \tau)$, $\theta = \theta'$ 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 $p \in Ports_P \backslash \{p_c\}$ and $r'(p_c) = r(p_c) + 1$. For $\ell = r(p_c)$ we call function $r_\ell = r$ the *port valuation at time* ℓ. For this transition it also holds $v' = v$ and for each $h \in H$ we have: (1) the queue τ'_h results from τ_h by replacing each trigger binding (δ, u) by $(\delta - 1, u)$, (2) the thread set θ'_h results from θ_h by replacing each thread (u, δ) by $(u, \delta + 1)$, (3) let $X_h = \{x \mid (u, \cdot) \in \theta_h, \rho(u) = $ dispatch, $\nu(u) = x\}$ and let $\bar{x} \in X_h$ be a task

or message to be executed on h; 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 $x = \bar{x}$, and $c'(x) = c(x)$ if $x \neq \bar{x}$; in case $c'(x) = \bot$ we say that on the transition (q, q'), task or message x *completes* after execution time $c(x) + 1$.

2.6.3 Interface Compliance and Time Safety

For the compositional analysis of a distributed SCC program we need the following two properties. Let G be a multiple-mode Giotto program, $T_{s,h}$ a timing interface for a supplier s and a host h, $P_{s,h}$ a G-generated SCC module, and $w_{s,h}$ 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. 2.6 and 2.8 interface-complies with the timing interface $T_{s,h}$ shown in Fig. 2.7 because the S code in Fig. 2.8 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 [33]. 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. 2.6 and 2.8 is time-safe. For example, in P_{s_2,h_2}, input ports of 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 time $2ms$.

We give the formal definitions of the two properties on the program state to be clear that they can be checked in constant time:

- 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 $\rho(u) = \mathtt{dispatch}$, $\eta(v_h) = (m, k)$, and either (1) $\nu(u) \in Tasks_{s,h}$ and $D_{s,h}^m(k\gamma[m] + \delta) = 0$, or (2) $\nu(u) = Msgs_{s,h}$ and $X_{s,h}^m(k\gamma[m] + \delta) = 0$. 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 $\kappa(v_h, v_h') = \mathtt{call}$ and $\lambda(v_h, v_h') = d$ (E code driver), or (b) there exists a thread $(u, \cdot) \in \theta_h$ with $\rho(u) = \mathtt{call}$, 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 $c(x) \neq \bot$, 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 [34] discusses time safety checking for single-mode, single-CPU Giotto programs. These results are here generalized to both the distributed and multiple-mode settings. For distributed single-mode programs we give an efficent algorithm that checks if $P_{s,h}$ complies to a given interface and if it is time-safe. For distributed multi-mode programs we give a sufficient condition that can be efficiently checked.

Let a G-generated SCC module be given with 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 $V_{s,h}^{\mathcal{E}}$ (resp. $V_{s,h}^{\mathcal{S}}$) to a source of $V_{s,h}^{\mathcal{S}}$ (resp. $V_{s,h}^{\mathcal{E}}$) determined by the map $\Phi_{s,h}$ and control flow of $\mathcal{E}_{s,h}$. If G is a single-mode program each graph $\mathcal{P}_{s,h}$ is a chain.

We next argue that graph $\mathcal{P}_{s,h}$ is an acyclic graph even if G is a multi-mode program.

For instance, let Giotto program G_A have both mode m_1 and mode m_2 given in Fig. 2.3. The Fig. 2.10 shows a graph in which each edge abstracts a chain of $O(g_{s,h})$ edges of the graph $\mathcal{P}_{s,h}$. As discussed with respect to the Algorithm 1 and as defined by the Giotto semantics, 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 $E^P_{s,h}$ 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 $\mathcal{P}_{s,h}$ is an acyclic directed graph.
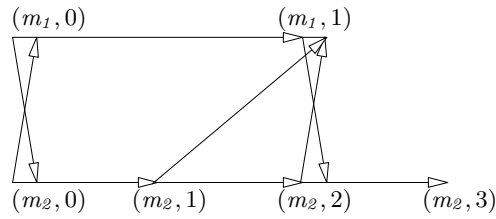
Figure 2.10. Graph related to $\mathcal{P}_{s,h}$ for G_A with additional mode m_2

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 is 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

50

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 suffcient 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 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 [33]. However, if G is a single-mode program, then the state-transition graph $\mathcal{P}_{s,h}$ is a disconnected 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 $(V_{s,h}^{\mathcal{E}}, E_{s,h}^{\mathcal{E}})$ and $(V_{s,h}^{\mathcal{S}}, E_{s,h}^{\mathcal{S}})$ 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 proposition:

Proposition 1 *Let G be a single-mode 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.*

Note that for multi-mode Giotto the pseudo-polynomial check is sufficient but not necessary.

2.6.4 Distributed Code Generation Correctness

We show that LET semantics of a Giotto program is preserved by the distibuted 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, and this property is what we define first.

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_ℓ^G and r_ℓ^P be the port valuation functions at time $\ell \in \mathbb{N}_0$ for G and P [31]. 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., if $r_\ell^G(p) = r_\ell^P(p)$ for each sensor port $p \in SensePorts$ (resp. $p \in ActPorts$) and each time instant $\ell \in \mathbb{N}_0$. We say that (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).

We say that (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 $h \in H$.

Proposition 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.*

We first give informal explanation why interface feasibility, interface compliance, and

52

time safety ensure correctness of the implementation. If interface feasibility condition is not satisifed, e.g. 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 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 any of the two time safety conditions, e.g. a time slot in the interface is not suffcently large, a task or message invocation may result in incorrect output.

Proof. Note first that the resource sharing property of T and interface 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 resource sharing property of T and interface 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 $\nu(u) \in Msgs$. So, if T is feasible and (P,w) interface-complies to T then there are no resource sharing conflicts.

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

Lemma 1 *If* $p \in OutPorts$, $h \in recHosts(p) \cup \{\bar{h}(p)\}$, *then* $r_\ell^G(p) = r_\ell^P(p_h)$ *for any time* $\ell \in \mathbb{N}_0$.

Proof.[Lemma] We use induction on time ℓ. For time $\ell = 0$ the statement holds because the initialization driver $init[p]$ is called on $\bar{h}(p)$ and $init[p_h]$ is called on all $h \in recHosts(p)$ (E transitions with $\texttt{call}$ instructions). They set p and p_h to initial $r_0^G(p)$ value.

Since $p \in OutPorts$ there exists a task t such that $p \in Out[t]$ and $t \in Tasks_{s,\bar{h}(p)}$ for some $s \in S$. In the code generated by the Algorithm 1 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 $copy[p_h]$ (call E transition) if $t \in taskOutPorts(m, k)$ 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 $t \in taskOutPorts(m, k)$, so we only have to prove that $r^P(p_h)$ is modified with a correct value.

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

1. $h = \bar{h}(p)$:

   Let ℓ' be the last time instant task t was released before ℓ. Let the mode and the unit of the corresponding $\mathtt{release}$ E transition be m' and k' respectively, $t \in taskOutPorts(m', k')$. Let d be the task t input driver, i.e. $(\cdot, t, d) \in Invokes[m']$, and let p' be an input port of d, $p' \in Dst[d]$. By the definition of the $recHosts$ operator we have $h \in recHosts(p') \cup \{\bar{h}(p')\}$.

   - If $p' \in OutPorts$ by induction hypothesis we also have $r_{\ell'}^G(p') = r_{\ell'}^P(p'_h)$.

   - If $p' \in SensePorts$ and $\bar{h}(p') = h$ the port p' is updated on the host h at time ℓ' by execution of $dev[p']$ driver ($\mathtt{call}$ E transition) and by input-compatibility assumption we have $r_{\ell'}^G(p') = r_{\ell'}^P(p'_h) = r_{\ell'}^P(p')$.

   - Let $p' \in SensePorts$ and $h \in recHosts(p')$. According to the Algorithm 1 and input-compatibility the driver $dev[p']$ is invoked at the unit k' on the host $\bar{h}(p')$ and the message $\mu(p')$ with the port p' value $r_{\ell'}^G(p')$ is released $\mathtt{release}$ E transition). If the program (P, w) is time-safe, then $(P_{s,h}, w)$ is also time-safe. Therefore, the message transmission completes before time $\ell' + \epsilon$ because at this time instant driver d is called and sharing property should not hold. By assumption, the data reception property is satisfied throughout the message transmission, so the message completion S transition correctly updates the port p'_h.

So, for all $p' \in Dst[d]$ we have $r_{\ell'}^G(p') = r_{\ell'+\epsilon}^P(p')$. 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 $\ell' + \epsilon$, but completed (completion S transition) by time $\ell - \epsilon$ at which the local copy of p is updated. So, $r_\ell^G(p) = r_\ell^P(p)$ for all $p \in Out[t]$. Since $h = \bar{h}(p)$ we have $r_\ell^G(p) = r_\ell^P(p_h)$.

2. $h \in recHosts(p)$:

   By the similar argument as above it can be proved that $r_\ell^G(p) = r_\ell^P(p)$. According to the Algorithm 1 on the host $\bar{h}(p)$ the message with the port p value $r_\ell^G(p)$ is released (`release` E transition). Again, time safety and data reception properties ensure that the message is transmitted to the host h after the task t completes but before time ℓ. Since $h \in recHosts(p)$ the driver $copy[p_h]$ is invoked on the host h at time ℓ and we have $r_\ell^G(p) = r_\ell^P(p_h)$.

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 $p \in ActPorts$ and let $h = \bar{h}(p)$. The code in P generated by the Algorithm 1 updates p in mode m at unit k only if $p \in actPorts(m,k)$. The same is true for the execution of the Giotto program G. Let d be an actuator driver such that $p \in Dst[d]$. Since each driver input port $p' \in Src[d]$ is also in the set of task output ports $OutPorts$ and since by the definition of the $recHosts$ operator $h \in recHosts(p') \cup \{\bar{h}(p')\}$ by the lemma we have $r_\ell^G(p') = r_\ell^P(p'_h)$. After applying driver function $drv[d]$ on $Dst[d]$, which updates p on h, we have $r_\ell^G(p) = r_\ell^P(p_h) = r_\ell^P(p)$. □

The compositional nature of interface compliance and time safety of (P,w) ensures that if, for some $s \in S$ and $h \in H$, one module $P_{s,h}$ is modified, then for P to implement G it is sufficient to check if $(P_{s,h}, w_{s,h})$ interface-complies with $T_{s,h}$ and if it is time-safe. So, combining propositions 1 and 2 we have:

Corollary 1 *Let G be a single-mode 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.*

Again, for multi-mode Giotto the pseudo-polynomial check is 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_{s3,h2}(m_1, 0):              S_{s3,h2}(m_1, 1):
  call(InDrv_3)                   call(InDrv_4)
  call(InDrv_4)                   idle(2)
  idle(2)                         dispatch(Generator_2, 3)
  dispatch(Generator_2, 3)        dispatch(Generator, 3)
  dispatch(Generator, 3)          idle(4)
  idle(4)
```

2.7   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. Given a timing interface, EDF S code was proved an optimal strategy with respect to schedule feasibility. Hence 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. The following chapter explores the related tradeoffs further.

Chapter 3

Component Resource Abstraction and Tradeoffs

3.1   Introduction

As the number of applications that share the same resources increases, the integration of software components in real-time or embedded systems becomes more pertinent. In this chapter we further study composability and resource abstraction of the LET programming model introduced in the previous chapter, comparing it with some other commonly used models of computation.

A key challenge is to have real-time assurance and, at the same time, a high degree of flexibility in component integration. This problem is addressed within an *open* real-time system [16] that consists of mutually *independent* components with sets of tasks of different time criticality. Research in open real-time systems concentrates on partitioning and scheduling schemes that make both the implementation and temporal behavior of a component independent of the presence of other components in the system. However, embedded real-time systems, e.g., automotive [46] or aircraft [64] systems, are often put

together from several *interacting* software components corresponding to different control loops. The problem is even more demanding when components have to be implemented by different suppliers [28].

In an open system, schedulability analysis and the admission test for a task group cannot depend on the properties of any other task group in the system. In recent years work in the composition for open systems has shifted towards *hierarchical* scheduling frameworks [56, 59, 67, 61], which extend resource partitioning over multiple levels. In such a framework a resource is often allocated by a higher to a lower scheduling level through a *scheduling interface*. The interface specifies the resource requirement from the lower level and the resource guarantee from the higher-level scheduler. A hierarchical scheduling framework should exhibit *separation* among levels, i.e., the interface should be minimal. Moreover, the main benefits of hierarchical scheduling arise if the framework is *compositional*, i.e., if properties established at the lower also hold at the higher level.

Abstraction of the internal complexity of a task group into a single requirement can be used to ensure the favorable properties and to reduce scheduling difficulties in the hierarchical scheduling framework. Early work in task group abstraction [68, 51] considers the *periodic resource* model (T, C), a resource abstraction under which a component is guaranteed to get C units of the resource every T units of time. This research showed how to abstract a group of independent periodic tasks with EDF or RM scheduling algorithms into a single periodic task. The compositionality of the framework was demonstrated by combining multiple scheduling interfaces into a single higher-level interface. Whereas these initial efforts with the periodic resource model addressed only independent periodic tasks, more recent efforts considered sets of tasks with blocking synchronization operations [2, 52].

In this chapter we study, under the same periodic resource model, hierarchies of tasks with *data dependencies*. Namely, we assume that all applications that execute on the con-

sidered resources are specified in the conventional periodic task model with an underlying task precedence graph. We first (Sec. 3.2) discuss two different application interpretations, i.e., we present two semantics, RTW and LET, which differ in the propagation of data between tasks. While LET was introduced in Chapter 2, RTW follows the semantics of real-time code generated from a Simulink environment [71]. The RTW scheme transfers the output of a task as soon as the task completes execution. The LET scheme makes the output of a task available at the prespecified time, namely, at the relative deadline defined by the task period. Compared to the RTW semantics, this typically increases the latency.

The composition with abstracted components inevitably incurs higher resource utilization than the component utilization sum. Therefore, effectiveness of composition can be compromised. If component abstraction is too coarse, only a few components can be correctly composed, and the rest may be disallowed on the admission test, even when actual required resource utilization is low. Therefore, we focus on *tight* abstractions, i.e., abstractions that minimize lower level resource requirements. We show that the tightness of abstractions, and therefore composability, depends on the application semantics. So, although at the lower levels the end-to-end latency is less for the RTW semantics, at the higher levels, when task group abstraction is taken into account, the LET semantics permits tighter abstractions.

Outline of the Chapter. We compare the composability of the two data transfer semantics in several scenarios. Sec. 3.3 studies the abstraction of a task group that executes on a single resource and with precedence constraints among tasks within the group (*intragroup* task precedences). We show that the tightness difference in favor of the LET semantics can come from the underlying scheduling algorithm used to implement a particular semantics. Sec. 3.4 generalizes the result for the case of a task group distributed over several resources. We characterize how large the gap in the tightness of abstractions between the two schemes, RTW and LET, can be. Moreover, we show that with LET semantics both abstraction and scheduling is simpler. This is important for hierarchical open systems, since complicated

interaction between scheduling levels increases unpredictability in task execution. Finally, Sec. 3.5 studies higher levels of the hierarchical scheduling framework. In this context task precedences among different task groups are allowed (*intergroup* task precedences). The LET semantics again results in tighter and simpler abstractions. In addition, and contrary to the RTW semantics, the LET semantics enables a compositional framework with separation between levels.

3.2 Multirate Task Programs

Let $\mathbb{Q}$ be a finite set of numbers that are all multiples of a certain sufficiently small unit rational number, and let $\mathbb{R}$ be the set of real numbers. A *task $t = (p, e)$* consists of a period $p \in \mathbb{Q}$ and a worst-case execution time requirement $e \in \mathbb{R}$. A *task graph $G = (V, E)$* is a directed graph with a set of tasks V and a set of task data dependencies $E \subseteq V^2$. In general, a program may exhibit multirate behavior because each task is characterized by its own period. Therefore, the program is fully specified only with the semantics of data transfer between tasks. In this chapter we assume that all task graph edges comply with the same semantics. In particular, we focus on two dataflow semantics, RTW and LET.

Real-Time Workshop semantics. Real-Time Workshop (RTW) [71] is a tool for automatic code generation in the MATLAB/Simulink environment. For a given task graph the tool generates multithreaded code, one thread per each sample time of the graph. The code is supposed to run on an RTOS that offers a priority based preemption mechanism. Each task is assigned to a thread based on its period, and the schedule within a thread is constructed from the task dependencies. The rate monotonic (RM) scheduler invokes the generated code, enabling preemption between rates.

To make multirate models operate correctly in real time, the program is implicitly modified by placing *rate transition* blocks, hold or delay blocks, between tasks that have unequal
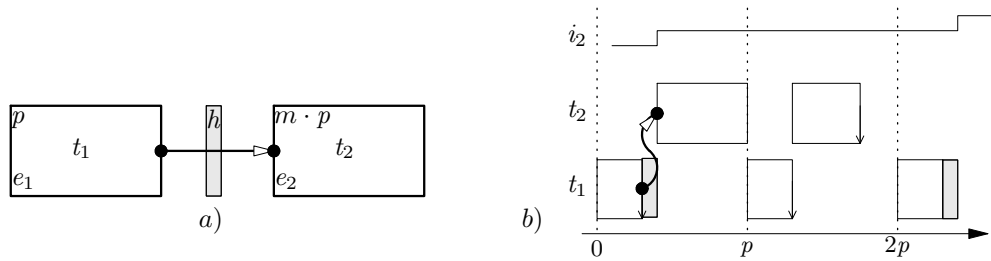
Figure 3.1. RTW: fast to slow data transfer - (a) task graph; (b) task and signal timeline for $m = 2$

periods. The rate transition blocks are assumed to execute in negligible time. Consider the data dependency shown in Fig. 3.1(a), where the period of the data consumer task t_2 is a multiple of the period of the data producer task t_1. A problem of data integrity exists when the input to task t_2 changes after its execution starts. Also, the output is nondeterministic and depends on how late t_2 starts. Adding a hold block h ensures that the second invocation of t_1 does not overwrite the data. The hold block executes with the slower period of t_2, but with the higher priority of the faster task t_1. In that way, it executes before task t_2 and its output value is held constant while t_2 executes. Beside the schedule for the tasks, Fig. 3.1(b) shows the input signal i_2 of t_2 over time, assuming incrementing functionality of t_1.
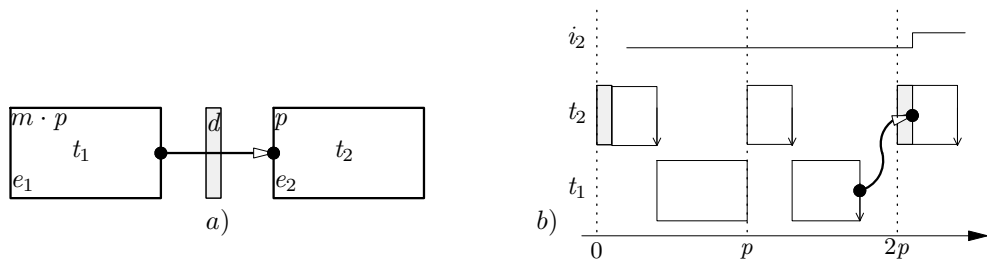
Figure 3.2. RTW: slow to fast data transfer - (a) task graph; (b) task and signal timeline for $m = 2$

In the inverse case shown in Fig. 3.2, the period of the data producer task t_1 is a multiple of the period of the data consumer task t_2. The delay rate transition block d compensates for the varying execution time of t_1, i.e., it makes the time of data transfer deterministic no matter how early t_1 completes. The delay block executes with the period of t_1, but with the higher priority, so that its output value is written before required invocations of t_2.

The RTW rate transition mechanism limits the set of syntactically correct programs.

First, the period of each task must be an integer multiple of a base period (e.g., the smallest period). Second, each cycle of the task graph G must contain a dependency resolved with a delay block.
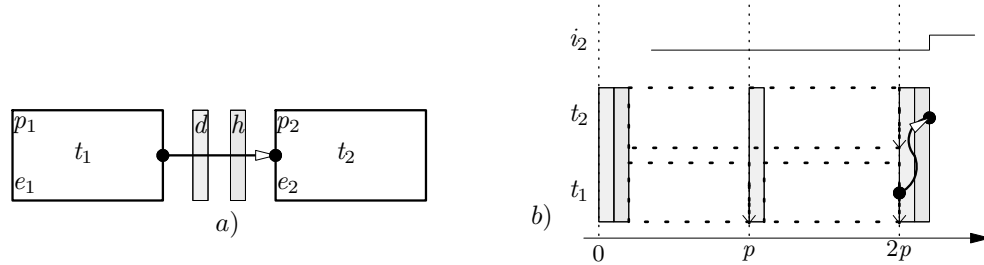
Figure 3.3. LET data transfer - (a) task graph; (b) task and signal timeline for $p_1 = p_2/2 = p$

Logical Execution Time semantics. According to the logical execution time (LET) concurrency model defined in the scope of the Giotto programming language [31], each 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 task reads and writes data are decoupled from the task execution. For the periodic tasks that we consider in this chapter, release and termination time instants of a task are equal to multiples of the task period. The LET model is an abstract programming model that does not prescribe any particular scheduling strategy. This is shown in Fig. 3.3(b) with dashed box throughout a period of a task. Of course, it must be ensured that the generated code satisfies the LET assumption.

For data precedences with LET semantics, a data transfer occurs at release/termination time instants, which abstracts away precedence constraints and makes tasks independently schedulable. Every LET precedence can be modeled as a sequence of a delay d and a hold h block as shown in Fig. 3.3(a). The delay block executes with the period of the data producer t_1 delaying its output until its termination time. The hold block executes with the period of the data consumer t_2 holding its input value during its LET. The LET semantics

63

imposes no limitations on the program, i.e., task periods need not be harmonic and the task graph may be an arbitrary directed graph.

RTW versus LET. We end this section by comparing the two semantics with respect to latency and synchronization requirements. These properties will favor the RTW semantics. In the remaining sections we will compare the two semantics with respect to composability, which makes the LET semantics look better.

The end-to-end latency D of a sequence of n tasks t_j with task precedences (t_{j-1}, t_j) for each $j = 2, ..., n$, is the time between the release of task t_1 and the completion of task t_n. Unlike the RTW semantics, with the LET semantics each fast to slow precedence constraint increases the end-to-end latency by the period of the data producer task. In the worst case all task precedences in the sequence are such, i.e., let $p_j = m_j \cdot p_{j-1}$ for $j = 2, ..., n$ and $m_j \geq 1$. With the RTW semantics, the end-to-end latency D_{RTW} of the sequence is bounded by p_n. With the LET semantics, the latency D_{LET} is bounded by $p_1 + \cdots + p_n$. So, if all tasks have the same period, i.e., if $m_j = 1$, then the worst-case latency with the RTW semantics is n times smaller than the latency with the LET semantics.

Moreover, the latency of the sequence in the RTW case depends on the worst-case execution times of tasks and, as such, can be arbitrarily small. On the other hand, the latency in the LET case depends on the logical execution times of tasks, i.e., on the task periods, no matter how small actual execution times are. This difference is important for the case when a single task graph has a dedicated resource. However, in the case of a partitioned resource, as discussed in [68], the feasibility problem is more relevant than the latency minimization problem. Note that even the latency of the RTW can be reduced either by compromising determinism, or by more complicated synchronization or dataflow models [50, 24].

RTW and LET semantics differ also in their synchronization and memory requirements. Let task t_1 precede task t_2, and let $p_2 = m \cdot p_1$. In p_2 time units of RTW execution, the hold synchronization block is invoked only once. In the same time interval of LET execution, the

hold block is invoked once and the delay block m times. However, both blocks represent data transfers and typically execute in negligible amounts of time.

The memory required for the execution of the program with n tasks and RTW semantics is bounded by n. The same program with LET semantics requires memory twice as large, since the output data must be stored even after a task completes. Moreover, this memory is used all the time during program execution, while in the RTW case memory needed for a task is used only during task execution. A more detailed study of memory requirements for a run-time system with the LET semantics is presented in [42].

3.3   Task Group Abstraction

3.3.1   Independent Task Set Abstraction

We first briefly present results from [68] for schedulability of a set of independent tasks under a periodic resource. A resource can be modeled as a *periodic resource* $R = (T, C)$ if it can guarantee allocations of at least C time units every T time units. The model does not specify how the guaranteed C time units are distributed over a time interval of size T. An *instance* of the periodic resource is any time trace of resource allocations that satisfies the guarantee (T, C). For a given periodic resource $R = (T, C)$, the resource *supply bound function* $\mathtt{sbf}_R : \mathbb{R} \to \mathbb{R}$ maps $\tau \in \mathbb{R}$ into the minimum supply of the resource R over all time intervals of size τ. For details on computing the supply bound function the reader is referred to [68]. As an example, Fig. 3.4 shows the supply bound function $\mathtt{sbf}_R$ for the periodic resource $R = (8, 7)$. Let V be the set of independent, periodic, and preemptive tasks. For a given set of tasks V, the resource *demand bound function* $\mathtt{dbf}_V : \mathbb{R} \to \mathbb{R}$ maps $\tau \in \mathbb{R}$ into the maximum resource demand over all time intervals of size τ. If the scheduling algorithm is earliest deadline first (EDF), we have $\mathtt{dbf}_V(\tau) = \sum_{t_i \in V} \lfloor \tau/p_i \rfloor \cdot e_i$, where $t_i = (p_i, e_i)$. For the rate monotonic scheduling algorithm (RM),

the demand bound function is calculated for each task t_i as a cumulative resource demand of the task over an interval of time τ, i.e., $\mathtt{dbf}_V(\tau, t_i) = e_i + \sum_{t_k \in V(t_i)} \lceil \tau/p_k \rceil \cdot e_k$, where $V(t_i)$ is the set of tasks of higher priority than t_i.
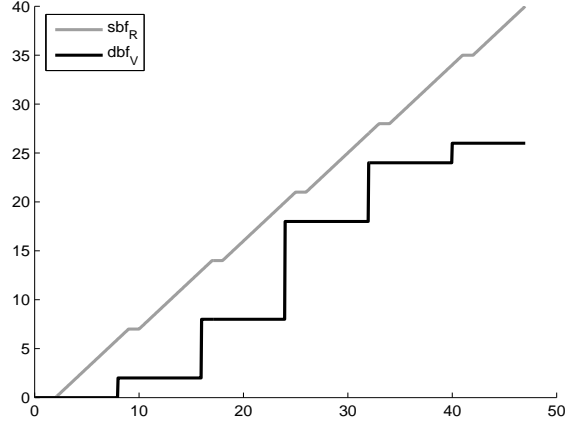
Figure 3.4. Supply and demand bound functions

We say that the scheduling model (V, T, C, A) is *schedulable* if under every instance of allocations of the periodic resource (T, C), there exists a feasible schedule for the task set V with the scheduling algorithm A. Theorems 1 and 2 in [68] give sufficient and necessary conditions for the schedulability of (V, T, C, A) with EDF or RM scheduling algorithms. Let $\mathtt{lcm}_V$ be the least common multiple of the periods of tasks in V. A scheduling model (V, T, C, EDF) is schedulable if and only if for all $0 < \tau \le 2 \cdot \mathtt{lcm}_V$, the maximal resource demand is no greater than the minimum resource supply, i.e., $\mathtt{dbf}_V(\tau) \le \mathtt{sbf}_{(T,C)}(\tau)$. For instance, if V is the set of three tasks $V = \{(24, 8), (8, 2), (16, 4)\}$, then Fig. 3.4 shows the demand bound function $\mathtt{dbf}_V$, and also illustrates that (V, T, C, EDF) is schedulable if $(T, C) = (8, 7)$. A scheduling model (V, T, C, RM) is schedulable if and only if for all tasks $t_i \in V$, there exists $0 < \tau_i \le p_i$ such that $\mathtt{dbf}_V(\tau_i, t_i) \le \mathtt{sbf}_{(T,C)}(\tau_i)$.

In a hierarchical scheduling framework (Fig. 3.5), a separate scheduling problem is solved at each level of the hierarchy. If (V, T, C, A) is schedulable, then the set of independent periodic tasks V under resource (T, C) and algorithm A can be abstracted as a single periodic task (T, C). So, in a hierarchical scheduling framework, the higher-level
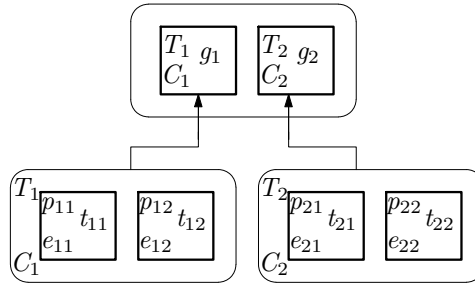
Figure 3.5. Hierarchical scheduling framework

scheduler allocates partitions for the set V as it was a periodic task (T, C). In figures we represent abstractions as rounded boxes, in this case characterized by (T, C) pairs.

3.3.2 Intragroup Task Precedence Abstraction

In this subsection we add precedences to a set of periodic tasks that execute on a single resource. Whereas here we consider a single group of tasks represented by a task graph, later we will discuss multiple, hierarchically structured task groups with precedences between them. We use the term "program" to capture tasks, constraints (timing and precedences), and the task graph dataflow semantics. Formally, a *program* (G, S) consists of a task graph $G = (V, E)$ and semantics $S \in \{\text{RTW}, \text{LET}\}$. For instance, the task graph shown in Fig. 3.6(a) is defined with $G_0 = (\{t_1, t_2, t_3\}, \{(t_1, t_2), (t_2, t_3)\})$.

Definition 1 (Program schedulability) *If under all instances of the periodic resource (T, C), there exists a schedule feasible for task graph G with semantics S the scheduling model (G, T, C, S) is* schedulable.

We assume that a child scheduler, when communicating resource requirements to its parent scheduler, provides not only a single pair (T, C), but a set of pairs, and, in particular, a function with the domain set $\mathbb{Q}$ that maps each period to an execution time requirement (capacity). Such a function enables a tighter hierarchy than a single pair, and also avoids computation of the optimal pair at the child scheduler (e.g., when the switching overhead is

not known). We assume context switching time takes negligible time. This can be avoided by adding the appropriate overhead to task execution time.

Definition 2 (Program abstraction) *A function* $c : \mathbb{Q} \rightarrow \mathbb{R}$ *tightly abstracts* program (G, S) *if* c *maps each period* T *into the smallest capacity* C *such that* (G, T, C, S) *is schedulable.*

If the function c tightly abstracts the program (G, S), then the *abstraction utilization function* $u : \mathbb{Q} \rightarrow \mathbb{R}$ of (G, S) maps each period T into $u(T) = c(T)/T$. In the rest of the chapter, for two functions, f_1 and f_2, with arbitrary domain set A and range set $\mathbb{R}$, and for a relation $\sigma \in \{<, \leq, >, \geq\}$, we write $f_1 \, \sigma \, f_2$, if $f_1(a) \, \sigma \, f_2(a)$ for all $a \in A$. Note that the abstraction utilization function u satisfies $0 \leq u \leq 1$.

The results summarized in Sec. 3.3.1 cannot be used directly on task graphs because precedence constraints with semantics must be taken into account. As explained in Sec. 3.2, the RTW method uses fixed priority scheduling of tasks to maintain the order of task execution. The LET method is not restricted to any scheduling algorithm, and, in principle, its semantics can be implemented with the EDF algorithm where precedences are ignored. Thus, benefits of better schedulability, may in compositional scheduling frameworks be turned into tighter abstraction. However, if the EDF algorithm is not an option [9], some other, simpler scheduling algorithm might also give a tighter abstraction. For example, if the periods of all tasks have a common divisor d, then a simple round robin (RR) technique may be used. For each task (p_i, e_i), let $k_i = p_i/d$. A quantum of e_i/k_i time units is allocated to a task in each round. In this case the demand bound function for the RR scheduling algorithm is $\mathtt{dbf}_V(\tau) = \sum_{t_i \in V} \lfloor \tau/p_i \cdot k_i \rfloor \cdot e_i/k_i$.

Example. Fig. 3.7 shows the functions that tightly abstract the program from Fig. 3.6(a) for different semantics, i.e., scheduling algorithms. As expected, the tightest abstraction is for the EDF scheduling algorithm, i.e., EDF for LET semantics. For this example, the RR algorithm for LET semantics gives the abstraction function that lies between
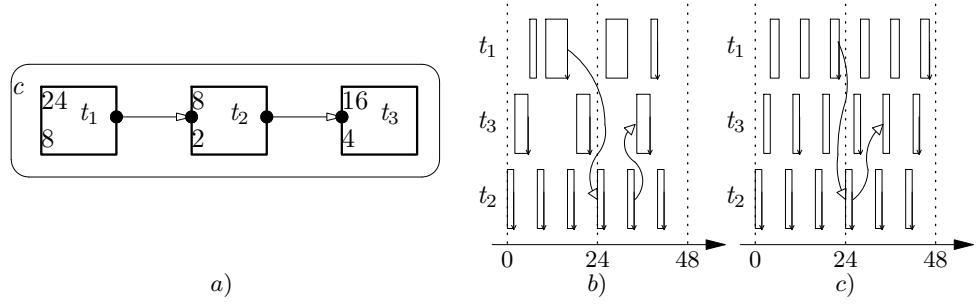
68

Figure 3.6. (a) Task graph; (b) RTW schedule; (c) LET schedule (RR schedule)

EDF and RTW abstraction functions. In particular, for $T = 7.5$, the required capacities are $c_{\text{EDF}}(T) = 6.5$, $c_{\text{RR}}(T) = 6.83$, and $c_{\text{RTW}}(T) = 7.1$.

We performed simulations to evaluate the difference between the two semantics with respect to latency and composability properties. In all simulations we assumed a chain of tasks as the task graph. The size of the chain, i.e., the task workload size, was the parameter of the simulation. The periods of the tasks were randomly assigned from the range [1,20), and the task execution times were chosen such that total workload utilization was in the range [0.3,0.7]. For each pair of successive tasks in a task chain one period was the multiple of the other period, because of the limitation of the RTW dataflow model. For each workload size we ran simulations on 300 task graphs for both RTW and LET semantics (under EDF), and the relative difference, averaged over all task graphs, is shown in Fig. 3.8. The relative difference in end-to-end latency was calculated using the delay between the release of the first task and the worst-case completion time instant of the last task in the task chain. Under a shared periodic resource the delay for both semantics is determined by task periods. The relative difference in composability was calculated as the relative difference in the abstraction functions taken at the smallest period of chain tasks.

We now formalize the observed abstraction gap for a shared single resource, so that it can be compared with the distributed case in Sec. 3.4, and to use it as a base case for our hierarchical scheduling framework discussed in Sec. 3.5.

Lemma 2 *Let* $G = (V, E)$ *be a task graph and* (T, C) *a periodic resource. (1) If*

69
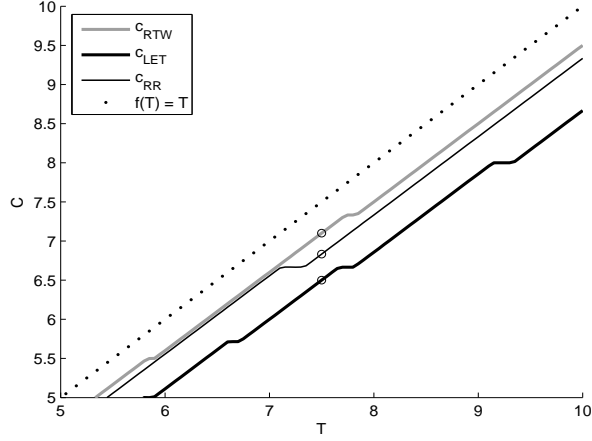
Figure 3.7. Abstraction functions for Fig. 3.6(a)
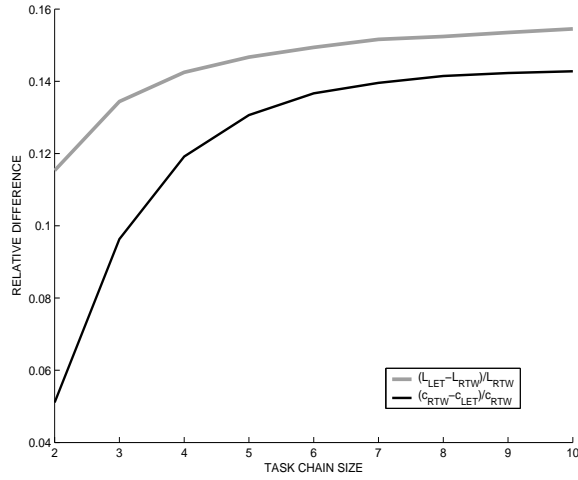
Figure 3.8. Relative difference between RTW and LET semantics w.r.t. latency and composability

(G, T, C, S) is schedulable for $S = \{\text{RTW}, \text{LET}\}$, then (V, T, C, EDF) is schedulable.
(2) If (V, T, C, EDF) is schedulable, then (G, T, C, LET) is schedulable.

Proof. (1) If (G, T, C, S) is schedulable, then schedulability is preserved by removing all precedence constraints to obtain (V, T, C, S). Since for independent tasks EDF is the optimal scheduling algorithm even under a partitioned resource [56], it follows that (V, T, C, EDF) is schedulable. (2) If the independent task set (V, T, C, EDF) is schedulable, then G is schedulable with the LET semantics even with task precedence constraints, since the concurrent task instances are independent. □

Proposition 3 (Tightness) *Let G be a task graph. If there exists a function c_{RTW} that tightly abstracts (G, RTW), then there exists a function c_{LET} that tightly abstracts (G, LET) and $u_{\text{RTW}} - u_{\text{LET}} \geq 0$.*

Proof. Let $G = (V, E)$ and suppose that c_{RTW} tightly abstracts (G, RTW). For all $T \in \mathbb{Q}$, we have that $(G, T, c_{\text{RTW}}(T), \text{RTW})$ is schedulable. From Lemma 2 it follows that $(V, T, c_{\text{RTW}}(T), \text{EDF})$ is schedulable, and consequently, that $(G, T, c_{\text{RTW}}(T), \text{LET})$ is schedulable. So, $c_{\text{LET}}(T)$ is defined and can only be smaller than $c_{\text{RTW}}(T)$. $\square$

Proposition 4 *There exists a task graph G such that in Prop. 3 strict inequality holds, i.e., $u_{\text{RTW}} - u_{\text{LET}} > 0$.*

Similar to the case with a dedicated resource (e.g., [9]), a task graph G with a pair of tasks t_1 and t_2 whose periods p_1 and p_2 are not in a harmonic relation (i.e., there exists no $m \geq 1$ such that $p_2 = m \cdot p_1$ or $p_1 = m \cdot p_2$) can satisfy the proposition. An example is the task graph in Fig. 3.6(a), with c_{RTW} and c_{LET} shown in Fig. 3.7.

3.4 Distributed Task Precedence Abstraction

In this section tasks are distributed over a set of resources. We still consider abstractions of a task graph, i.e. a single task group with task precedence constraints. We again show that the LET approach provides tighter abstraction, and therefore, better composability. In this case the benefits do not only come from the fixed-priority scheduling of the RTW approach. To motivate the problem, consider a teleconferencing application with video and audio streams studied in [12]. The task graph is shown in Fig. 3.9 and the task parameters in Tab. 3.1. The application is distributed over five resources, and the goal is to find its tight abstraction.

Let $\mathcal{R} = \{r_1, ..., r_m\}$ be a set of computational resources on which tasks from the task
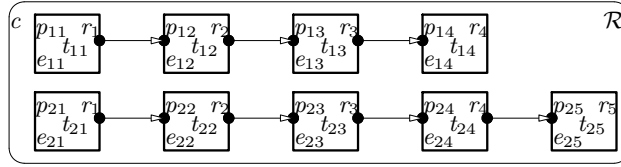
Figure 3.9. Teleconferencing application task graph

Video Tasks	Get frame	IO route	IO route	Display
t_{ij}	t_{11}	t_{12}	t_{13}	t_{14}
Resource r_j	Disk	Sparc	FDDI	PC
Period p_{ij}	2	2	1	720
Exec. time e_{ij}	0.66	1.11	0.44	401.38

Audio Tasks	Get sample	LP filter	IO route	IO route	DA conv.
t_{ij}	t_{21}	t_{22}	t_{23}	t_{24}	t_{25}
Resource r_j	Disk	Sparc	FDDI	PC	DSP
Period p_{ij}	384	384	768	3	3
Exec. time e_{ij}	0.73	18.43	0.49	0.49	0.60

Table 3.1. Example teleconferencing application data

graph G execute, and let m be the size of $\mathcal{R}$. A task is preallocated to a resource and there is no task migration. So, each task is defined with a triple (p, e, r), where $r \in \mathcal{R}$. If a task graph G consists of tasks defined with such triples and S is a semantics, we refer to (G, S) as a *distributed* program. We assume that communication between tasks can either be modeled as a task or it takes a negligible amount of time.

In the multi-resource case each resource has its own independent periodic model, so the period $T \in \mathbb{Q}^m$ and the capacity $C \in \mathbb{R}^m$ are m-tuples. We assume that the scheduler allocates different resources independently, i.e., it only ensures that for each resource the periodic requirement is individually satisfied. First note that the program schedulability definition remains exactly the same as Def. 1 for a single resource. To define tight abstraction in this case, we have to consider minimal capacity with respect to some metric, and here we use a simple multi-resource utilization metric. Given a tuple $T = (T_1, ..., T_m) \in \mathbb{Q}^m$ and a tuple $C = (C_1, ..., C_m) \in \mathbb{R}^m$, let $\mu(T, C) = \sum_{j=1}^{m} \frac{C_j}{T_j}$.

Definition 3 (Multi-resource program abstraction) *A function* $c : \mathbb{Q}^m \to \mathbb{R}^m$ *tightly abstracts* distributed program (G, S) *if* c *maps each period* $T \in \mathbb{Q}^m$ *into the capacity* $C \in \mathbb{R}^m$ *such that*

1. (G, T, C, S) *is schedulable;*

2. *for each* $C' \in \mathbb{R}^m$ *such that* (G, T, C', S) *is also schedulable, we have* $\mu(T, C') \geq \mu(T, C)$.

If the function c tightly abstracts the program (G, S), then the *multi-resource abstraction utilization* function $u : \mathbb{Q}^m \to \mathbb{R}$ of (G, S) is defined by $u(T) = \mu(T, c(T))$. Note that, for a given program (G, S), while there may be several functions c that tightly abstract (G, S), the multi-resource abstraction utilization function u of (G, S) is unique. Also, the function u satisfies $0 \leq u \leq m$.

Proposition 5 (Tightness) *Let* G *be a task graph. If there exists a function* c_{RTW} *that tightly abstracts* (G, RTW)*, then there exists a function* c_{LET} *that tightly abstracts* (G, LET) *and* $u_{\text{RTW}} - u_{\text{LET}} \geq 0$.

Proof. With an argument similar to the proof of Prop. 3, it can be shown that for each $T \in \mathbb{Q}^m$, if $(G, T, c_{\text{RTW}}(T), \text{RTW})$ is schedulable, then $(G, T, c_{\text{RTW}}(T), \text{LET})$ is also schedulable. Consequently, for each $T \in \mathbb{Q}^m$, $u_{\text{LET}}(T)$ is smaller or equal to $\mu(T, c_{\text{RTW}}(T)) = u_{\text{RTW}}(T)$. $\square$

Consider the task graph G in Fig. 3.10(a), distributed over $m = 2$ resources, and notice that the tasks have equal period p. Computing the abstraction utilization function u_{RTW} is more difficult than u_{LET}, because the capacity required for resource r_2 depends on the capacity for resource r_1. The worst case is when resource r_1 is allocated at the end of a period p, and resource r_2 is allocated at the beginning (see Fig. 3.10(b)). If $x \in [e_2, p - e_1]$ and the capacity for r_1 is $e_1 + x$, then the capacity for r_2 has to be at least $p - x + e_2$, so

73

that t_2 completes on time. Based on this task graph, we show in the following proposition that the difference between the abstraction utilization functions for the two semantics can be as large as $m - 1$.
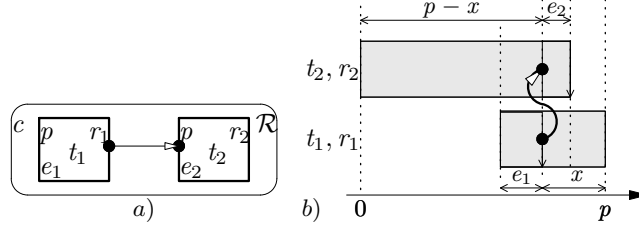

Figure 3.10. Example for $m = 2$ resources: (a) task graph; (b) resource partition

Proposition 6 *For all $\epsilon > 0$, there exists a task graph G and a period $T \in \mathbb{Q}^m$ such that $u_{\mathrm{RTW}}(T) - u_{\mathrm{LET}}(T)$ is within ϵ of $m - 1$.*

Proof. Let $G = (V, E)$ be a chain of m tasks with the same period p assigned to m different resources, i.e., a generalization of the task graph from Fig. 3.10(a). Let $\overline{V}$ be the task set obtained from V by modifying, for each $j = 1, ..., m$, task $(p, e_j, r_j) \in V$ into task $(p, \overline{e}_j, r_j) \in \overline{V}$, where $\overline{e}_j = e_j + x_j$ if $j = 1$, and $\overline{e}_j = p - x_{j-1} + e_j + x_j$ if $1 < j < m$, and $\overline{e}_j = p - x_{j-1} + e_j$ if $j = m$ (see Fig. 3.11). Let $T = (p, ..., p)$. Then (G, T, C, RTW) is schedulable if and only if $(\overline{V}, T, C, \mathrm{EDF})$ is schedulable and $x_j \in [e_{j+1}, p - e_j]$ for each $j = 1, ..., m - 1$. If e_1 is close to p, and e_j is close to 0 for each $j = 2, ..., m$, then $\overline{e}_j$ is close to p for each $j = 1, ..., m$. Consequently, $u_{\mathrm{RTW}}(T)$ can be arbitrarily close to m. We also have that (G, T, C, LET) is schedulable if and only if (V, T, C, EDF) is schedulable. Since e_1 can be arbitrarily close to p, and e_j arbitrarily close to 0 for each $j = 2, ..., m$, $u_{\mathrm{LET}}(T)$ can be arbitrarily close to 1. $\square$

Remark 1 If we assume that for all tasks (p_j, e_j, r_j) of the task graph G in Prop. 6 the execution time e_j is less than p/m, then we can consider a pipelined execution, i.e., the task graph G' which differs from G in that each task period p_j (for $j = 1, ..., m$) is equal to p/m. Such a task graph with LET semantics has the same latency as the original task graph

74
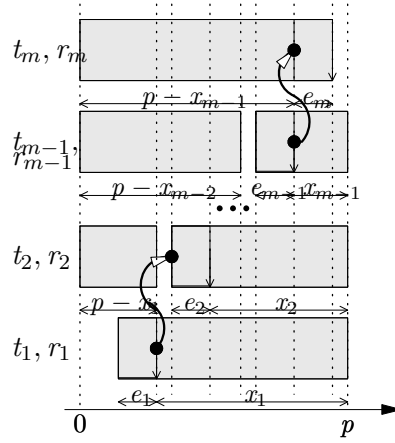
Figure 3.11. Resource partition for Prop. 6

with RTW semantics. Moreover, abstraction is still tighter for LET: $u_{\text{RTW}}(T) - u'_{\text{LET}}(T)$ can be arbitrarily close to $((m-1)p + \sum e_j - m\sum e_j)/p$, and thus can be greater than 0, since $p - \sum e_j \geq 0$. $\square$

Remark 2 Although Prop. 6 holds for $m > 1$, its form for $m = 1$ is similar to Prop.4. However, Prop. 4 holds globally with strict inequality, which is not the case for Prop. 6 since both u_{RTW} and u_{LET} approach m as the argument T becomes large. $\square$

We next argue that abstraction and scheduling for tasks with precedences and multiple shared resources are simpler with the LET semantics. Let $G = (V, E)$ be a task graph, and let $V_j \subseteq V$ (for $j = 1, ..., m$) be the set of all tasks allocated to the resource r_j. Let the task graph $G_j = (V_j, E \cap V_j^2)$ be the r_j-*task graph*, and let $c_j : \mathbb{Q} \to \mathbb{R}$ be a function that tightly abstracts (G_j, S).

Proposition 7 (Abstraction) *(1) The function c_{LET} that maps each period $T = (T_1, ..., T_m)$ to $c_{\text{LET}}(T) = (c_1(T_1), ..., c_m(T_m))$, tightly abstracts (G, LET). (2) There exist two task graphs G and G' with the same r_j-task graph for each $j = 1, ..., m$, such that the functions that tightly abstract (G, RTW) and (G', RTW) are not equal.*

75

Consider, for instance, Fig. 3.12. The function c_{LET} that tightly abstracts the teleconferencing program from Fig. 3.9 is defined by the $m = 5$ single-variable functions $c_{j,\mathrm{LET}}$ computed independently for each resource r_j as discussed in Sec. 3.3.2. According to Prop. 7(2), knowing all functions $c_{j,\mathrm{RTW}}$ is not sufficient to construct the function c_{RTW}. An example for the task graph G in Prop. 7(2) is the graph from Fig. 3.10(a), and for the task graph G', the same graph without the edge between t_1 and t_2.
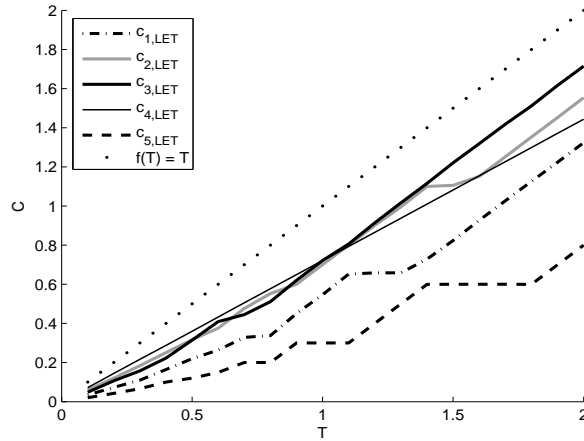
Figure 3.12. LET abstraction functions for Fig. 3.9

Note that the complexity of a scheduling problem on distributed task graphs, asking whether an end-to-end latency requirement is satisfied, depends on the choice of semantics. For the RTW semantics the problem is similar to the job-shop scheduling problem, and is NP-hard even in simple variants [8]. If the LET semantics is acceptable, then the scheduling problem can be decomposed into a set of simple single-resource scheduling problems. This simplicity is favorable for program admission tests.

3.5   Hierarchical Intergroup Abstraction

In this section we allow for the existence of precedences between different task groups, i.e., different task graphs. We discuss tightness and the construction of abstractions for such task graphs. To simplify the presentation we restrict the discussion to the single-resource

case. We define a hierarchical scheduling framework that takes into account precedence constraints. In this framework the separation property between parent and children levels is not satisfied, i.e., the parent scheduler has to know the details of the entire hierarchy below it. However, we formally show that the separation property holds with LET, albeit not with RTW semantics.

We use the term "program" for the first level of a hierarchy, and "hierarchical program" for higher levels. So, programs are composed into hierarchical programs, and these are composed into higher-level hierarchical programs. Let $G = (V, E)$ be a *flat* task graph, a graph defined with the set V of all tasks and the set E of all precedences. We first inductively define a *task hierarchy*. (1) Every set of tasks $\mathcal{H} \subseteq V$ is a task hierarchy on G. In this case, the set of vertices $\mathcal{V}$ of the task hierarchy $\mathcal{H}$ is equal to $\mathcal{H}$. (2) If $\mathcal{H}_j$ is a task hierarchy on G with a set of vertices $\mathcal{V}_j$ for each $j = 1, ..., k$, and all sets of vertices are mutually disjoint, then the collection $\mathcal{H} = \{\mathcal{H}_1, ..., \mathcal{H}_k\}$ is a task hierarchy on G. In this case, the set of vertices $\mathcal{V}$ of a task hierarchy $\mathcal{H}$ is the union of the sets of vertices of its elements, i.e., $\mathcal{V} = \cup_{j=1}^{k} \mathcal{V}_j$.

A *hierarchical task graph* $\mathcal{G} = (\mathcal{H}, \mathcal{E})$ on the flat task graph G consists of a task hierarchy $\mathcal{H}$ on G with a set of vertices $\mathcal{V}$ and the set of precedences $\mathcal{E} = E \cap \mathcal{V}^2$. A *hierarchical program* $(\mathcal{G}, S)$ on the flat task graph G consists of a hierarchical task graph $\mathcal{G}$ on G and a semantics $S \in \{\text{RTW}, \text{LET}\}$.
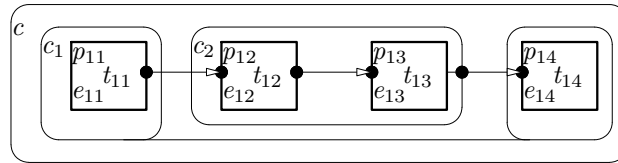

Figure 3.13. Video stream hierarchical abstraction

Example. Assume that the video stream from the teleconferencing application (Sec. 3.4) executes entirely on the same resource, and that all execution times are scaled down 4 times. Fig. 3.13 gives an example of task groups with intergroup precedences.

There are $k = 2$ task groups at the leaf level of the hierarchy, and note that there is a precedence between the groups in each direction. The flat task graph $G = (V, E)$ is defined with $V = \{t_{11}, t_{12}, t_{13}, t_{14}\}$ and $E = \{(t_{11}, t_{12}), (t_{12}, t_{13}), (t_{13}, t_{14})\}$. There are three task hierarchies $\mathcal{H}_1 = \{t_{11}, t_{14}\}$, $\mathcal{H}_2 = \{t_{12}, t_{13}\}$, and $\mathcal{H} = \{\mathcal{H}_1, \mathcal{H}_2\} = \{\{t_{11}, t_{14}\}, \{t_{12}, t_{13}\}\}$. The corresponding sets of vertices are $\mathcal{V}_1 = \{t_{11}, t_{14}\}$, $\mathcal{V}_2 = \{t_{12}, t_{13}\}$, $\mathcal{V} = V$, and the sets of precedences are $\mathcal{E}_1 = \emptyset$, $\mathcal{E}_2 = \{(t_{12}, t_{13})\}$, and $\mathcal{E} = E$. Finally, the three hierarchical task graphs defined by the hierarchy are $\mathcal{G}_1 = (\mathcal{H}_1, \mathcal{E}_1)$, $\mathcal{G}_2 = (\mathcal{H}_2, \mathcal{E}_2)$, and $\mathcal{G} = (\mathcal{H}, \mathcal{E})$. $\square$

Note that if in a hierarchical task graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the task hierarchy $\mathcal{H}$ is equal to a subset of tasks V, then $\mathcal{G}$ reduces to a subgraph of G, and the hierarchical program $(\mathcal{G}, S)$ reduces to a program as defined in 3.3.2. An example is the hierarchical task graph $\mathcal{G}_2$. For such a hierarchical program, Def. 1 defines schedulability and Def. 2 defines abstraction functions. We use these definitions as a base case for Def. 4 and Def. 5, which respectively define the same properties for higher-level hierarchical programs. The following convention holds for all remaining propositions in this section.

Convention. Let $(\mathcal{G}, S)$ be a hierarchical program on a flat task graph G with $\mathcal{G} = (\mathcal{H}, \mathcal{E})$ and $\mathcal{H} = \{\mathcal{H}_1, ..., \mathcal{H}_k\}$. Let $\mathcal{V}$ be the set of vertices of $\mathcal{H}$, and let $\mathcal{E}$ (resp. $\mathcal{E}_j$) be the set of precedences of $\mathcal{H}$ (resp. $\mathcal{H}_j$). Let $\mathcal{G}_j = (\mathcal{H}_j, \mathcal{E}_j)$ for $j = 1, ..., k$; we refer to $\{\mathcal{G}_1, ..., \mathcal{G}_k\}$ as the set of *component graphs* of $\mathcal{G}$. Let $\mathcal{C} = (c_1, ..., c_k)$ be a tuple of functions such that for each $j = 1, ..., k$ the function $c_j : \mathbb{Q} \to \mathbb{R}$ tightly abstracts the hierarchical program $(\mathcal{G}_j, S)$. Given a tuple $P = (P_1, ..., P_k) \in \mathbb{Q}^k$ and a tuple $\mathcal{C} = (c_1, ..., c_k)$ of functions $c_j : \mathbb{Q} \to \mathbb{R}$, let $V_{P,\mathcal{C}}$ be the set of independent tasks defined with $V_{P,\mathcal{C}} = \{(P_j, c_j(P_j)) \mid j = 1, ..., k\}$. $\square$

In a hierarchical scheduling framework, a separate scheduling problem is solved at each level of the hierarchy. We assume that the scheduler at the level of a hierarchical

program $((\mathcal{H}, \mathcal{E}), S)$, knows only about the precedences in $\mathcal{E}$, but not about the entire set E. The scheduler has to determine a schedule that satisfies both the requirements of the components, i.e., the requirements of the task set $V_{P,\mathcal{C}}$ for some tuple P, and all precedences introduced up to this level, i.e., the requirements of the program $((\mathcal{V}, \mathcal{E}), S)$. In the example from Fig. 3.13 there are two levels of scheduling. Let c_1 and c_2 be the functions that respectively abstract programs $\mathcal{G}_1$ and $\mathcal{G}_2$. The higher-level scheduler has to satisfy the requirements of the entire program (G, S), but also the requirements of the task set $\{(P_1, c_1(P_1)), (P_2, c_2(P_2))\}$ for some rationals P_1 and P_1. Since components do not specify resource requirements as a single (T, C) pair, the following definition contains an additional existential quantifier.

Definition 4 (Hierarchical program schedulability) *If under all instances of a given periodic resource (T, C), there exist a tuple $P \in \mathbb{Q}^k$ and a schedule feasible both for the set of independent tasks $V_{P,\mathcal{C}}$ and the program $((\mathcal{V}, \mathcal{E}), S)$, we say that $(\mathcal{G}, T, C, S)$ is schedulable.*

Definition 5 (Hierarchical program abstraction) *A function $c : \mathbb{Q} \to \mathbb{R}$ tightly abstracts a hierarchical program $(\mathcal{G}, S)$ if c maps each period T into the smallest capacity C such that $(\mathcal{G}, T, C, S)$ is schedulable.*

The following proposition shows that the hierarchical scheduling framework is *compositional* for the LET semantics, but not for the RTW semantics, because in the LET case, the abstraction function (i.e., timing properties) for the composition can be established from independent abstraction functions for the components. In the case of LET semantics, we show how to construct a function that tightly abstracts a hierarchical program from the tuple $\mathcal{C}$ of functions that tightly abstract hierarchical programs at the next lower level of the hierarchy. Given $T \in \mathbb{Q}$, let

$$c_{min}(T) = \min_{P \in \mathbb{Q}^k} \{c'(T) \mid c' : \mathbb{Q} \to \mathbb{R} \text{ tightly abstracts } V_{P,\mathcal{C}}\}.$$

Consider the example from Fig. 3.13, and Fig. 3.14. The two functions drawn with dash lines, $c_{1,\text{LET}}$ and $c_{2,\text{LET}}$, are tight abstractions of the leaf-level hierarchical programs $(\mathcal{G}_1, \text{LET})$ and $(\mathcal{G}_2, \text{LET})$, respectively. These are computed as explained in Sec. 3.3.2. The third function, c_{LET}, which tightly abstracts $(\mathcal{G}, \text{LET})$, is the function c_{min} from the above expression computed using $c_{1,\text{LET}}$ and $c_{2,\text{LET}}$. Beside this, the following proposition also shows that, in general, knowing the tuple of functions $c_{j,\text{RTW}}$ is not sufficient to construct the function c_{RTW} that tightly abstracts $(\mathcal{G}, \text{RTW})$.
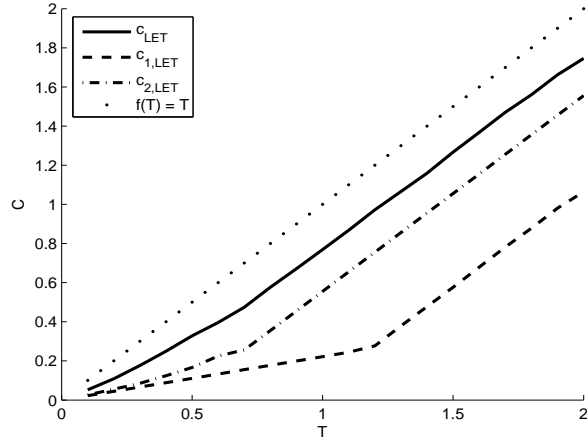
Figure 3.14. LET abstraction functions for Fig. 3.13

Proposition 8 (Abstraction) *(1) The function c_{LET} that maps each period T to $c_{\text{LET}}(T) = c_{min}(T)$, tightly abstracts $(\mathcal{G}, \text{LET})$. (2) There exist two hierarchical task graphs $\mathcal{G}$ and $\mathcal{G}'$ with the same set of component graphs, such that the functions that tightly abstract $(\mathcal{G}, \text{RTW})$ and $(\mathcal{G}', \text{RTW})$ are not equal.*

Proof. (1) Given $T \in \mathbb{Q}^k$, we first prove that $(\mathcal{G}, T, c_{min}(T), \text{LET})$ is schedulable. From the definition of the function c_{min}, let $P \in \mathbb{Q}^k$ and a function c' be such that c' tightly abstracts $V_{P,\mathcal{C}}$ and $c'(T) = c_{min}(T)$. From Def. 2, it follows that $V_{P,\mathcal{C}}$ is schedulable under each instance of the periodic resource $(T, c'(T)) = (T, c_{min}(T))$. From Lemma 3 it follows that $(\mathcal{G}, T, c_{min}(T), \text{LET})$ is schedulable. Assume that there exists $C < c_{min}(T)$ such that $(\mathcal{G}, T, C, \text{LET})$ is schedulable. By Def. 4, there exists $P \in \mathbb{Q}^k$

such that $(V_{P,\mathcal{C}}, T, C, \mathrm{LET})$ is schedulable. Let c' be the function that tightly abstracts $V_{P,\mathcal{C}}$. From Def. 2, we have $c'(T) \leq C$ and from definition of the function c_{min}, we have $c_{min}(T) \leq c'(T)$. This is a contradiction.

(2) Consider the example shown in Fig. 3.16(a). Let $\mathcal{G}_0 = (\{t_1, t_2, t_3\}, \{(t_1, t_2), (t_2, t_3)\})$ and $\mathcal{G} = (\{\{t_1, t_2\}, \{t_3\}\}, \{(t_1, t_2), (t_2, t_3)\})$. Let $c_{0,S}$ (resp., c_S) be the function that tightly abstracts hierarchical program $(\mathcal{G}_0, S)$ (resp., $(\mathcal{G}, S)$). Fig. 3.17(b) shows that $c_{0,\mathrm{LET}} < c_{\mathrm{LET}} < c_{0,\mathrm{RTW}}$. On the other hand, $c_{0,\mathrm{RTW}} \leq c_{\mathrm{RTW}}$, since $\mathcal{G}_0$ is a leaf-level task graph containing the same tasks as $\mathcal{G}$. Let $\mathcal{G}' = (\{\{t_1, t_2\}, \{t_3\}\}, \{(t_1, t_2), (t_3, t_2)\})$ be the hierarchical task graph equal to $\mathcal{G}$, except that the edge (t_2, t_3) is of opposite direction. The function that tightly abstracts $(\mathcal{G}', \mathrm{RTW})$ is equal to $c_{\mathrm{LET}} < c_{\mathrm{RTW}}$, since both edges in the hierarchical task graph $\mathcal{G}'$ introduce delays. This means that knowing only the functions that tightly abstract the lower-level hierarchical programs, and not the direction of all edges, is not sufficient to compute c_{RTW}. □

The next proposition shows that the hierarchical scheduling framework exhibits *separation* for the LET semantics, because for scheduling only component abstractions, and not component internals, are sufficient. For the RTW semantics, in general, knowing the tuple of functions $c_{j,\mathrm{RTW}}$ is not sufficient to construct a feasible schedule. We explain how to construct the composite schedule in the case of LET semantics. Given $T \in \mathbb{Q}$, let $P_{min}(T)$ be $P \in \mathbb{Q}^k$ such that for a function c' that tightly abstracts $V_{P,\mathcal{C}}$ we have $c'(T) = c_{min}(T)$, i.e., let it be equal to the P that minimizes the expression defining the function c_{min}.

Proposition 9 (Scheduling) *Let $(\mathcal{G}, S)$ be a hierarchical program and (T, C) a periodic resource such that $(\mathcal{G}, T, C, S)$ is schedulable. (1) If $S = \mathrm{LET}$, then the EDF algorithm for $V_{P_{min}(T),\mathcal{C}}$ constructs a feasible schedule for $(\mathcal{G}, T, C, \mathrm{LET})$. (2) If $S = \mathrm{RTW}$, then there exists a hierarchical task graph $\mathcal{G}'$ with the same set of component graphs as $\mathcal{G}$, such that no schedule is feasible both for $(\mathcal{G}, T, C, \mathrm{RTW})$ and $(\mathcal{G}', T, C, \mathrm{RTW})$.*

81

Proof. (1) The schedule generated with $P = P_{min}(T)$ is feasible even if the provided capacity C is $c_{min}(T)$. From Lemma 3, it follows that the schedule is feasible for $(\mathcal{G}, T, C, \text{LET})$. (2) Consider the hierarchical task graph shown in Fig. 3.16(b). Let $x \in [-p/4, p/4]$ be a variable parameter of the task execution requirements not known to the scheduler at the higher-level. Let, for instance, the hierarchical graphs $\mathcal{G}$ and $\mathcal{G}'$ from the statement of Prop. 9 be as in Fig. 3.16(b) with $x = -p/4$ and $x = p/4$ respectively. With RTW semantics, depending on x, one or the other data precedence becomes more critical. The total requirement per hierarchical program is independent of the value for x, i.e., the functions that tightly abstract the hierarchical programs do not depend on x. However, to construct a schedule that satisfies all data precedences, the scheduler would have to know the value of x. □

Consider again the example from Fig. 3.13. According to Fig. 3.14, if $T = 1$ then the required capacity for the hierarchical program $(\mathcal{G}, \text{LET})$ is $c_{\text{LET}}(T) = 0.776$. Fig. 3.15(a) shows an instance of the resource model $R = (1, 0.776)$ for the first three periods T, in which the resource is respectively allocated at the beginning, at the end, and in the middle of the period T. The rest of Fig. 3.15 shows the hierarchical generation of the schedule. If $T = 1$, then $P_{min}(T) = (1, 0.7)$, and from Fig. 3.14 we have $c_{1,\text{LET}}(1) = 0.22$ and $c_{2,\text{LET}}(0.7) = 0.25$. Fig. 3.15(b) shows the higher-level schedule: the EDF schedule for the two periodic tasks $(1, 0.22)$ and $(0.7, 0.25)$ in the partitions from Fig. 3.15(a), i.e., it shows the schedule for $(\mathcal{G}_1, \text{LET})$ and $(\mathcal{G}_2, \text{LET})$. Fig. 3.15(c) shows the lower-level schedule for $\mathcal{G}_1$, i.e., it shows the schedule for tasks $t_{11} = (2, 0.16)$ and $t_{14} = (720, 100.34)$. Finally, Fig. 3.15(d) shows the same for tasks $t_{12} = (2, 0.27)$ and $t_{13} = (1, 0.11)$.

The following two statements generalize Prop. 3 and Prop. 4.

Proposition 10 (Tightness) *If there exists a function c_{RTW} that tightly abstracts $(\mathcal{G}, \text{RTW})$, then there exists a function c_{LET} that tightly abstracts $(\mathcal{G}, \text{LET})$ and $u_{\text{RTW}} \geq u_{\text{LET}}$.*

Prop. 10 follows from Lemma 3 similar to the proof of Prop. 3.
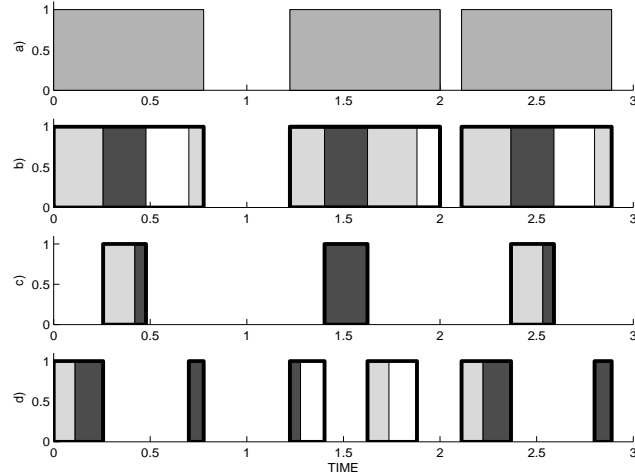
Figure 3.15. (a) Instance of $R = (1, 0.776)$; (b) L: $\mathcal{G}_2$, D: $\mathcal{G}_1$; (c) L: t_{11}, D: t_{14}; (d) L: t_{13}, D: t_{12} (L=light, D=dark)

Proposition 11 *There exists a hierarchical task graph $\mathcal{G}$ such that in Prop. 10 strict inequality holds, i.e., $u_{\text{RTW}} - u_{\text{LET}} > 0$.*

Prop. 11 follows from the proof of Prop. 8(2). An example is the hierarchical task graph from Fig. 3.16(a), whose c_{LET} function is shown in Fig. 3.17(a), (b).
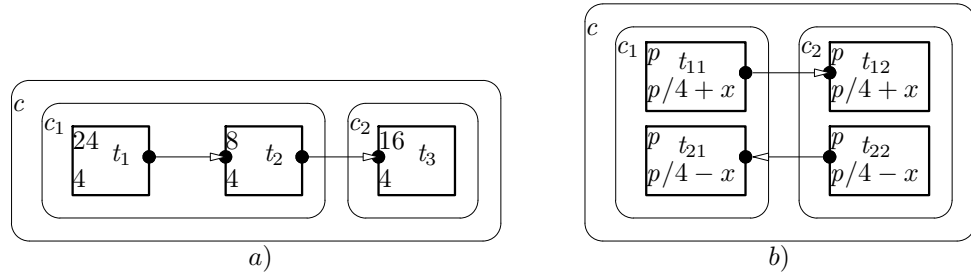
Figure 3.16. Intergroup precedence abstraction examples for (a) Prop. 8(2); (b) Prop. 9(2)

Lemma 3 generalizes Lemma 2 to the hierarchical framework.

Lemma 3 *Let (T, C) be a periodic resource. (1) If $(\mathcal{G}, T, C, S)$ is schedulable for $S = \{\text{RTW}, \text{LET}\}$, then there exists a tuple $P \in \mathbb{Q}^k$ such that $(V_{P,\mathcal{C}}, T, C, \text{EDF})$ is schedulable. (2) If there exists a tuple $P \in \mathbb{Q}^k$ such that $(V_{P,\mathcal{C}}, T, C, \text{EDF})$ is schedulable, then $(\mathcal{G}, T, C, \text{LET})$ is schedulable.*

Proof. (1) Follows directly from Def. 4. (2) We prove that the schedule for the task set $V_{P,\mathcal{C}}$ is also a feasible schedule for $((\mathcal{V}, \mathcal{E}), \text{LET})$, which by Def. 4 makes $(\mathcal{G}, T, C, \text{LET})$
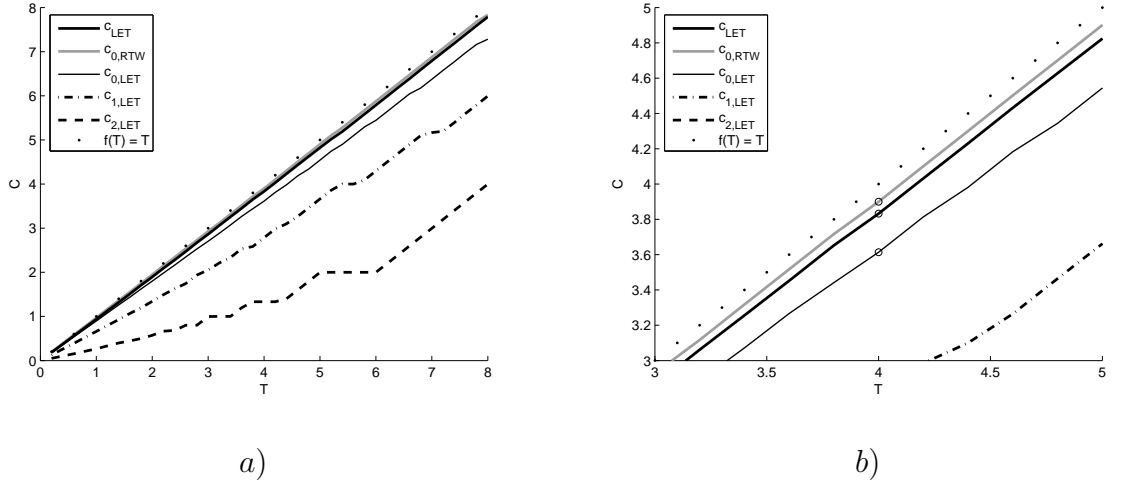
Figure 3.17. (a) Component abstraction function for the hierarchical program in Fig.3.16(a); (b) Detailed view

schedulable. A schedule for $(\mathcal{V}, \mathcal{E})$ is feasible with LET semantics if all tasks in $\mathcal{V}$ individually satisfy their timing requirements. Note that each task in $\mathcal{V}$ is an element of the set from the task hierarchy. We use induction on the structure of $\mathcal{G}$. At each level of the hierarchy a feasible schedule for $V_{P,\mathcal{C}}$ makes $(\mathcal{G}_j, P_j, c_j(P_j), \text{LET})$ schedulable for each $j = 1, ..., k$. At the leaf level this condition guarantees schedulability of each task group of the hierarchy. $\square$

3.6 Conclusion

We addressed the problem of abstracting interacting periodic real-time components in the scope of hierarchical scheduling. We compared two semantics, RTW and LET, for task precedences, within and between components, on single or distributed resources. The results of the last two sections can be generalized for applications with both intergroup and distributed task precedences. We recognized the latency vs. composability trade-off between the two semantics. We showed that advantageous properties of a hierarchical

framework, separation and compositionality, can be achieved with the LET semantics. A natural way to extend the framework would be combining the two semantics, i.e., defining a framework in which a particular semantics would be specified for each precedence constraint. The LET semantics would typically be selected for less time-critical paths in the application task graph. A potential solution for both low latency and tight abstractions is a more complicated scheduling interface. Another approach may use a different resource model. There are related efforts in this direction [58, 69]; how they can be used in the the context of interacting components is a topic for future research.

Chapter 4

Interface Formalism for Real-time Components

4.1 Introduction

As discussed in previous chapters, the increasing complexity of real-time and embedded systems necessitates advanced design and maintenance procedures for the assurance of timing requirements. Automatic tools are highly desired for such an error-prone and tedious process. Since in design exploration the timing performance estimation is performed for a large number of design alternatives, the tools are required to be efficient. The systems are typically put together from several interacting software components that are often provided by different providers. In addition to that, common modification of system requirements demands flexible procedures. In this chapter we develop effcient and flexible interface-based framework for real-time component integration.

Component-based design simplifies the design process since system decomposition provides a solution to the original large problem by solving several smaller problems. Another advantage of such an approach lies in the fact that component performance analysis detects design errors before the components are implemented and composed. As we

noted before, the previous research in component-based real-time systems concentrated on partitioning and scheduling frameworks that make both the implementation and temporal behavior of a component independent of the presence of other components in the system [16, 61]. More recent works present methods that abstract internal complexity of a real-time component into a component *interface* that is subsequently used for the rest of the design [68, 51, 2]. This research considers the *periodic resource* model (T, C), a resource abstraction under which a component is guaranteed to get C units of the resource every T units of time. The methods show how to abstract a set of independent periodic tasks with EDF or RM scheduling algorithms into a single periodic task. Later work [69] shows how to abstract a set of independent periodic tasks into the *bounded-delay* interface. The bounded-delay resource model (c, δ), studied in [59, 56, 57], guarantees fraction c of the resource with at most δ time units of delay.

In this chapter we start with a different task group model and use a method, similar to the one presented in [69], for abstracting such a group into a bounded-delay interface. The task model consists of a set of aperiodic tasks each specified with an arrival rate function and a relative deadline. The arrival rate function bounds the number of task requests in a given interval of time. To abstract such a task group we consider only the bounded-delay resource model with the EDF algorithm, although other mentioned results can be applied in a different setting. We then consider such a task group as a part, i.e., a component, of a larger real-time system specified with a set of task sequences that define task precedence constraints. The objective of the chapter is to study automatic, efficient and flexible component-based design of such a system.

To address the problem we apply concepts from interface theories [13, 14]. In this formalism an interface of a component specifies what the component expects (assumes) from its environment and what it provides back (guarantees) to it. This constraint should be sufficient to check if two interfaces are *compatible*, i.e., if the underlying components work properly when composed together. In the real-time context 'proper' means satisfying

timing requirements, e.g. end-to-end latency. Since the system specification includes dependencies between the tasks from different components, the interface cannot just contain resource constraints as in previous works, but also dataflow propagation constraints. Therefore, beside the resource model assumption, an interface also specifies the task sequence arrival rate assumption and the latency guarantee.

We define an *interface algebra* for real-time interfaces, a formal algebra that enables tool support for our formalism [14]. Beside the compatibility relation, the algebra consists of two operations and a relation. The interface *composition* operation collects the interfaces and sums resource requirements of the underlying components. The other operation, the interface *connection* operation relates components by interconnection. The refinement relation aims at formalizing the relation between abstract and concrete versions of the same component. A more refined version of a component may make a weaker input assumptions and stronger output guarantees than a more abstract description. Therefore, in a design we can always substitute a refined version for an abstract one.

One of the beneficial properties of the interface formalism is *incremental design*. According to this property the composition of interfaces can be performed in any order, i.e., it is associative. Beside having more flexible framework, this also means being able to check compatibility and compute composition of the two interfaces without specifying interfaces of other components. Note that task group abstraction procedures are generally not associative.

Additionally, in component-based design, one wants to refine an interface towards an implementation, independently of the design of other components. If all implementations satisfy their respective interfaces, the components will properly work together. The *independent refinement* property of the formalism states that in order to refine a given composition of two interfaces, it suffices to independently refine each interface and to compose

the obtained refinements. This property enables the system correctness to be established during interface design, without global check after components are implemented.

Our formalism supports automatic interface compatibility and interface refinement checking. The interfaces are stateless, i.e., represented by predicates, and, thus, checking of the two properties is efficient. In this chapter we are concerned with defining the algebra and showing how it can be used on a few examples of real-time applications of moderate complexity.

Beside the theoretical work in compositional real-time scheduling frameworks, the increased interest in real-time component-based systems has recently resulted in first implementations. In [77] the interface of a software component is extended to include real-time assumptions and guarantees of the component. We use similar functional and temporal specifications, except that we allow for multiple levels of service of a component, i.e., for the component performance polymorphism. However, since the goal of [77] is reusability across different platforms, the resource consumption specification is not part of the component interface. So, the resource utilization computation is separated from the application design which assumes virtual resources. Beside, no abstraction of resource requirements is studied. Instead as units of reuse, we consider components more as units of design.

The approach taken in this chapter is most similar to the recent work [75]. That work is the first research effort that formally combines the network calculus and interface design theories in the real-time context. It is not limited to a particular task set characterization or to a particular resource model. Opposite to the traditional real-time approaches, it allows for the composition of software process components before the hardware resource components are specified. In their work each component represents a task. There is no abstraction of task groups into components, and no discussion of interface refinement, which is one of the goals of our work. Also, the task model in [75] assumes independent tasks, so interface compatibility checking does not have to take into account dataflow constraints. Finally,

they assume preemptive fixed-priority scheduling. However, although each component is specified with a certain priority, the interface composition does not have to be performed in a certain order.

In interface theory research [13, 15] the component interaction is specified using richer interfaces. The temporal input/output behavior of a component is typically captured by an automaton. Therefore, the automaton of the composite interface is constructed by pruning all violating states from the product of the component automata. Such stateful approach is a more general way to address multiple levels of component performance. However, in this chapter we keep the interface formalism simple in order to focus more on real-time issues.

Outline of the Chapter. Sec. 4.2.1 informally introduces a real-time component studied in the chapter with its functional, temporal and resource parts. The temporal portion of the component interface consists of a request arrival function and delay. The same section introduces the resource portion of the interface in the form of the bounded-delay resource model. How to obtain resource partition parameters for a group of tasks is presented in Sec. 4.2.2. We introduce interfaces in Sec. 4.3.1, and formally define an interface algebra Sec. 4.3.2. Discussion of how interfaces can be used for efficient and automatic component-based design and verification is left for Sec. 4.4. In particular, incremental design is discussed in Sec. 4.4.1 and independent refinement in Sec. 4.4.2. Sec. 4.5 considers the corresponding interface algebra for general task graphs that may contain cycles. We work with simpler event models and under certain conditions we are able to prove the associativity and independent refinement even in this case.

4.2 Real-Time Components

4.2.1 Resource Model

Functional model. Let a *task sequence* $\pi = t_1 t_2 \ldots t_k$ be a sequence of tasks with a precedence constraint between t_j and t_{j+1}, $(j = 1, \ldots, k-1)$. Although our arguments can be generalized for trees of tasks, we keep the task sequence model for simplicity reasons. We consider components as units for implementation, reuse and composition of task sequences. The functional description of a component consists of a set of task sequences. Two task sequences from the same component can contain the same task. Figure 4.1 shows an example of a component with two task sequences $t_1 t_3$ and $t_2 t_3$. For the purposes of the chapter it is not important whether task inputs/outputs are data processed by tasks or only requests for task execution. The M and D blocks represent no tasks. Only in figures they technically denote multiplexing and demultiplexing, i.e., task sequences $t_1 t_3$ and $t_2 t_3$ are interleaved and independent.
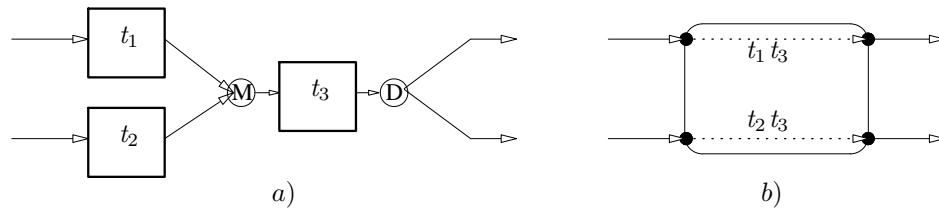
Figure 4.1. (a) Task graph; (b) Component

Arrival-delay temporal model. The temporal interface of a component is similar to the interface of a component in [77], and consists of an arrival function and a maximum delay for each sequence of the component. In fact, it consists of several pairs of arrival function and delay, one for each level of service of the component, as formally defined in Sec. 4.3.

An *arrival function* a of a task sequence is a function that bounds the number of the

invocations of the task sequence: for a time interval of length τ the number of invocations is bounded by $a(\tau)$. In this chapter we concentrate on the *bursty arrival* pattern which is defined with the function $a(\tau) = \sigma + \rho \cdot \tau$ for some $\sigma, \rho \in \mathbb{R}_{\geq 0}$. Both periodic and sporadic invocation patterns can be modeled by the bursty arrival functions. The expression for a gives the upper bound on the number of invocations. When required we consider integer upper bound $\lfloor a(\tau) \rfloor$.

A number $d \in \mathbb{R}$ is a *delay* of a task sequence if all tasks of the sequence must be completed within d units of time, i.e., a sequence output must be generated at most d time units after the occurrence of a sequence input.

Bounded-delay resource model. Let *capacity* $0 \leq c \leq 1$ be a fraction of the resource assigned to a component and $\delta \geq 0$ the maximum time the component may have to wait to receive this fraction. A resource is called a *bounded-delay resource* $R = (c, \delta)$ if for any $L > 0$ it can guarantee allocations of at least $c \cdot L$ units of the resource in any interval of the length $L + \delta$ [59].

The motivation for the bounded-delay resource model comes from the fact that the resource demand of a component cannot be precisely described only with a required fraction of the resource. This is so, because different components may have considerably different delay requirements [57]. The choice of the delay bound δ addresses the trade-off between high context switch costs (smaller δ) and high task execution latencies (larger δ).

For a given bounded-delay resource $R = (c, \delta)$ the resource *supply bound function* $\mathtt{sbf}_R : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ maps $\tau \in \mathbb{R}_{\geq 0}$ into the minimum supply of the resource R over all time intervals of size τ. From the definition of the bounded-delay resource model it directly follows:

$$\mathtt{sbf}_R(\tau) = \mathtt{sbf}_{(c,\delta)}(\tau) = \begin{cases} 0, & \text{if } \tau \leq \delta, \\ c(\tau - \delta), & \text{if } \tau > \delta. \end{cases}$$

4.2.2 Task Group Composition

We first briefly review the results from [68, 69] for schedulability conditions under the bounded-delay model and EDF scheduling algorithm. Then we apply and generalize them for the task model used in this chapter.

Let W be a set of independent and preemptive tasks that share the same resource, and let R be a bounded-delay resource model. We say that (W, R, EDF) is *schedulable* if under every instance of allocations of the resource R there exists a feasible EDF schedule for W [68]. If (W, R, EDF) is schedulable then the set of tasks W under the resource $R = (c, \delta)$ and the EDF scheduling algorithm can be abstracted as a single requirement (c, δ), i.e., no global knowledge of task internals is necessary. The discussion on how to schedule several (c, δ) resource requirements can be found in [57].

Let W be a set of periodic tasks $t_j = (p_j, e_j)$, where p_j is the period, e_j is the worst-case execution time (wcet) requirement of the task t_j and the deadline of each task is assumed to be equal to its period. For a given set of tasks W the resource *demand bound function* $\text{dbf} :$ $\mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ maps $\tau \in \mathbb{R}_{\geq 0}$ into the maximum resource demand over all time intervals of size τ. For the EDF scheduling algorithm we have $\text{dbf}_W(\tau) = \sum_{t_j \in W} \lfloor \tau/p_j \rfloor \cdot e_j$.

For the case of periodic workloads W, Thm. 1 in [69] gives the sufficient and necessary condition for schedulability of (W, R, EDF): (W, R, EDF) is schedulable iff for all $0 < \tau \leq 2 \cdot \text{lcm}_W$ maximal resource demand is no greater than the minimum resource supply, i.e., $\text{dbf}_W(\tau) \leq \text{sbf}_R(\tau)$. In previous condition lcm_W is the least common multiple of the periods in W.

Finally, Thm. 3 in [69] gives a general schedulability condition for the case of other workload models W for which dbf_W can be computed: (W, R, EDF) is schedulable iff for all $\tau > 0$ we have $\text{dbf}_W(\tau) \leq \text{sbf}_R(\tau)$.

We apply this result for the case of the aperiodic workload defined with the task arrival

W	t_1	t_2	t_3	t_{13}	t_{23}
σ	1	1	3	1	1
ρ	1/2	1/3	1/2+1/3	1/2	1/3
d	2/3	2	1	2/3+1	2+1
e	0.1	0.3	0.1	0.1+0.1	0.3+0.1

Table 4.1. Temporal interface and wcet's for tasks in Fig. 4.1(a)

functions and delays. Let W be a set of tasks $t_j = (a_j, d_j, e_j)$, where a_j is the arrival function, d_j the delay and e_j the wcet of the task t_j, and let $R = (c, \delta)$ be the bounded-delay resource model. To apply the theorem we first compute the demand bound function of the task t_j. We note that there are at most $\lfloor a_j(\tau - d_j) \rfloor$ invocations of the task t_j that are released and required to complete in an interval of time of size τ. Therefore, we have

$$
\mathtt{dbf}_{t_j}(\tau) = \begin{cases} 0, & \text{if } \tau \leq d_j, \\ \lfloor a_j(\tau - d_j) \rfloor \cdot e_j, & \text{if } \tau > d_j. \end{cases}
$$

The demand bound function of the total workload set W is $\mathtt{dbf}_W(\tau) = \sum_{t_j \in W} \mathtt{dbf}_{t_j}(\tau)$. Thus, both $\mathtt{dbf}_W$ and $\mathtt{sbf}_R$ are known, and we can apply Thm. 3 [69] to check if (W, R, EDF) is schedulable.

Given the task set W let c_W be the *capacity function* that maps each bounded delay $\delta \geq 0$ to the smallest resource fraction $c_W(\delta)$ such that the component (W, R, EDF) is schedulable with $R = (c_W(\delta), \delta)$. Tab. 4.1 shows an instance of the task workload of the component in Fig. 4.1, with each task modeled as a bursty arrival task. Fig. 4.2 shows capacity functions for each W consisting of only a single task $W = \{t_j\}$. For such a simple set W, the analytical expression for c_W can be derived. For instance, we have $c_W(0) = \max\{e_j \cdot \rho_j, \ \sigma_j \cdot e_j / d_j\}$, or $\delta_1 = \mathtt{inv}(c_W)(1) = d_j - \sigma_j \cdot e_j$.

In the rest of the section we assume that capacity functions are computed in some sufficiently large number N_c of points. Also, for two functions, g_1 and g_2, with arbitrary domain set X and range set $\mathbb{R}$, and for a relation $\phi \in \{<, \leq, >, \geq\}$, we write $g_1 \ \phi \ g_2$, if
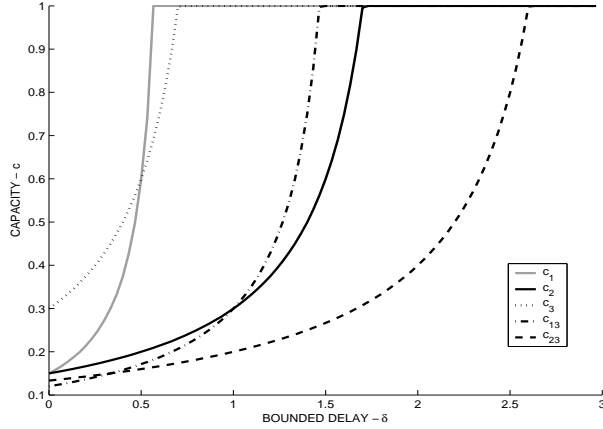
Figure 4.2. Capacity functions for Tab. 4.1

$g_1(x) \; \phi \; g_2(x)$ for all $x \in X$. For instance, $g_1 \; > \; 1$ means $g_1(x) \; > \; 1$ for all $x \in X$. Similarly, the function $g_1 + g_2$ is defined with $(g_1 + g_2)(x) = g_1(x) + g_2(x)$ for all $x \in X$.

Component composition. In the formalism that we present in the next section the capacity function c_W represents a part of the interface of the component consisting of the task set W. In order to compose such components we need to compose resource assumptions in the form of such capacity functions. For this we again recall Thm. 3 [69], but now for the workload consisting of two bounded-delay tasks, $\{(c_1, \delta_1), (c_2, \delta_2)\}$. It follows from the theorem that this workload can be abstracted by the bounded-delay resource (c, δ), where $c = c_1 + c_2$ and $\delta = \min\{\delta_1, \delta_2\}$. This equation shows how to compute capacity function of the component composition: If two components (workloads) W_1 and W_2 are specified with their respective capacity functions c_{W_1} and c_{W_2}, the sum of two functions, $c_{W_1} + c_{W_2}$, ensures schedulability of the composition. That is why, in our interface algebra (Sec. 4.3.2), when we perform component composition we add the corresponding capacity functions. Note that such an operation is associative. The task group composition, which we previously explained in this subsection, does not have that property.

4.3 Task Sequence Interfaces

We start this section by motivating the assume-guarantee principle of the formalism and introducing interface predicates that are used in compatibility and refinement checking. In the subsection 4.3.2 we formally define interfaces and prove an important proposition about independent refinement.

4.3.1 Informal Description

Let a component implement a single task sequence, i.e., let it have a single input port i and a single output port o. An interface is a constraint on the environment consisting of an input assumption and an output guarantee parts [13, 14]. In our formalism the values of the interface input and output ports are arrival functions. Let $\mathbb{A}$ be the set of all arrival functions, i.e., the set of all monotonically increasing functions $a : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. An interface assumption may be that the input arrival function $i \in \mathbb{A}$ is bounded by a given function $a \in \mathbb{A}$, i.e., $i \leq a$. Given the maximal delay $d \in \mathbb{R}_{\geq 0}$ of the component, for the arrival function of the output o we have the output guarantee $o(\tau) \leq i(\tau + d)$ for all $\tau \in \mathbb{R}_{\geq 0}$. This inequality holds because, if the delay is at most d, then for all input requests in an interval of $\tau + d$ units, the outputs are produced in an interval of at least τ units. Let i^d be the function defined with $i^d(\tau) = i(\tau + d)$.

More complex interface includes a measure of resource consumption. We assume that such an interface also contains an input port r whose value is the capacity function of the component. Let $\mathbb{C}$ be the set of all capacity functions, i.e., the set of all monotonically increasing functions $c : \mathbb{R}_{\geq 0} \to [0, 1]$. The resource capacity assumption is $r \geq c$. Formally, the input predicate of the interface is $r \geq c \ \wedge \ i \leq a$, and the output predicate is $o \leq i^d$. This interface asserts that "the environment provides capacity larger than c and input requests upper bounded by a and the component produces outputs with delay smaller

than d". Fig. 4.3 graphically represents an interface of a component that implements single sequence consisting only task t_1.

$$i_1 \leq a_1 \xrightarrow{\quad i_1 \quad} \boxed{\begin{array}{ccc} a_1 & t_1 & d_1 \\ & F_1 & \\ c_1 & & \end{array}} \xrightarrow{\quad o_1 \quad} o_1 \leq i_1^{d_1}$$

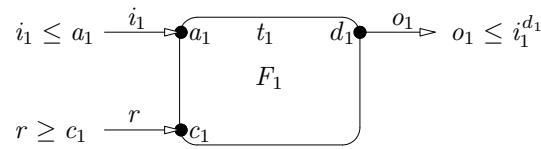$$r \geq c_1 \xrightarrow{\quad r \quad}$$

Figure 4.3. Interface for single task sequence

Let a component implements $n \in \mathbb{N}_{>0}$ task sequences through pairs (i_j, o_j) of input-output ports $(j = 1, \ldots, n)$. The interface of such a component bounds arrival function a_j of i_j and delay d_j of o_j for each $j = 1, \ldots, n$. Such a workload is still to be executed with a single resource partition requirement given with a capacity function c. Fig. 4.4 shows an interface with two single-task sequences and the corresponding predicates. We write F_W for the interface obtained by task composition of tasks in the set W.

$$\begin{array}{l} i_2 \leq a_2 \\ i_3 \leq a_3 \end{array} \xrightarrow[\quad i_3 \quad]{\quad i_2 \quad} \boxed{\begin{array}{ccc} a_2 & t_2 & d_2 \\ a_3 & t_3 & d_3 \\ & F_{2,3} & \\ c_{2,3} & & \end{array}} \xrightarrow[\quad o_3 \quad]{\quad o_2 \quad} \begin{array}{l} o_2 \leq i_2^{d_2} \\ o_3 \leq i_3^{d_3} \end{array}$$

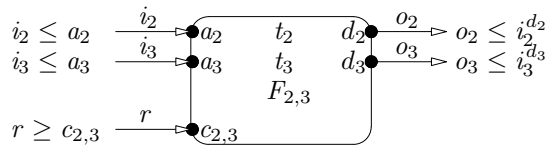$$r \geq c_{2,3} \xrightarrow{\quad r \quad}$$

Figure 4.4. Interface for multiple task sequences

We introduce two operations to construct more complex interfaces from the simpler ones. The *composition* operation puts together input-output ports of the two interfaces, and sums their resource assumptions, i.e., their capacity functions, as presented at the end of Sec. 4.2.2. For compatibility we check whether the sum is larger than 1. Fig. 4.5 shows the interface resulting by composing an interface from Fig. 4.4 with an interface F_3. Thus, the interface describes three single-task sequences, t_1, t_2, and t_3.

The *connection* operation connects tasks of an interface into sequences. This operation extends the set of interface sequences, i.e., previously present input-output ports are still

$$\begin{array}{llll}
i_1 \le a_1 & \xrightarrow{i_1} & a_1 \quad t_1 \quad d_1 & \xrightarrow{o_1} o_1 \le i_1^{d_1} \\
i_2 \le a_2 & \xrightarrow{i_2} & a_2 \quad t_2 \quad d_2 & \xrightarrow{o_2} o_2 \le i_2^{d_2} \\
i_3 \le a_3 & \xrightarrow{i_3} & a_3 \quad t_3 \quad d_3 & \xrightarrow{o_3} o_3 \le i_3^{d_3} \\
& & F_1 \| F_{2,3} & \\
r \ge c_1 + c_{2,3} & \xrightarrow{r} & c_1 + c_{2,3} &
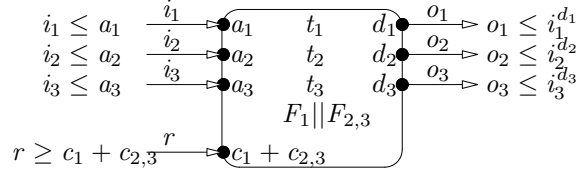\end{array}$$

Figure 4.5. Interface composition

part of the interface. Fig. 4.6 shows the interface from Fig. 4.5 after the two-task sequence $\pi = t_1 t_2$ is appended using the connection operation. The figure shows that the resource capacity assumption is not changed through connection, and that the delay guarantee of a new sequence is computed as a sum of delays of individual tasks of the sequence. However, the interface input assumptions describe the most general constraint on arrival rates of the the extended set of sequences. In particular, there is a constraint for each task that occurs in a sequence of the interface, $i_1 + i_{12} \le a_1$, $i_2 + i_{12}^{d_1} \le a_2$, and $i_3 \le a_3$. For instance, the rate of requests for task sequences that contain t_2, i.e., the sum of arrival functions i_2 and i_{12} (delayed for d_1), is bounded by a_2.

$$\begin{array}{llll}
& \xrightarrow{i_1} & t_1 \quad d_1 & \xrightarrow{o_1} o_1 \le i_1^{d_1} \\
& \xrightarrow{i_{12}} & t_{12} \quad d_1 + d_2 & \xrightarrow{o_{12}} o_{12} \le i_{12}^{d_1 + d_2} \\
& \xrightarrow{i_2} & t_2 \quad d_2 & \xrightarrow{o_2} o_2 \le i_2^{d_2} \\
& \xrightarrow{i_3} & t_3 \quad d_3 & \xrightarrow{o_3} o_3 \le i_3^{d_3} \\
& & (F_1 \| F_{2,3}) \oplus S_{12} & \\
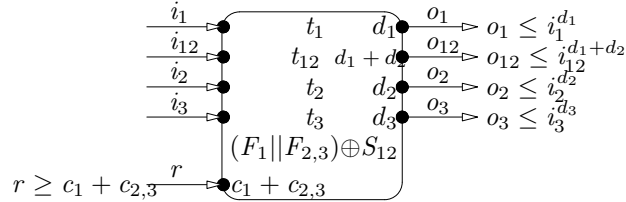r \ge c_1 + c_{2,3} & \xrightarrow{r} & c_1 + c_{2,3} &
\end{array}$$

Figure 4.6. Interface connection

In such a general definition of connection, i.e., with arbitrary arrival rates of tasks that are contained in the connected sequence, the constraints are not of the simple form $i \le a$. To illustrate this we consider bursty arrival functions with a simple example of the two-task connection sequence $\pi = t_1 t_2$. We assume that t_1 is specified with arrival function $a_1(\tau) = \sigma_1 + \rho_1 \cdot \tau$ and delay d_1, and t_2 with $a_2(\tau) = \sigma_2 + \rho_2 \cdot \tau$ and d_2. If we assume input arrival function of task t_1 and t_2 sequences to be 0, i.e., $i_1 = 0$ and $i_2 = 0$, then

98

for the input arrival function i_{12} of $t_1 t_2$ sequence we have the constraints $i_{12} \leq a_1$ and $i_{12}^{d_1} \leq a_2$. If we assume $i_{12}(\tau) = \sigma + \rho \cdot \tau$ this is equivalent to $\sigma + \rho \cdot \tau \leq \sigma_1 + \rho_1 \cdot \tau$, and $\sigma + \rho \cdot (\tau + d_1) \leq \sigma_2 + \rho_2 \cdot \tau$. The possible values of σ and ρ parameters of i_{12} are shown as a shaded area on the rightmost graph of Fig. 4.7. There is a tradeoff in choice of the parameters, and this area cannot be specified in the $i_{12} \leq a_{12}$ form.
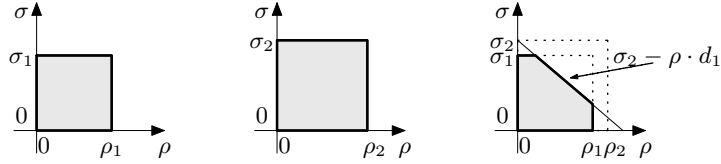

Figure 4.7. Bursty functions for $t_1 t_2$ sequence

If the connection operation with the sequence $t_1 t_2 t_3$ is applied on the interface $F_1 \| F_{2,3}$ the resulting interface $(F_1 \| F_{2,3}) \oplus \pi_{123}$ contains an input-output port for each of the sequences t_1, t_2, t_3, and t_{123}. The input assumptions are $i_1 + i_{123} \leq a_1$, $i_2 + i_{123}^{d_1} \leq a_2$, and $i_3 + i_{123}^{d_1+d_2} \leq a_3$. In fact, the connection operation is defined with a set of sequences, each of which does not contain a cycle of tasks. For instance, the interface $(F_1 \| F_{2,3}) \oplus \{\pi_{12}, \pi_{21}\}$, consists of the input assumptions $i_1 + i_{12} + i_{21}^{d_2} \leq a_1$, $i_2 + i_{12}^{d_1} + i_{21} \leq a_2$, and $i_3 \leq a_3$.

Finally, the *refinement* relation is defined as an implication from more abstract to more refined interface, in order to be able to substitute a refined component for an abstract one. A more refined version of a component makes weaker input assumptions and stronger output guarantees than a more abstract description. In our context, an interface can be refined by either decreasing capacity function, or increasing arrival function of a task, or decreasing a deadline of a task.

In Sec. 4.3.2 we formally define the refinement relation and argue about its properties. In the following paragraphs we illustrate component composition and interface refinement at the task composition level. We first show examples of interface refinement through modification of the capacity function, while keeping other interface parameters constant. In

$$a_{1,3}(\tau) = 1 + \tau/2, a_{2,3}(\tau) = 1 + \tau/3, d_{1,3} = 2/3 + 1, d_{2,3} = 2 + 1$$

F	F_a	F_b	F_c
Expr.	$F_{1,2,3} \oplus \{\pi_{13}, \pi_{23}\}$	$(F_{1,2} \| F_3) \oplus \{\pi_{13}, \pi_{23}\}$	$(F_1 \| F_2 \| F_3) \oplus \{\pi_{13}, \pi_{23}\}$
c_F	c_a	c_b	α_c
a_F		$(a_{1,3}, a_{2,3})$	
d_F		$(d_{1,3}, d_{2,3})$	

Table 4.2. Interface refinement

general, if initial task composition is performed with larger task sets, then refined interface is obtained. The extreme case is when all tasks in the task graph are composed. The other extreme case, when each task is considered as a separate component, results in more abstract interface.
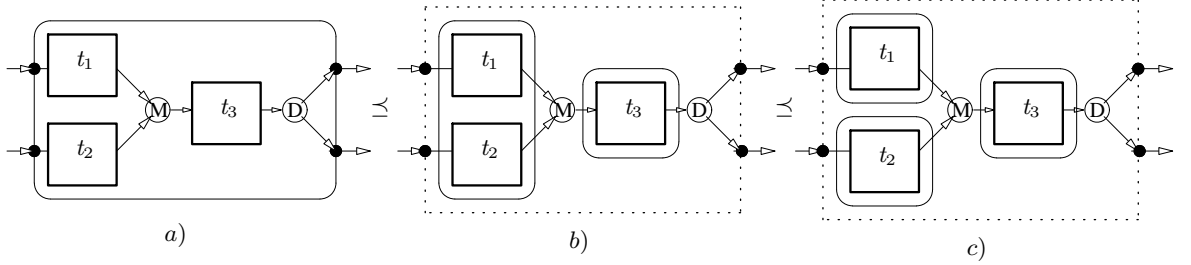

Figure 4.8. (a) $F_a = F_{1,2,3} \oplus \{\pi_{13}, \pi_{23}\}$; $\quad$ (b) $F_b = (F_{1,2} \| F_3) \oplus \{\pi_{13}, \pi_{23}\}$; $\quad$ (c) $F_c = (F_1 \| F_2 \| F_3) \oplus \{\pi_{13}, \pi_{23}\}$

Fig. 4.8 shows three interfaces F_a, F_b and F_c of the component shown in Fig. 4.1. Task composition is shown with rounded rectangles, and interface composition with dashed rectangles. The interface expressions and predicates are given in Tab. 4.2. In the example we use the task graph and data from Fig. 4.1 and Tab. 4.1. We assume that only task sequences $t_1 t_3$ and $t_2 t_3$, and not sequences t_i for $i = 1, 2, 3$, are implemented by the component. Therefore, we assume $a_3 = a_1^{d_1} + a_2^{d_2}$. All three interfaces have the same arrival functions $(a_{1,3}, a_{2,3})$ and delays $(d_{1,3}, d_{2,3})$. However, corresponding capacity functions c_a, c_b and c_c are different and Fig. 4.9 shows that $c_a \leq c_b \leq c_c$. Therefore, we have $F_a \preceq F_b \preceq F_c$.
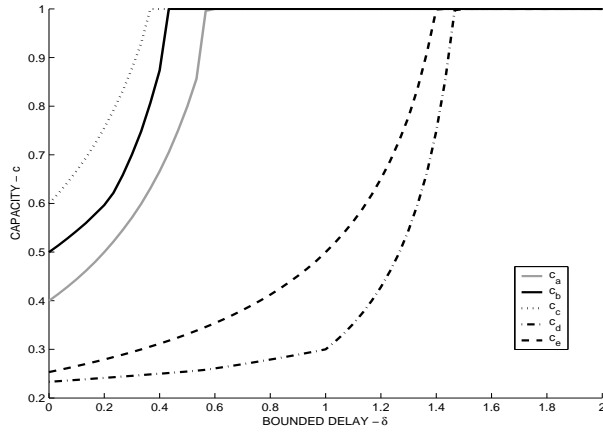
Figure 4.9. Capacity functions from Tab. 4.2

For some interfaces it is possible to increase arrival functions or decrease delays of the interface task sequences while keeping resource capacity function constant. In some cases it is even possible to add a new task sequence to the component without affecting the capacity function. For instance, let $t_1 = (a_1, d_1, e_1) = (1 + \tau/2, 1, 0.2)$ and $t_2 = (a_2, d_2, e_2) = (1 + \tau/2, 2, 0.2)$, i.e., let the two tasks differ only in delay. It can be showed that the capacity functions c_1 and $c_{1,2}$, the capacity functions for independent task sets $\{t_1\}$ and $\{t_1, t_2\}$ are equal, $c_1 = c_{1,2}$ This comes as a consequence of the small delay requirement d_1. So, we have that $F_{1,2} \preceq F_1$. As explained in the following subsection, the definition of the refinement allows larger number of ports in the refined interface.

4.3.2 Interface Algebra

Let T be a set of tasks. A *task sequence* $\pi = t_1 t_2 \ldots t_k$ is a finite sequence of different tasks $t_j \in T$, i.e., for all $1 \leq i < j \leq k$ we have $t_i \neq t_j$, and t_i has to complete before t_j starts execution.

An *interface* $F = (S_F, T_F^+, A_F, D_F, c_F)$ consists of:

- A set S_F of task sequences, and a set T_F^+ of *available* tasks.

101

The set of available tasks $T_F^+ \subseteq T$ is a set of tasks available for the implementation of the interface F or its refinements. Let the set $T_F \subseteq T_F^+$ of tasks contain all tasks in all sequences of S_F, $T_F = \{t \in \pi \mid \pi \in S_F\}$. For each task sequence $\pi \in S_F$ there exist an input port i_π and an output port o_π. Let $I_F = \{i_\pi \mid \pi \in S_F\} \cup \{r\}$, $O_F = \{o_\pi \mid \pi \in S_F\}$, and $P_F = I_F \cup O_F$. The type of a port $x \in P_F$ is $\mathbb{A}$ if $x \neq r$, and $\mathbb{C}$ if $x = r$. Let a *valuation* v be a function on P_F that maps each port $x \in P_F$ to a value of the port type $v(x)$.

- A function A_F that maps each task $t \in T_F$ into an arrival rate function $A_F(t) \in \mathbb{A}$, and a function D_F that maps each task $t \in T_F$ into a delay $D_F(t) \in \mathbb{R}_{\geq 0}$. Given a task sequence $\pi \in S_F$, let $D_F(\pi)$ be the sum of delays of its tasks, $D_F(\pi) = \sum_{t \in \pi} D_F(t)$. By definition, for an empty task sequence ϵ we have $D_F(\epsilon) = 0$.

- A capacity function $c_F \in \mathbb{C}$.

The *input* predicate $\phi_F^I = \phi^I(S_F, A_F, D_F, c_F)$ over input ports I_F is defined to be

$$\phi_F^I = r \geq c_F \;\wedge\; \bigwedge_{t \in T_F} \Big(\sum_{\pi = \pi_1 \cdot t \cdot \pi_2 \in S_F} i_\pi^{D_F(\pi_1)} \leq A_F(t) \Big). \tag{4.1}$$

The *output* predicate $\phi_F^O = \phi^O(S_F, D_F)$ over P_F is defined to be

$$\phi_F^O = \bigwedge_{\pi \in S_F} o_\pi \leq i_\pi^{D_F(\pi)}. \tag{4.2}$$

The interface *algebra* for real-time components consists of:

- A partial binary function called *composition*, mapping two interfaces F and G into an interface $F \| G$.

The composition $F \| G$ is defined if $T_F^+ \cap T_G^+ = \emptyset$, and if not $c_F + c_G > 1$, i.e., if $(c_F + c_G)(0) \leq 1$. If $F \| G$ is defined, then $S_{F \| G} = S_F \cup S_G$, $T_{F \| G}^+ = T_F^+ \cup T_G^+$, and $c_{F \| G} = \min\{c_F + c_G, 1\}$. In addition, $A_{F \| G}(t) = A_F(t)$ if $t \in T_F$, and $A_{F \| G}(t) =$

$A_G(t)$ if $t \in T_G$. Similarly, $D_{F\|G}(t) = D_F(t)$ if $t \in T_F$, and $D_{F\|G}(t) = D_G(t)$ if $t \in T_G$.

- A partial binary function called *connection*, mapping an interface F and a set S of task sequences to an interface $F \oplus S$.

  The connection $F \oplus S$ is defined if T_F contains all tasks in all sequences of S, i.e., if for all sequences π of S, every task t of π is also an element of T_F.

  If $F \oplus S$ is defined, then $S_{F \oplus S} = S_F \cup S$, $T_{F \oplus S}^+ = T_F^+$, $A_{F \oplus S} = A_F$, $D_{F \oplus S} = D_F$, and $c_{F \oplus S} = c_F$.

- A binary relation $\preceq$ between interfaces, called *refinement*. If $F' \preceq F$ then the interface F' is said to *refine* the interface F, and F is said to *abstract* F'.

  An interface F' refines a component F if (a) $S_{F'} \supseteq S_F$, (b) $T_{F'}^+ = T_F^+$, and (c) for each valuation v on P_F there exists a valuation v' on $P_{F'}$ such that $v = v'$ on P_F, and both predicates $\phi_F^I \Rightarrow \phi_{F'}^I$ and $\phi_{F'}^O \Rightarrow \phi_F^O$ are valid.

We next formally present four different ways for interface refinement. The interface $F = (S_F, A_F, D_F, c_F)$ is refined by an interface F' if one of the following:

- The connection operator is applied. Formally, $F' = F \oplus S \preceq F$, for each set of task sequences S.

  The refinement condition (a) is satisfied since $S_{F'} = S_F \cup S \supseteq S_F$, and (b) since $T_{F'}^+ = T_{F \oplus S}^+ = T_F^+$.

  If in the refinement condition (c) we define $v'(x) = 0$ for each $x \in P_{F'} \setminus P_F$ (i.e., arrival functions of all sequences from $S \setminus S_F$ are 0), we have $\phi_{F'}^I = \phi_F^I$ and $\phi_{F'}^O = \phi_F^O$, and therefore, $F' \preceq F$.

- The function A_F is modified to $A_{F'}$ by increasing arrival function $A_F(t)$ for some tasks t in T_F, i.e., $F' = (S_F, A_{F'}, D_F, c_F) \preceq F$.

103

The refinement condition (c) is satisfied since for any valuation on P_F from Equ. 4.1 we have $\phi_F^I = \phi^I(S_F, A_F, D_F, c_F) \Rightarrow \phi^I(S_F, A_{F'}, D_F, c_F) = \phi_{F'}^I$, and from Equ. 4.2 we have $\phi_{F'}^O = \phi_F^O$.

- The function D_F is modified to $D_{F'}$ by decreasing $D_F(t)$ for some tasks t in T_F, i.e., $F' = (S_F, A_F, D_{F'}, c_F) \preceq F$.

  In this case, $\phi_F^I = \phi^I(S_F, A_F, D_F, c_F) \Rightarrow \phi^I(S_F, A_F, D_{F'}, c_F) = \phi_{F'}^I$, and $\phi_{F'}^O = \phi^O(S_F, D_{F'}) \Rightarrow \phi^O(S_F, D_F) = \phi_F^O$.

- The function c_F is decreased to $c_{F'}$, i.e., $F' = (S_F, A_F, D_F, c_{F'}) \preceq F$.

  In this case, $\phi_F^I = \phi^I(S_F, A_F, D_F, c_F) \Rightarrow \phi^I(S_F, A_F, D_F, c_{F'}) = \phi_{F'}^I$, and $\phi_{F'}^O = \phi_F^O$.

The next two propositions formalize the two properties that will further be explored in the following section.

Proposition 12 (Incremental Design) *For all interfaces F_1, F_2, and F_3, and all sets of task sequences S_1 and S_2*

1. *If $(F_1\|F_2)\|F_3$ is defined then $F_1\|(F_2\|F_3)$ is defined, and $(F_1\|F_2)\|F_3 = F_1\|(F_2\|F_3)$.*

2. *If $(F_1 \oplus S_1) \oplus S_2$ is defined then $(F_1 \oplus S_2) \oplus S_1$ is defined, and $(F_1 \oplus S_1) \oplus S_2 = (F_1 \oplus S_2) \oplus S_1$.*

3. *If $(F_1\|F_2) \oplus S_1$ and $F_1 \oplus S_1$ are defined then $(F_1 \oplus S_1)\|F_2$ is defined, and $(F_1\|F_2) \oplus S_1 = (F_1 \oplus S_1)\|F_2$.*

Proof.

1. The two expressions are defined if $T_{F_j} \cap T_{F_k} = \emptyset$ for $1 \leq j < k \leq 3$. The equality follows from $(S_{F_1} \cup S_{F_2}) \cup S_{F_3} = S_{F_1} \cup (S_{F_2} \cup S_{F_3})$ and $(c_{F_1} + c_{F_2}) + c_{F_3} =$

$c_{F_1} + (c_{F_2} + c_{F_3})$. In addition, both $A_{(F_1 \| F_2) \| F_3}(t)$ and $A_{F_1 \| (F_2 \| F_3)}(t)$ are equal $A_{F_j}(t)$ if $t \in T_{F_j}$ for $j = 1, 2, 3$. Similar argument holds for $D_{(F_1 \| F_2) \| F_3} = D_{F_1 \| (F_2 \| F_3)}$.

2. $(F_1 \oplus S_1) \oplus S_2 = F_1 \oplus (S_1 \cup S_2)$ follows directly from the connection definition.

3. Similar to 1.

Proposition 13 (Independent Refinement) *For all interfaces F, F' and G, and all sets S of task sequences,*

1. *If $F \| G$ is defined and $F' \preceq F$, then $F' \| G$ is defined and $F' \| G \preceq F \| G$.*

2. *If $F \oplus S$ is defined and $F' \preceq F$, then $F' \oplus S$ is defined and $F' \oplus S \preceq F \oplus S$.*

Proof.

To simplify notation of the proof we first introduce the following predicates: $\phi_F^r = r \geq c_F$, $\phi_F^i = \bigwedge_{t \in T_F} \left(\sum_{\pi = \pi_1 \cdot t \cdot \pi_2 \in S_F} i_\pi^{D_F(\pi_1)} \right) \leq A_F(t)$, and $\phi_S^O = \bigwedge_{\pi \in S} o_\pi \leq i_\pi^{D_F(\pi)}$. Note that $\phi_F^I = \phi_F^r \wedge \phi_F^i$.

1. Since $T_{F'}^+ \cap T_G^+ = T_F^+ \cap T_G^+ = \emptyset$, and $c_{F'} \leq c_F$, the interface $F \| G$ is defined.

   (a) $S_{F' \| G} = S_{F'} \cup S_G \supseteq S_F \cup S_G = S_{F \| G}$.

   (b) $T_{F' \| G}^+ = T_{F'}^+ \cup T_G^+ = T_F^+ \cup T_G^+ = T_{F \| G}^+$.

   (c) If $F' \preceq F$ then for each valuation on P_F there exists a valuation on P_F' such that the predicate $\phi_F^I \Rightarrow \phi_{F'}^I$ is valid, i.e., both $\phi_F^r \Rightarrow \phi_{F'}^r$ and $\phi_F^i \Rightarrow \phi_{F'}^i$ are valid. If $\phi_F^r \Rightarrow \phi_{F'}^r$ is valid for each r, then $c_F \geq c_{F'}$, and therefore, $c_{F \| G} = c_F + c_G \geq c_{F'} + c_G = c_{F' \| G}$, Consequently, for each r the predicate $\phi_{F \| G}^r \Rightarrow \phi_{F' \| G}^r$ is valid. If $F \| G$ is defined then $\phi_{F \| G}^i = \phi_F^i \wedge \phi_G^i$ and $\phi_{F \| G}^O = \phi_F^O \wedge \phi_G^O$. So, if both $\phi_F^i \Rightarrow \phi_{F'}^i$ and $\phi_{F \| G}^i = \phi_F^i \wedge \phi_G^i$ are valid, then $\phi_{F' \| G}^i = \phi_{F'}^i \wedge \phi_G^i$ is valid, i.e., $\phi_{F \| G}^i \Rightarrow \phi_{F' \| G}^i$ is valid. Similarly, if both $\phi_{F'}^O \Rightarrow \phi_F^O$ and $\phi_{F' \| G}^O = \phi_{F'}^O \wedge \phi_G^O$ are valid, then $\phi_{F \| G}^O = \phi_F^O \wedge \phi_G^O$ is valid, i.e., $\phi_{F' \| G}^O \Rightarrow \phi_{F \| G}^O$ is valid.

2. Since $F \oplus S$ is defined and $T_{F'} \supseteq T_F$, the interface $F' \oplus S$ defined.

(a) $S_{F' \oplus S} = S_{F'} \cup S \supseteq S_F \cup S = S_{F \oplus S}$.

(b) $T^+_{F' \oplus S} = T^+_{F'} = T^+_F = T^+_{F \oplus S}$.

(c) If $\phi^r_F \Rightarrow \phi^r_{F'}$ is valid for each r, then $c_F \geq c_{F'}$, and therefore, $c_{F \oplus S} = c_F \geq c_{F'} = c_{F' \oplus S}$, Consequently, for each r the predicate $\phi^r_{F \oplus S} \Rightarrow \phi^r_{F' \oplus S}$ is valid. If both $\phi^O_{F'} \Rightarrow \phi^O_F$ and $\phi^O_{F' \oplus S} = \phi^O_{F'} \wedge \phi^O_S$ are valid, then $\phi^O_{F \oplus S} = \phi^O_F \wedge \phi^O_S$ is valid, i.e., $\phi^O_{F' \oplus S} \Rightarrow \phi^O_{F \oplus S}$ is valid. If $F' \preceq F$ then $\phi^O_{F'} \Rightarrow \phi^O_F$, i.e., for each sequence π of S_F the predicate $o_\pi \leq i^{D_{F'}(\pi)}_\pi$ implies the predicate $o_\pi \leq i^{D_F(\pi)}_\pi$. This implies $i^{D_{F'}(\pi)}_\pi \leq i^{D_F(\pi)}_\pi$, i.e., $D_{F'}(\pi) \leq D_F(\pi)$. Since, for each $t \in T_F$, we have $\pi = t \in S_F$ then $D_{F'}(t) \leq D_F(t)$. Similarly, from $\phi^i_F \Rightarrow \phi^i_{F'}$ we have $A_{F'}(t) \geq A_F(t)$ for each $t \in T_F$. For a given $t \in T_{F \oplus S}$, if $\sum_{\pi = \pi_1 \cdot t \cdot \pi_2 \in S_F} i^{D_F(\pi_1)}_\pi \leq A_F(t)$, and if we take $i_\pi = 0$ for each $\pi \in S_{F'} \setminus (S_F \cup S)$, we have

$$\sum_{\pi \in S_{F' \oplus S}} i^{D_{F'}(\pi_1)}_\pi \leq \sum_{\pi \in S_{F \oplus S}} i^{D_F(\pi_1)}_\pi \leq A_F(t) \leq A_{F'}(t)$$

Consequently, $\phi^i_{F \oplus S} \Rightarrow \phi^i_{F' \oplus S}$.

Remark 1 *If $\mathcal{F}(F_1, ..., F_k, S_1, ..., S_l)$ is an interface computed by applying finitely many composition and connection operations on interfaces $F_1, ..., F_k$ and task sequences $S_1, ..., S_l$, and $F'_1 \preceq F_1$, ... $F'_k \preceq F_k$, then $\mathcal{F}(F'_1, ..., F'_k, S_1, ..., S_l)$ is defined and $\mathcal{F}(F'_1, ..., F'_k, S_1, ..., S_l) \preceq \mathcal{F}(F_1, ..., F_k, S_1, ..., S_l)$.*

4.4 Real-Time Component-Based Design

4.4.1 Incremental Design

Since the interface composition is associative, the order in which we can compose components is arbitrary. Moreover, this means that compatibility can be checked even before all interfaces are fully specified, i.e., before the system becomes closed. Formally, we can check whether $k > 0$ interfaces are compatible, i.e., whether $(F_1 \| \ldots \| F_{k-1} \| F_k)\theta_{1,\ldots,k-1}\theta_k$ is defined, by constructing $((F_1 \| \ldots \| F_{i-1})\theta_{1,\ldots,i-1}) \| F_i)\theta_i$ for $i = 1, \ldots, k$. The computational complexity of this operation is typically less than $m_{F_1} \cdot \ldots \cdot m_{F_k}$ because incompatible levels of services are eliminated as soon as possible. This procedure can be further improved by composing interfaces in a tree-like order, rather than in a linear order.

We demonstrate the efficiency due to the incremental design on a real-time robotic application adapted from [27]. The application consists of three subsystems, command (S_1), measurement (S_2), and control (S_3) subsystem. There is a total of five task sequences and 13 tasks. Tab. 4.3 shows details of each subsystem, and Fig. 4.10 and 4.12 show two different component decompositions of the system.

The interfaces for the components were initially designed for three levels of service: 80%, 100% and 120% of nominal arrival rates given in Tab. 4.3. The execution times and deadlines of the tasks were also part of the specification.

Let the system be composed out of components A, B and C as shown in Fig. 4.10. Results of checking for compatibility of the corresponding interfaces are shown in Fig. 4.11. Instead of showing entire capacity functions in the last two columns of the table we characterize the functions with two numbers: $c(0)$ is the resource capacity at delay 0 and δ_1 is the delay at which capacity has to be 1. The interface $(F_A \| F_B)\theta_{AB}$ consists of 5 levels of service, since 4 were eliminated due to incompatibility. Similarly, the interface for the entire system consists of 11, and not $3^3 = 27$, levels of service. Note that computing

S	S_1						S_2					S_3	
π	π_1			π_2			π_3		π_4			π_5	
t	t_{11}	t_{12}	t_{13}	t_{21}	t_{22}	t_{23}	t_{31}	t_{32}	t_{41}	t_{42}	t_{43}	t_{51}	t_{52}
ρ	0.02			0.04			0.14		0.07			0.18	
e	0.2	1.2	1.0	1.0	2.0	0.3	0.8	1.2	1.0	0.5	0.5	0.1	0.5
d	9.06	27.78	18.72	5.08	14.24	8.46	2.04	4.91	2.98	6.44	4.46	1.34	4.21
σ	1	1.16	1.64	1	1.15	1.59	1	1.18	1	1.14	1.57	1	1.23

Table 4.3. Task data for robotic application

the composition interface, $(F_A\|F_B\|F_C)\theta_{AB}$, i.e., checking for compatibility, involves both composition and connection operations of our interface algebra.
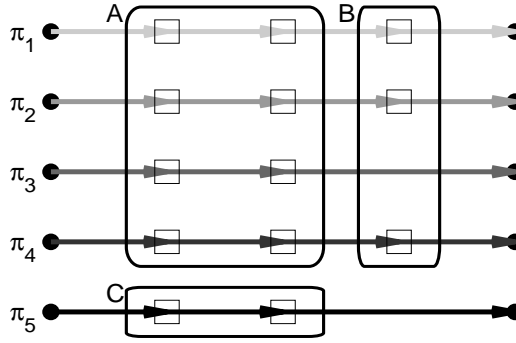

Figure 4.10. $(F_A\|F_B\|F_C)\theta_{AB}$

If the system is composed out of components a, b and C as shown in Fig. 4.12, the resulting interfaces $(F_a\|F_b)\theta_{ab}$ and $(F_a\|F_b\|F_C)\theta_{ab}$ are shown in Fig. 4.13. The table shows that with this composition only two combinations of the service is attainable even though the properties of the arrival sequences and tasks are the same in both cases. This confirms that, although interface composition is associative, the task composition is not. In particular, even though $((F_A\|F_B)\|F_C)\theta_{AB} = (F_A\|(F_B\|F_C))\theta_{AB}$, and $((F_a\|F_b)\|F_C)\theta_{ab} = (F_a\|(F_b\|F_C))\theta_{ab}$, we have that $(F_A\|F_B\|F_C)\theta_{AB}$ and $(F_a\|F_b\|F_C)\theta_{ab}$ are not equivalent.

$(F_A\|F_B)\theta_{AB}$				
k	S_A	S_B	$c(0)$	δ_1
1	80	80	0.69	1.80
2	80	100	0.71	1.80
3	80	120	0.73	1.80
4	100	100	0.85	0.80
5	100	120	0.87	0.70

$(F_A\|F_B\|F_C)\theta_{AB}$					
k	S_A	S_B	S_C	$c(0)$	δ_1
1	80	80	80	0.81	1.20
2	80	80	100	0.84	1.00
3	80	80	120	0.87	0.80
4	80	100	80	0.83	1.20
5	80	100	100	0.86	0.90
6	80	100	120	0.89	0.60
7	80	120	80	0.85	1.00
8	80	120	100	0.88	0.80
9	80	120	120	0.91	0.50
10	100	100	80	0.97	0.10
11	100	120	80	0.99	0.00

Figure 4.11. Levels of service of $(F_A\|F_B)\theta_{AB}$ and $(F_A\|F_B\|F_C)\theta_{AB}$

4.4.2 Independent Refinement

The formalism presented in Sec. 4.3 enables compositional refinement, i.e., it enables independent refinement from component interfaces to component implementations. This means that in order to refine a given composition of interfaces, it suffices to independently refine each interface and to compose the obtained refinements. Formally, if $(F_1\|\ldots\|F_k)\theta \preceq F$ and $F_j' \preceq F_j$ for $j = 1\ldots k$, then $(F_1'\|\ldots\|F_k')\theta \preceq F$. This follows from the definition of the refinement relation since a more refined version of a component makes a weaker input assumptions and stronger output guarantees than a more abstract description. The higher efficiency of such a procedure lies in the fact that now refinement checks involve smaller interfaces. In that way, a single complex problem is reduced to multiple simpler problems.

To illustrate this concept we discuss a design of a real-time application with randomly generated parameters. The underlying graph in Fig. 4.15 shows an instance of a task precedence graph consisting of 20 task sequences of length 5. We assumed that all tasks are
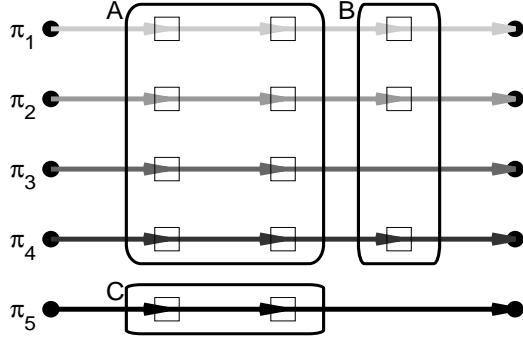
Figure 4.12. $(F_a\|F_b\|F_C)\theta_{ab}$

$(F_a\|F_b)\theta_{ab}$				
k	S_a	S_b	$c(0)$	δ_1
1	80	80	0.85	0.90
2	80	100	0.94	0.30

$(F_a\|F_b\|F_C)\theta_{ab}$					
k	S_a	S_b	S_C	$c(0)$	δ_1
1	80	80	80	0.97	0.10
2	80	80	100	1.00	0.00

Figure 4.13. Levels of service of $(F_a\|F_b)\theta_{ab}$ and $(F_a\|F_b\|F_C)\theta_{ab}$

divided into two sets of 20 tasks, such that 10 task sequences consist of tasks from one, and remaining 10 task sequences of tasks from the other set. For each sequence and for each of its 5 stages, one of 4 tasks was randomly selected. For each sequence the burst was 1 and the rate was randomly selected from the interval [0.025,0.05]. After burst σ_i and rate ρ_i were computed for each task t_i, the execution time of each task e_i was chosen randomly such that the sum of the terms $\sigma_i \cdot \rho_i \cdot e_i$ of tasks in each half of the design was in the interval [0.4,0.5].

The specification for the entire application is an interface F with a given capacity function c_F for which $c_F(0) = 0.9$ and $c_F(1) = 1$. The design goal is to implement the system as a composition of real-time components that refine the specification interface F. To that purpose the specification interface F is represented as a composition $F = F_1\|F_2$ of two interfaces, F_1 and F_2, aimed for independent implementation. Due to the random character

of the application, the capacity functions are assumed to be equal, $c_{F_1} = c_{F_2} = c_F/2$. Therefore, $c_{F_i}(0) = 0.45$ and $c_{F_i}(1) = 0.5$ $(i = 1, 2)$.

Let F_1' be the composition of components that consist of individual tasks in the upper part of the design. Let F_2' be the same for the lower part of the design. For a particular instance of the random task graph the capacity functions of F_1' and F_2' are shown in Fig. 4.14 with solid lines. The figure shows that the interface F_2' refines F_2, but F_1' does not refine F_1. However, if task composition is performed at the stage level (see Fig. 4.15), i.e., with four tasks in a component, the obtained composition interface F_1'', refines F_1, as shown in Fig. 4.14 with dashed gray line. In our repeated simulation experiments with other problem instances, there was also no need for considering composition of components that consisted of more than four tasks. Moreover, the other part of the design may be further independently refined by increasing the rate of the first task sequence as shown with the capacity function for F_2''. In some instances potential increase of rate was up to 20%. In conclusion, according to compositional refinement procedure, composition of the refinements, $F_1'' \| F_2'$, refines the original specification interface F.

4.5   Task Graph Interfaces

In this section we address the interface-based verification of real-time properties in systems with more complicated task dependencies. We study general task graphs that may contain task cycles, i.e., functional cycles. Some examples of signal processing applications with task cycles and real-time requirements can be found in [23, 24]. However, it should be noted that the related problem of performance cycles is also very relevant since nonfunctional dependency cycles often occur in multiprocessor systems with communication sharing [62]. The objective here is similar to the one discussed with respect to the task sequence case. We would like to have an interface-based theory that satisfies both the
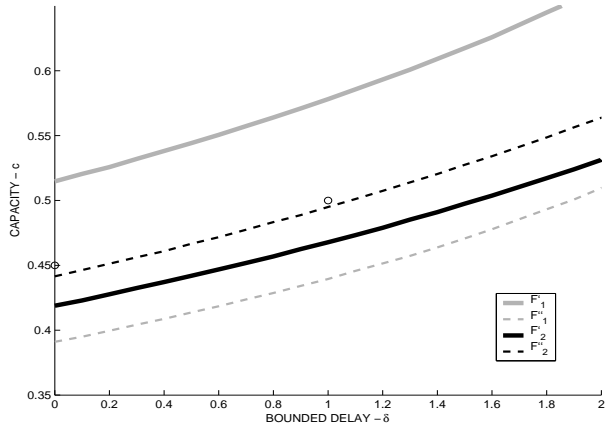
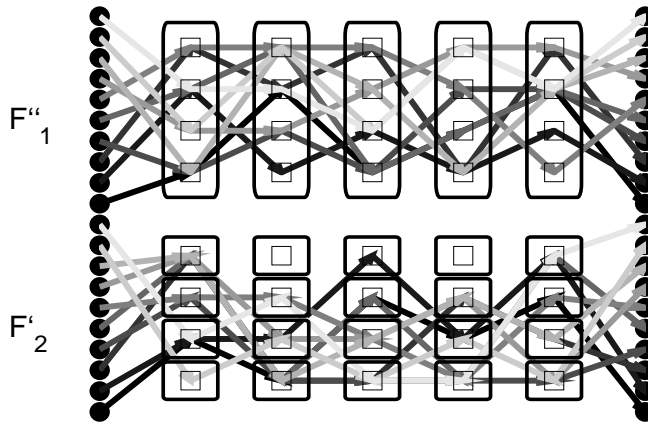Figure 4.14. Capacity functions for F'_1, F''_1, F'_2, F''_2

Figure 4.15. $F''_1 \| F'_2 \preceq F$

incremental design and independent refinement properties. We try to establish what is the minimal but complete form of interface needed in this case.

The case of functional cycles, i.e., cycles of task precedence relations, turns out to be more difficult to analyze [40]. The general input-output event model relationship is difficult to calculate and often requires complex event model propagation operations. In such approaches, fixed-point calculation is usually necessary for models with cycles. Problems occur because output event model is often too conservative to be used for a feedback input.

In [40] such an approach is taken for periodic models with jitter. However, the approach is not sufficient for general task graphs. As discussed below, we extend the periodic with jitter event model with phase information to better address graphs with cycles. Thus, we define an interface with both phase and jitter information, and for the corresponding algebra we show when it satisfies incremental design and independent refinement properties.

In this section we will assume that the occurrence of an event is modeled with a time interval that describes its timing unpredictability. An interface expresses input-output dependencies in the form of output time interval guarantees under certain input interval assumptions. The semantics of such a specification allows one input time instant to be related to multiple output time instants and vice versa. This form of interface is useful when exact event instant can hardly be specified, but when its bounds are available. It contains the graph of ports and a time interval for each port. In general, it may contain multiple tuples of intervals, but no general relationship between input and output intervals is a part of the interface. This affects composability, but allows for simple operations, i.e., simple verification.

4.5.1 Component Model

A component implements an arbitrary directed graph of tasks. Each edge represents a data (or event) precedence relationship. Each task may have single, multiple, or no inputs and outputs. After a data, or event, becomes available at its input, the task consumes the data, executes, and then produces the output data. Although in this work we do not consider graphs where producer-consumer relations are not unitary (e.g., as in general Static Data Flow models [50]), we believe that most conclusions can be extended also to these cases. We assume that task communication is performed through input data buffers, that store the arriving events before they can be processed. For instance, an event is stored while previous

events for the task are being processed, while the shared resource is being used for other tasks, or while other synchronizing events are being waited upon.

For tasks with multiple inputs we assume so called AND type of task triggering. Namely, a task is activated only when events are available at all incoming edges. We do so because in graphs with cycles closed with OR type of triggering, an event at an open, i.e., cycle-external input would lead to ever increasing number of events to be processed in the cycle (see Fig. 4.17 d). Also, if the graph contains cycles, we assume that there exist sufficient number of data/events on each cycle such that the execution does not block. To address tasks with multiple inputs and task graphs with both the cycles and open inputs, the algebra that will be presented in this chapter contains additional operation, the $join$ operation. This operation takes two input event streams, matches the corresponding events for the AND type of triggering, and generates such the matched event stream.

In [63], one of the rare methods that addresses functional cycles in a compositional manner, the event model is the *periodic with jitter* event model. This event model is not as general as the one presented in Sec. 4.3. In such a model the event period p is the same for all system ports, but different ports have different jitter values. The exact interval of an event time uncertainty is not known, i.e., the event phase is not known. In such an approach the cycles are analyzed by iterative propagation of input event parameters until all event parameters along cycle converge. If even a single task on the cycle has response time jitter, then after the first round of event model propagation the cycle-internal input jitter of the AND-activated task will be larger than the cycle-external input jitter. The simple event model allows for only a very conservative calculation of the jitter of the AND-activated task which is equal to the maximum of the two jitters. Thus, the larger jitter is propagated around the cycle again, resulting in an even larger jitter at the cycle-internal input of the AND-activated task. Obviously, this method would not converge. However, in [63] it is shown that for a restricted set of cyclic graphs (e.g. a cycle is allowed to have only one

external input) and under certain conditions that may be rather pessimistic, the fixed-point computation approach can be avoided.
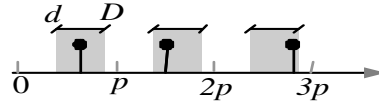

Figure 4.16. Periodic event model with jitter and phase

This problem is a consequence of the fact that event model propagation does not capture correlations between the timing of events in different event streams, in particular, in input event streams of the AND-activated task. We include a phase of an event stream, i.e., we make the event timing uncertainty fixed with respect to the period instances. The phase information increases the composability of models with task cycles, because, as will be evident from our interface algebra, it allows for better jitter calculation for the the AND-activated task. Thus, we limit the discussion here to a variant of the periodic event model with jitter and phase. This means that the occurrence of an event is assumed (or guaranteed) up to the time interval of the form $[m \cdot p + d, m \cdot p + D]$, where p is the period of the model, m is the instance of the period, and d ($d < p$) and D define the lower and upper bound of the interval. An event sequence instance specified with this event model is shown in Fig. 4.16. Note that the standard periodic with jitter event models are defined with a single value $D - d$ for each event. We present the theory for the case when event jitter is less than the period of the system ($D - d < p$). However, most of the results hold even if jitter is larger than the period, thus allowing for modeling of event bursts.

Fig. 4.17 a) shows a task, its input and output ports, and their event models. The full circles for ports denote that the time intervals are precisely known up to the period instance, i.e., the first input event occurs in the interval $[m_i \cdot p + d_i, m_i \cdot p + D_i]$ and the first output event is generated in the interval $[m_o \cdot p + d_o, m_o \cdot p + D_o]$. Such a full information for input ports is necessary for the correct join operation (Fig. 4.17 b). However, many input-
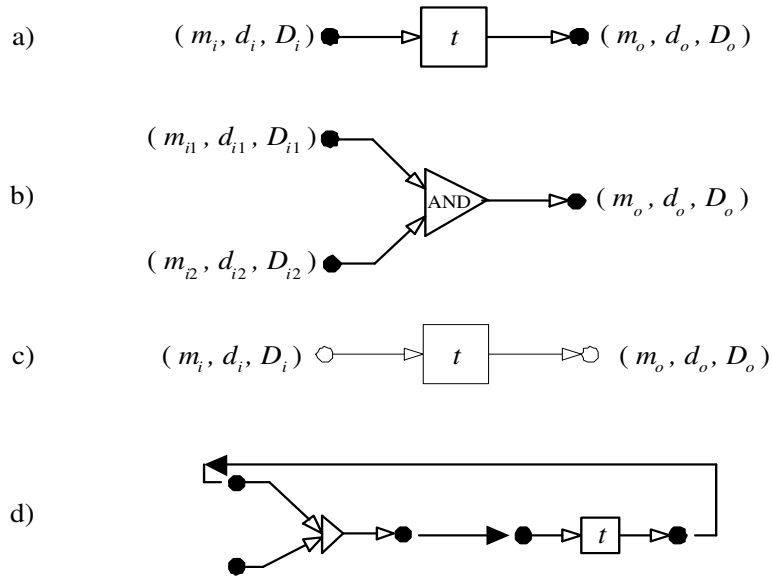
a) $(m_i, d_i, D_i) \bullet \longrightarrow \boxed{t} \longrightarrow \!\!\triangleright\!\bullet\; (m_o, d_o, D_o)$

b) $(m_{i1}, d_{i1}, D_{i1}) \bullet$, $(m_{i2}, d_{i2}, D_{i2}) \bullet$ AND $\longrightarrow \!\!\triangleright\!\bullet\; (m_o, d_o, D_o)$

c) $(m_i, d_i, D_i) \circ \longrightarrow \boxed{t} \longrightarrow \!\!\triangleright\!\circ\; (m_o, d_o, D_o)$

d)

Figure 4.17. Simple graph components with port event models

output assertions of the assume-guarantee type only establish the delay between outputs and inputs, and do not enforce exact time intervals, i.e., exact period instance of the intervals. In such specifications the first event of a stream can occur in any period instance m. To depict this kind of port dependency in figures we use empty circles, as shown in Fig. 4.17 c). In particular, for each $n \in \mathbb{N}_0$ if the first input event occurs in the interval $[(n + m_i) \cdot p + d_i, (n + m_i) \cdot p + D_i]$ then the first output event is generated in the interval $[(n + m_o) \cdot p + d_o, (n + m_o) \cdot p + D_o]$. In this case port values m determine only the period latency between the corresponding event occurrences on different ports. That is why, in our interface definition, we distinguish between two types of ports, locked - which makes occurrence time specific to a single period (Fig. 4.17 a), and unlocked - which allows for the occurrence in any period instance (Fig. 4.17 c). For the unlocked ports, the interface does not specify exact port time intervals, but only the time difference between ports. Note that all ports that are connected irrespective of the direction of dependencies have the same port type. Fig. 4.17 d) shows how two connection operations represented with full arrows

can be applied on simple graphs from Fig. 4.17 a) and b) to generate a functional cycle with a single external input.

To make the presentation simpler, we concentrate on the composition of event models, and not on the resource models. Although this is not a constraint of the theory, we present it as there is no resource sharing. So, we assume that allocation of tasks to processing elements and resource scheduling policy is given, such that bounds on task response time can be computed. For instance, if each task has its own dedicated resource, bounds on response time equal bounds on execution time. In [78] a procedure for tight computation of such intervals is given in the general case. Along the lines of the informal discussion given in this section, we next formally define the interface for task graphs.

4.5.2   Interface

Let $p \in \mathbb{R}_{>0}$ be the constant *period* for all interfaces.

An *interface* $F = (G_F, m_F, d_F, D_F, l_F)$ consists of:

- A directed graph $G_F = (P_F, \rho_F)$, that consists of a set of *ports* P_F and a set of direct port *dependencies* $\rho_F \subseteq P_F^2$. Given G_F, let the set of *input ports* I_F be the set of all ports without predecessors, i.e., $I_F = \{i \in P_F \mid \forall x \in P_F \ . \ (x, i) \notin \rho_F\}$. Let the set of *output ports* O_F contain the remaining ports, i.e., $O_F = P_F \setminus I_F$.

- A period *instance* function $m_F : P_F \to \mathbb{Z}$.

- A *lower*-bound function $d_F : P_F \to \mathbb{R}$ such that for all $x \in P_F$ it holds $0 \leq d_F(x) < p$. An *upper*-bound function $D_F : P_F \to \mathbb{R}$ such that for all $x \in P_F$ it holds $d_F(x) \leq D_F(x) < d_F(x) + p$.

- A *lock* function $l_F : P_F \to \mathbb{B}$. If $l_F(x) = 1$ for a port $x \in P_F$ then for each $n \in \mathbb{N}_0$ the period instance of the $n-$th event $x(n)$ on port x is given with $m_F(x) +$

n. We assume that for each two ports $x_1, x_2 \in P_F$, if the two ports belong to the same weakly connected component of G_F, then $l_F(x_1) = l_F(x_2)$ (a directed graph is weakly connected if it would be fully connected by ignoring the direction of edges).

The condition $D_F(x) < d_F(x) + p$ is introduced to make the event jitter $D_F(x) - d_F(x)$ smaller than p. This means there are no event bursts, i.e., no simultaneous occurrence of events, on port x. This constraint is introduced only to make the presentation simpler, most results hold even if it is not satisfied. According to the definition above if there exists $x \in P_F$ such that $l_F(x) = 1$, i.e., if port x is locked, then we have $l_F(x') = 1$ for all $x' \in P$ where P is a weakly connected component containing x. In that case we also write $l_F(P) = 1$ and say the component P is locked. The simplest locked and unlocked components are shown in Fig. 4.17 a) and c) respectively. To avoid a formal introduction of initial events of a cycle, in following sections we also assume that for each cycle of interface graph G_F there exists a port $x \in P_F$ of the cycle and an input port $i \in I_F$ such that $(i, x) \in \rho_F$.

Interface Semantics. We assume that the event sequence on each port is indexed starting from the event index zero. Let $\mathcal{S} = [\mathbb{N}_0 \to \mathbb{R}]$ be the set of all infinite sequences of real numbers. Each event occurrence is bounded in time by an interval specified with a triple of numbers. Given numbers $m \in \mathbb{Z}$, $d \in \mathbb{R}$ and $D \in \mathbb{R}$, let $\mathcal{S}(m, d, D)$ be the set of sequences $\mathcal{S}(m, d, D) = \{s \in \mathcal{S} \mid \forall n \in \mathbb{N}_0 . (n + m) \cdot p + d \leq s(n) \leq (n + m) \cdot p + D\}$. Finally, let Π_F be the weakly connected component partition of G_F, i.e., the set of all weakly connected components of G_F.

The semantics $\mathcal{S}_F$ of an interface F is a set of signals $s : P_F \to \mathcal{S}$, i.e., a set of tuples of event sequences, that satisfy the constraints of F. Formally, given $F = (G_F, m_F, d_F, D_F, l_F)$, $s \in \mathcal{S}_F$ iff for each weakly connected component $P \in \Pi_F$ we have

- if $l_F(P) = 1$ (P is locked), then for each port $x \in P$ it holds $s(x) \in \mathcal{S}(m_F(x), d_F(x), D_F(x))$, and

- if $l_F(P) = 0$ (P is unlocked), then there exists a number $m \in \mathbb{Z}$ such that for each port $x \in P$ it holds $s(x) \in \mathcal{S}(m + m_F(x), d_F(x), D_F(x))$.

According to this definition, for a locked port $x \in P_F$ ($l_F(x) = 1$) and for each $n \in \mathbb{N}_0$, the timing uncertainty of the $n-$th event $x(n)$ on port x is given with the interval $[(n + m_F(x)) \cdot p + d_F(x), (n + m_F(x)) \cdot p + D_F(x)]$. The difference in the two cases of the previous definition is in the additional existential quantifier in the unlocked case that enables the exact period instance of a port not to be specified. However, the value of m has to be the same for all ports x in the component P. The uncertainty in the locked case comes only in the form of a single interval, whereas in the locked case there are infinitely many such intervals.

The semantics $\mathcal{S}_F$ determines P_F, d_F, D_F, and l_F, but not ρ_F and not m_F. The function m_F is determined for all locked ports, whereas for unlocked ports only the differences in the function values between ports in the same weakly connected components. Therefore, we have the following lemma:

Lemma 4 *Let $F = (G_F, m_F, d_F, D_F, l_F)$ and $E = (G_E, m_E, d_E, D_E, l_E)$ be two interfaces. If $\Pi_F = \Pi_E$ (and consequently $P_F = P_E$), $d_F = d_E$, $D_F = D_E$, $l_F = l_E$, for each weakly connected component $P \in \Pi_F$ and each two ports $x_1, x_2 \in P$ it holds $m_F(x_1) - m_F(x_2) = m_E(x_1) - m_E(x_2)$, and for each locked port $x \in P_F$ it holds $m_F(x) = m_E(x)$, then the two interfaces have the same semantics, $\mathcal{S}_F = \mathcal{S}_E$.*

In the rest of the section we write $f_{|X}$ for the restriction of the function f on a domain set X. Given $P \subseteq P_F$ let $\mathcal{S}_{F|P}$ be the set of sequences from $\mathcal{S}$ restricted to ports in P, i.e., $\mathcal{S}_{F|P} = \{s' : P \to \mathcal{S} \mid \exists s \in \mathcal{S}_F . s' = s_{|P}\}$.

4.5.3 Interface Algebra

The algebra contains three operations and a relation. Two operations, *composition* and *connection*, and the *refinement* relation are analogous to the task sequences case. The third operation, the *join* operation, is related to tasks with multiple inputs, i.e., AND task triggering.

Composition. The binary composition operation $\|$ on two interfaces F and E puts together their ports without making new dependencies. The operation is illustrated in Fig. 4.18.
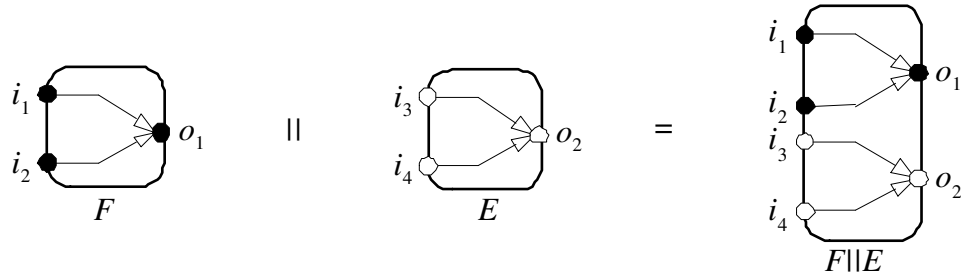
Figure 4.18. Interface composition operation for graphs

Let $F = (G_F, m_F, d_F, D_F, l_F)$ and $E = (G_E, m_E, d_E, D_E, l_E)$. Interface $F' = F\|E$ is defined if $P_F \cap P_E = \emptyset$. In that case $F' = (G_{F'}, m_{F'}, d_{F'}, D_{F'}, l_{F'})$ where

- $G_{F'} = (P_{F'}, \rho_{F'})$ such that $P_{F'} = P_F \cup P_E$ and $\rho_{F'} = \rho_F \cup \rho_E$,

- $m_{F'|P_F} = m_F$, and $m_{F'|P_E} = m_E$,

- $d_{F'|P_F} = d_F$, and $d_{F'|P_E} = d_E$,

- $D_{F'|P_F} = D_F$, and $D_{F'|P_E} = D_E$, and

- $l_{F'|P_F} = l_F$, and $l_{F'|P_E} = l_E$.

Connection. The unary connection operation $\rightarrow$ connects an output port of an interface to an input port of the same interface, by taking out the input port from the set of ports and

establishing dependencies from the output port to the input port successors. An instance of the connection operation is shown in Fig. 4.19.
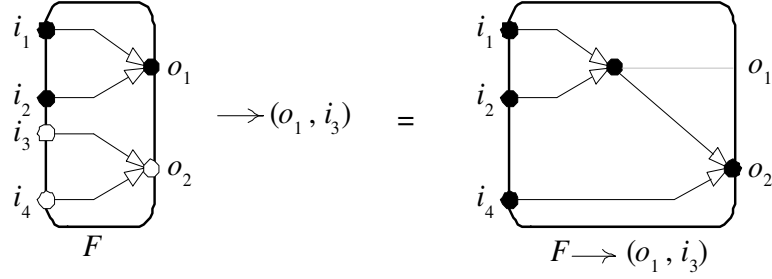

Figure 4.19. Interface connection operation for graphs

To simplify notation, for two ports $x_1, x_2 \in P_F$ we write $path_F(x_1, x_2) = 1$ if in G_F there exists a directed path from x_1 to x_2, and $path_F(x_1, x_2) = 0$ otherwise. Also, for a port $x \in P_F$ let $P_F(x) \in \Pi_F$ be the weakly connected component G_F containing port x. Finally, let $Succ_F(x)$ be the set of all direct successors of x in G_F.

Let $F = (G_F, m_F, d_F, D_F, l_F)$, $i_1 \in I_F$, and $o_1 \in O_F$. Interface $F' = F \rightarrow (o_1, i_1)$ is defined if the following conditions are satisfied

1. $d_F(o_1) \geq d_F(i_1)$,

2. $D_F(o_1) \leq D_F(i_1)$,

3. if $path_F(i_1, o_1) = 0$ and a) $l_F(i_1) = l_F(o_1) = 1$ (both ports locked) or b) $P_F(i_1) = P_F(o_1)$ (both ports in the same connected component), then $m_F(i_1) = m_F(o_1)$, and

4. if $path_F(i_1, o_1) = 1$, then $m_F(i_1) \leq m_F(o_1)$.

According to condition 3a) if the two ports are locked, the connection operation requires they have the same period instance values m_F. Conditions 3b) and 4 cover the cases in which i_1 and o_1 are contained in the same connected component. The condition 4 addresses the case when the connection operation introduces a new cycle in G_F because, by definition, $i_1 \in I_F$ does not have a direct predecessor in G_F.

121

If the connection operation is defined, we have $F' = (G_{F'}, m_{F'}, d_{F'}, D_{F'}, l_{F'}) = F \to (o_1, i_1)$ where

- $G_{F'} = (P_{F'}, \rho_{F'})$ such that a) if $path_F(i_1, o_1) = 1$ then $P_{F'} = P_F$ and $\rho_{F'} = \rho_F \cup (\{o_1\} \times Succ_F(i_1))$, and b) if $path_F(i_1, o_1) = 0$ then $P_{F'} = P_F \setminus \{i_1\}$ and $\rho_{F'} = \rho_F \setminus (\{i_1\} \times Succ_F(i_1)) \cup (\{o_1\} \times Succ_F(i_1))$,

- a) if $l_F(i_1) = l_F(o_1) = 1$ or $P_F(i_1) = P_F(o_1)$ then $m_{F'} = m_{F|P_{F'}}$, b) if $l_F(i_1) = 0$ and $P_F(i_1) \neq P_F(o_1)$ then $m_{F'}(x) = m_F(x) + (m_F(o_1) - m_F(i_1))$ for each port $x \in P_{F'} \cap P_F(i_1)$, and $m_{F'}(x) = m_F(x)$ for each port $x \in P_{F'} \setminus P_F(i_1)$, and c) if $l_F(i_1) = 1$, $l_F(o_1) = 0$ and $P_F(i_1) \neq P_F(o_1)$ then $m_{F'}(x) = m_F(x) + (m_F(i_1) - m_F(o_1))$ for each port $x \in P_{F'} \cap P_F(o_1)$, and $m_{F'}(x) = m_F(x)$ for each port $x \in P_{F'} \setminus P_F(o_1)$,

- $d_{F'} = d_{F|P_{F'}}$,

- $D_{F'} = D_{F|P_{F'}}$, and

- a) if $l_F(i_1) = 1$ or $l_F(o_1) = 1$ then $l_{F'}(x) = 1$ for each port $x \in P_{F'} \cap (P_F(i_1) \cup P_F(o_1))$, and $l_{F'}(x) = l_F(x)$ for each port $x \in P_{F'} \setminus (P_F(i_1) \cup P_F(o_1))$, and b) if $l_F(i_1) = l_F(o_1) = 0$ then $l_{F'}(x) = l_F(x)$ for each port $x \in P_{F'}$.

According to this definition, only in case when the connection operation closes a cycle the input port i_1 remains in the set of ports of F' with its edges, but with new dependencies between the output port o_1 and successors of i_1 included. In all other cases the edges from i_1 to its successors are substituted with edges from o_1 with i_1 removed. By our initial assumption, for each cycle of interface graph G_F there exists a port $x \in P_F$ of the cycle and an input port $i \in I_F$ such that $(i, x) \in \rho_F$. Taking into account the connection operation definition we have that this property is preserved under the connection operation. Consequently, for each output port $o \in O_F$ there exists an input port such $i \in I_F$ such that $path_F(i, o) = 1$. In fact, this property is preserved under all algebra operations.

From the above definition it also follows that when the operation is applied between two weakly connected components $P_F(i_1)$ and $P_F(o_1)$, the period instance function $m_{F'}$ values are translated to the values of the locked component, or to the values of the component $P_F(o_1)$ if both components are unlocked. The values of functions $d_{F'}$ and $D_{F'}$ are the same as d_F and D_F, respectively. If any of the two components is locked all ports in these components will be locked after the connection operation, and if both components are unlocked all ports in these components will be unlocked.

Join. The unary join operation $\succ$ connects two output ports of an interface, matches the corresponding event streams for the AND type of triggering, and adds the matched output to the set of ports. To simplify the presentation the definition given here is limited to only two output ports, but this can be generalized. The operation is illustrated in Fig. 4.20.
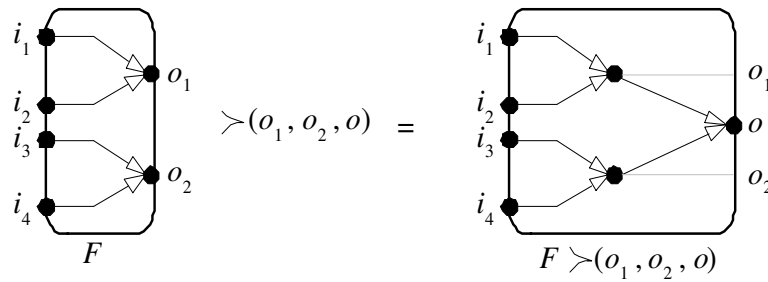

Figure 4.20. Interface join operation for graphs

Let $F = (G_F, m_F, d_F, D_F, l_F)$ and $o_1, o_2 \in O_F$. $F' = F\succ(o_1, o_2, o)$ is defined if $o \notin P_F$ and if $l_F(o_1) = l_F(o_2) = 1$. In that case $F' = (G_{F'}, m_{F'}, d_{F'}, D_{F'}, l_{F'})$ where

- $G_{F'} = (P_{F'}, \rho_{F'})$ such that $P_{F'} = P_F \cup \{o\}$ and $\rho_{F'} = \rho_F \cup \{(o_1, o), (o_2, o)\}$,

- $m_{F'}(o) = \texttt{max}\{m_F(o_1), m_F(o_2)\}$, and for each port $x \in P_F$, $m_{F'}(x) = m_F(x)$,

- $d_{F'}(o) = \texttt{max}\{m_F(o_1) \cdot p + d_F(o_1), m_F(o_2) \cdot p + d_F(o_2)\} - m_{F'}(o) \cdot p$, and for each port $x \in P_F$, $d_{F'}(x) = d_F(x)$,

123

- $D_{F'}(o) = \mathtt{max}\{m_F(o_1) \cdot p + D_F(o_1), m_F(o_2) \cdot p + D_F(o_2)\} - m_{F'}(o) \cdot p$, and for each port $x \in P_F$, $D_{F'}(x) = D_F(x)$, and

- $l_{F'}(o) = 1$, and for each port $x \in P_F$, $l_{F'}(x) = l_F(x)$.

As it will be cleared from the proposition below, the values of functions $m_{F'}$, $d_{F'}$ and $D_{F'}$ for port o are defined to respect the AND type of triggering. For instance, the $\mathtt{max}$ function is due to the fact that an event on port o occurs with the later of the corresponding events on ports o_1 and o_2. Note also that join operation is defined only if both output ports are locked. If this is not the case, i.e., if the period instance of one of the output ports to perform the join operation on is not locked, there would be infinitely many uncertainty intervals for the matched output which requires more complicated form of the interface and operations.

The following proposition justifies the definition of operations, by relating semantics of interfaces before and after operations. Note that operators given in the statement of the proposition assume pointwise function operations. For instance, $s'(o_1) = s(i_1)$ means for each $n \in \mathbb{N}_0$, $s'(o_1)(n) = s(i_1)(n)$.

Proposition 14 (Operation Semantics) *For all interfaces F, E*

1. *If $F' = F \| E$ is defined then*
   $$\mathcal{S}_{F'} = \{s' : P_{F'} \to \mathcal{S} \mid \exists s_F \in \mathcal{S}_F . \exists s_E \in \mathcal{S}_E . s'_{|P_F} = s_F \text{ and } s'_{|P_E} = s_E\},$$

2. *If $F' = F \to (o_1, i_1)$ is defined then*
   $$\mathcal{S}_{F'} = \{s' : P_{F'} \to \mathcal{S} \mid \exists s \in \mathcal{S}_F . s' = s_{|P_{F'}} \text{ and } (path_F(i_1, o_1) = 0 \implies s(o_1) = s(i_1))\}, \text{ and}$$

3. *If $F' = F \succ (o_1, o_2, o)$ is defined then*
   $$\mathcal{S}_{F'} =$$
   $$\{s' : P_{F'} \to \mathcal{S} \mid \exists s_1 \in \mathcal{S}_F . \exists s_2 \in \mathcal{S}_F . s'_{|P_F} = s_1 \wedge s'(o) = \mathtt{max}\{s_2(o_1), s_2(o_2)\}\}.$$

Proof.

1. The composition operation $F\|E$ does not modify the two interface graphs G_F and G_E, and does not introduce dependencies between the two graphs. Also functions m, d, D and l remain the same for each port of the composition. So, the semantics of $F\|E$ is the product of semantics of F and E.

2. Let $l_F(i_1) = l_F(o_1) = 1$ or $P_F(i_1) = P_F(o_1)$. In this case, according to the definition of the connection operation, all elements of interface F' are the restriction on $P_{F'}$ of the corresponding elements of F, which also means $\mathcal{S}_{F'} = \mathcal{S}_{F|P_{F'}}$. Assume first $path_F(i_1, o_1) = 1$ that satisfies condition $P_F(i_1) = P_F(o_1)$, i.e., connection operation $\rightarrow(o_1, i_1)$ generates a new cycle in $G_{F'}$. Since $P_{F'} = P_F$ we have $\mathcal{S}_{F'} = \mathcal{S}_F$ which is what the proposition states for this case. Assume $path_F(i_1, o_1) = 0$, i.e., no cycle is introduced by the connection operation. Since, by the connection condition, $d_F(o_1) \geq d_F(i_1)$ and $D_F(o_1) \leq D_F(i_1)$, i.e., $[d_F(o_1), D_F(o_1)] \subseteq [d_F(i_1), D_F(i_1)]$, we have that for each $s' \in \mathcal{S}_{F|P_{F'}}$ there exists $s \in \mathcal{S}_F$ such that $s' = s_{|P_{F'}}$ and $s(o_1) = s(i_1)$. Also, for each $s \in \mathcal{S}_F$ such that $s(o_1) = s(i_1)$ we have $s_{|P_{F'}} \in \mathcal{S}_{F|P_{F'}}$. If $l_F(i_1) = 0$ and $l_F(o_1) = 1$ and $P_F(i_1) \neq P_F(o_1)$, we have that $s(o_1) = s(i_1)$ holds only if $m_{F'}(i_1)$ would be equal to $m_F(o_1)$. This is reflected in $\mathcal{S}_{F'}$ due to the modifications of $m_{F'}$ and $l_{F'}$ functions on the ports in $P_F(i_1)$. The case $l_F(i_1) = 1$ and $l_F(o_1) = 0$ is treated analogously. Finally, if $l_F(i_1) = l_F(o_1) = 0$ and $P_F(i_1) \neq P_F(o_1)$, we have that $s(o_1) = s(i_1)$ is satisfied for all integer values of $m_{F'}(i_1) = m_F(o_1)$. The proposition is true even in this case since all the ports in $P_F(i_1)$ and $P_F(o_1)$ remain unlocked.

3. A necessary condition for $\succ(o_1, o_2, o)$ operation to be applied on interface F is $l_F(o_1) = l_F(o_2) = 1$, i.e., o_1 and o_2 and respective weakly connected components are locked. So, the fact that o_1 and o_2 are in the same connected component of $G_{F\succ(o_1,o_2,o)}$ is not an additional restriction on semantics of $\mathcal{S}_{F'} = \mathcal{S}_{F\succ(o_1,o_2,o)}$. That is why $s' \in \mathcal{S}_{F'}$ iff there exists $s_1 \in \mathcal{S}_F$ such that $s'_{|P_F} = s_1$. The

only difference in elements of F and $F{\succ}(o_1, o_2, o)$ is in port o. For instance, if $m_F(o_1) = m_F(o_2)$ then according to the definition of join operation we have $[d_{F'}(o), D_{F'}(o)] = [\max\{d_F(o_1), d_F(o_2)\}, \max\{D_F(o_1), D_F(o_2)\}]$. This is exactly the interval of possible outcomes of $\max\{s_2(o_1), s_2(o_2)\}$ for all $s_2 \in \mathcal{S}_F$. Similar argument holds if $m_F(o_1) \neq m_F(o_2)$.

$\square$

The idea behind the refinement relation between two interfaces is again to be able to substitute a more refined interface for a more abstract interface. The requirements in the following definition are analogous to those given for the task sequence interfaces except for the third requirement. The first and fourth requirement allow for weaker input assumptions in the refined interface, whereas the second and fifth requirement demand stronger output guarantees for the refined interface. As shown in a proposition of the next subsection, in case of general graphs the third requirement is necessary for the independent refinement property.

Refinement. Let $F = (G_F, m_F, d_F, D_F, l_F)$ and $F' = (G_{F'}, m_{F'}, d_{F'}, D_{F'}, l_{F'})$.

F' refines F, i.e., $F' \preceq F$ if and only if

1. $I_{F'} \subseteq I_F$,

2. $O_{F'} \supseteq O_F$,

3. $\rho_{F'} \subseteq \rho_F$,

4. $\mathcal{S}_{F'|I_{F'}} \supseteq \mathcal{S}_{F|I_{F'}}$, and

5. for each $s \in \mathcal{S}_{F'}$, if $s_{|I_{F'}} \in \mathcal{S}_{F|I_{F'}}$ then $s_{|O_F} \in \mathcal{S}_{F|O_F}$.

For instance, as shown in Fig. 4.21, if $G_F = G_{F'}$, $m_F = m_{F'}$, $l_F = l_{F'}$, for each input port $i \in I_{F'}$ it holds $d_F(i) \geq d_{F'}(i)$ and $D_F(i) \leq D_{F'}(i)$, and for each output port $o \in O_F$
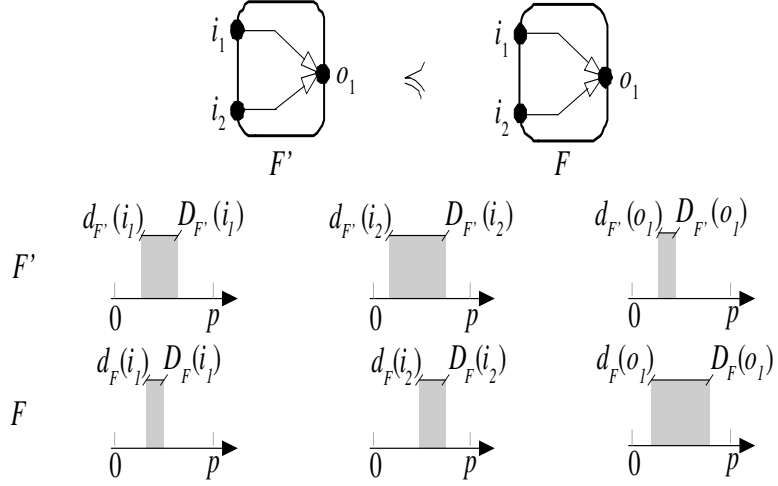
126

Figure 4.21. Interface refinement relation for graphs

it holds $d_F(o) \leq d_{F'}(o)$ and $D_F(o) \geq D_{F'}(o)$, then $F' \preceq F$. Note that the form of the fifth requirement of the refinement relation definition is not simply $\mathcal{S}_{F'|O_F} \subseteq \mathcal{S}_{F|O_F}$. This is to allow a graph with unlocked ports be a refinement of the same graph with locked ports and all other elements the same. In particular, if $G_F = G_{F'}$, $m_F = m_{F'}$, $d_F = d_{F'}$, $D_F = D_{F'}$, but $l_F(x) = 1$ and $l_{F'}(x) = 0$ for each port $x \in P_F$, then $F' \preceq F$.

4.5.4 Interface Algebra Properties

Ideally, for incremental design for any two operations op_1 and op_2 of the algebra the order of the operations does not matter, i.e., well defined $F \circ op_1 \circ op_2$ implies well defined $F \circ op_2 \circ op_1$, and $F \circ op_1 \circ op_2 = F \circ op_2 \circ op_1$. However, with this algebra we have weaker propositions. For some cases, only if both $F \circ op_1 \circ op_2$ and $F \circ op_2 \circ op_1$ are defined, we have $F \circ op_1 \circ op_2 = F \circ op_2 \circ op_1$. The following proposition makes clear the character of associativity property in each possible case, with cases 2 and 3 being the most relevant.

Proposition 15 (Incremental Design) *Let $i_1, i_2 \in I_F$ and $o_1, o_2, o_3, o_4 \in O_F$. For all interfaces F, E, and H,*

1. *(a) If $(F\|E)\|H$ is defined then $F\|(E\|H)$ is defined, and $(F\|E)\|H = F\|(E\|H)$.*

   (b) If $(F\|E)\to(o_1, i_1)$ is defined then $(F\to(o_1, i_1))\|E$ is defined, and $(F\|E)\to(o_1, i_1) = (F\to(o_1, i_1))\|E$.

   (c) If $(F\succ(o_1, o_2, o))\|E$ is defined then $(F\|E)\succ(o_1, o_2, o)$ is defined, and $(F\succ(o_1, o_2, o))\|E = (F\|E)\succ(o_1, o_2, o)$.

   (d) If $(F\|E)\succ(o_1, o_2, o)$ is defined then $(F\succ(o_1, o_2, o))\|E$ is defined, and $(F\|E)\succ(o_1, o_2, o) = (F\succ(o_1, o_2, o))\|E$.

2. *(a) If $F_1 = (F\to(o_1, i_1))\to(o_2, i_2)$ and $F_2 = (F\to(o_2, i_2))\to(o_1, i_1)$ are defined, then $\mathcal{S}_{F_1|P_F\setminus\{i_1, i_2\}} = \mathcal{S}_{F_2|P_F\setminus\{i_1, i_2\}}$.*

   (b) If $(F\succ(o_2, o_3, o))\to(o_1, i_1)$ is defined then $(F\to(o_1, i_1))\succ(o_2, o_3, o)$ is defined, and $(F\succ(o_2, o_3, o))\to(o_1, i_1) = (F\to(o_1, i_1))\succ(o_2, o_3, o)$.

3. *(a) If $(F\succ(o_1, o_2, o))\succ(o_3, o_4, o')$ is defined then $(F\succ(o_3, o_4, o'))\succ(o_1, o_2, o)$ is defined, and $(F\succ(o_1, o_2, o))\succ(o_3, o_4, o') = (F\succ(o_3, o_4, o'))\succ(o_1, o_2, o)$.*

   (b) If $F_1 = (F\succ(o_1, o_2, o))\succ(o, o_3, o')$ is defined then $F_2 = (F\succ(o_1, o_3, o))\succ(o, o_2, o')$ is defined, and $\mathcal{S}_{F_1|P_{F_1}\setminus\{o\}} = \mathcal{S}_{F_2|P_{F_2}\setminus\{o\}}$.

Proof.

1. (a) Both interfaces are defined if any two port sets of P_F, P_E and P_H have no common elements. The composition operation does not modify any element of individual component interfaces.

   (b) If $(F\|E)\to(o_1, i_1)$ is defined then $(F\to(o_1, i_1))\|E$ is defined because i_1 and o_1 are elements of P_F but not of P_E since $F\|E$ is defined. The composition

with E does not make changes in elements of F, so the operation order does not matter.

Note that if $(F{\rightarrow}(o_1, i_1))\|E$ is defined then it does not mean $(F\|E){\rightarrow}(o_1, i_1)$ is defined. For instance, consider $i_1 \in P_E$ and $i_1 \notin P_{F\rightarrow(o_1,i_1)}$.

(c) If $(F{\succ}(o_1, o_2, o))\|E$ is defined then $o \notin P_F$ and $o \notin P_E$ since $o \in F{\succ}(o_1, o_2, o)$. Therefore, $o \notin P_{F\|E}$. The ${\succ}(o_1, o_2, o)$ operation involves only F and since the composition operation does not change component graphs and other interface elements, the order does not matter.

(d) Follows similar to previous point since $o_1, o_2 \in P_F$ by proposition statement.

2. (a) Since the input port of a connection operation is taken out of the set of ports the two connection operations of the proposition are defined only if $i_1 \neq i_2$. Note that if F_1 is defined then F_2 is not necessarily defined. For instance, when two chains $path(i_1, o_2)$ and $path(i_2, o_1)$ of locked ports are connected into a cycle by the two successive operations such that $m(i_1) < m(o_1)$ the operation ${\rightarrow}(o_1, i_1)$ is possible only after the path from i_1 to o_1 is established through the operation ${\rightarrow}(o_2, i_2)$.

We first assume no cycle is created by the operations and use Lemma 4. By the connection operation definition we have $I_{F_1} = I_F \setminus \{i_1, i_2\} = I_{F_2}$ and $O_{F_1} = O_F = O_{F_2}$. Since a connection operation does not modify set of successors $Succ_F(i)$ of an input port i, we have $\rho_{F_1} = \rho_F \setminus (\{i_1\} \times Succ_F(i_1)) \setminus (\{i_2\} \times Succ_F(i_2)) \cup (\{o_1\} \times Succ_F(i_1)) \cup (\{o_2\} \times Succ_F(i_2)) = \rho_{F_2}$. Consequently, the two interface graphs are equal, $G_{F_1} = G_{F_2}$. The d and D functions are not modified by connection operation: $d_{F_1} = d_{F|P_{F_1}} = d_{F|P_{F_2}} = d_{F_2}$ and $D_{F_1} = D_{F|P_{F_1}} = D_{F|P_{F_2}} = D_{F_2}$. The function l_{F_1} (resp. l_{F_2}) is determined by the function l_F and by the partition set Π_{F_1} (resp. Π_{F_2}). Since $G_{F_1} = G_{F_2}$, i.e., $\Pi_{F_1} = \Pi_{F_2}$ we have $l_{F_1} = l_{F_2}$. Let $P \in \Pi_{F_1} = \Pi_{F_2}$, $x \in P$ and

$l_{F_1}(x) = l_{F_2}(x) = 1$. If also $l_F(x) = 1$ then no connection operation modifies period instance function for x, i.e., $m_{F_1}(x) = m_F(x) = m_{F_2}(x)$. If $l_F(x) = 0$ then the value of l_F for at least one of the ports i_1, o_1, i_2 or o_2 is 1, and thus the value of m_F of that port uniquely determines both $m_{F_1}(x)$ and $m_{F_2}(x)$, i.e., $m_{F_1}(x) = m_{F_2}(x)$. Let $P \in \Pi_{F_1} = \Pi_{F_2}$ and $x_1, x_2 \in P$ such that $l_{F_1}(x_1) = l_{F_2}(x_1) = 0$ and $l_{F_1}(x_2) = l_{F_2}(x_2) = 0$. Since x_1 and x_2 are elements of the same weakly connected component P there is an undirected path from x_1 to x_2, both in G_{F_1} and G_{F_2}, that may also contain o_1 and o_2. For instance, the path contains o_1 and o_2 if there exists no weakly connected component of G_F that contains both x_1 and x_2. Although it is not necessarily $m_{F_1}(x_1) = m_{F_2}(x_1)$ or $m_{F_1}(x_2) = m_{F_2}(x_2)$, we have $m_{F_1}(x_1) - m_{F_1}(x_2) = m_{F_2}(x_1) - m_{F_2}(x_2)$, because no connection operation modifies difference between m values of already connected ports. For instance, if in G_F there exists a path from x_1 to o_1 then $m_{F_1}(x_1) - m_{F_1}(o_1) = m_F(x_1) - m_F(o_1) = m_{F_2}(x_1) - m_{F_2}(o_1)$. From Lemma 4 follows $\mathcal{S}_{F_1} = \mathcal{S}_{F_2}$, i.e., $\mathcal{S}_{F_1|P_F \setminus \{i_1, i_2\}} = \mathcal{S}_{F_2|P_F \setminus \{i_1, i_2\}}$.

If there exists a path from i_2 to o_2 in G_F the connection operation $\rightarrow(o_1, i_1)$ does not break it. Therefore, if $path_F(i_2, o_2) = 1$ (resp. $path_F(i_1, o_1) = 1$) then $path_{F \rightarrow (o_1, i_1)}(i_2, o_2) = 1$ (resp. $path_{F \rightarrow (o_2, i_2)}(i_1, o_1) = 1$). If $path_F(i_2, o_2) = 0$ but $path_{F \rightarrow (o_1, i_1)}(i_2, o_2) = 1$ then it means $path_F(i_2, o_1) = 1$ and $path_F(i_1, o_2) = 1$. Consequently, in this case we also have $path_{F \rightarrow (o_2, i_2)}(i_1, o_1) = 1$. Thus, in all cases we have that a cycle exists in F_1 iff it exists in F_2. The two interface graphs are G_{F_1} and G_{F_2} may differ only in input ports i_1 and i_2 and edges coming from them. Since the entire argument given above can be repeated for ports other then i_1 and i_2, we have $\mathcal{S}_{F_1|P_F \setminus \{i_1, i_2\}} = \mathcal{S}_{F_2|P_F \setminus \{i_1, i_2\}}$.

(b) If $F \succ (o_2, o_3, o)$ is defined then $o \notin P_F$. Therefore, since $o_1 \in P_F$ by proposition statement, $o_1 \neq o$. The elements of interfaces $F \succ (o_2, o_3, o)$

and F are equal for all ports of P_F. Thus conditions needed for the connection operation $\rightarrow(o_1, i_1)$ are the same for $F\succ(o_2, o_3, o)$ and F. In addition, graphs $G_{(F\succ(o_2,o_3,o))\rightarrow(o_1,i_1)}$ and $G_{(F\rightarrow(o_1,i_1))\succ(o_2,o_3,o)}$ are equal. The same is true for functions m, d, D, and l. For instance, for all $x \in P_F \cup \{o\}$ it is $l_{(F\succ(o_2,o_3,o))\rightarrow(o_1,i_1)}(x) = l_{(F\rightarrow(o_1,i_1))\succ(o_2,o_3,o)}(x)$ since $l_F(o_2) = l_F(o_3) = 1$ by the definition of join operation.

Note that if $(F\rightarrow(o_1, i_1))\succ(o_2, o_3, o)$ is defined then $(F\succ(o_2, o_3, o))\rightarrow(o_1, i_1)$ is not necessarily defined. For instance, it can be $l_F(o_2) = 0$ and $l_{F\rightarrow(o_1,i_1)}(o_2) = 1$, but $(F\succ(o_2, o_3, o))\rightarrow(o_1, i_1)$ is not defined.

3. (a) If $(F\succ(o_1, o_2, o))\succ(o_3, o_4, o')$ is defined then $o \notin \{o_3, o_4\}$ since $o \notin P_F$ and $o_3, o_4 \in P_F$ by proposition statement. Similarly, we have $\{o, o'\} \cap \{o_1, o_2, o_3, o_4\} = \emptyset$. In addition, $o' \neq o$ and thus $(F\succ(o_3, o_4, o'))\succ(o_1, o_2, o)$ is defined. The order does not matter because each join operation just modifies the parameters of either o or o'.

(b) If $(F\succ(o_1, o_2, o))\succ(o, o_3, o')$ is defined then $\{o, o'\} \cap \{o_1, o_2, o_3\} = \emptyset$. Note that $F_1 \neq F_2$ since $G_{F_1} \neq G_{F_2}$. In general, $\mathcal{S}_{F_1} \neq \mathcal{S}_{F_2}$ because, for instance, $d_{F_1}(o) \neq d_{F_2}(o)$. However, $\mathcal{S}_{F_1|P_{F_1}\setminus\{o\}} = \mathcal{S}_{F_2|P_{F_2}\setminus\{o\}}$ due to the associativity of max function. For instance, $d_{F_1}(o') = \max\{m_F(o) \cdot p + d_F(o), m_F(o_3) \cdot p + d_F(o_3)\} - \max\{m_F(o), m_F(o_3)\} \cdot p=$

$\max\{\max\{m_F(o_1), m_F(o_2)\} \cdot p + \max\{m_F(o_1) \cdot p + d_F(o_1), m_F(o_2) \cdot p + d_F(o_2)\} - \max\{m_F(o_1), m_F(o_2)\} \cdot p, m_F(o_3) \cdot p + d_F(o_3)\} - \max\{\max\{m_F(o_1), m_F(o_2)\}, m_F(o_3)\} \cdot p=$

$\max\{m_F(o_1) \cdot p + d_F(o_1), m_F(o_2) \cdot p + d_F(o_2), m_F(o_3) \cdot p + d_F(o_3)\} - \max\{m_F(o_1), m_F(o_2), m_F(o_3)\} \cdot p = d_{F_2}(o')$.

$\square$

The following lemma, similar in form to Lemma 4, gives refinement sufficient condition entirely based on interface F and F' elements.

Lemma 5 *Let $F = (G_F, m_F, d_F, D_F, l_F)$ and $F' = (G_{F'}, m_{F'}, d_{F'}, D_{F'}, l_{F'})$ be two interfaces such that $I_{F'} \subseteq I_F$, $O_{F'} \supseteq O_F$ and $\rho_{F'} \subseteq \rho_F$. F' refines F, i.e., $F' \preceq F$ iff for each input port $i \in I_{F'}$ it holds $d_F(i) \geq d_{F'}(i)$ and $D_F(i) \leq D_{F'}(i)$, for each output port $o \in O_F$ it holds $d_F(o) \leq d_{F'}(o)$ and $D_F(o) \geq D_{F'}(o)$, for each weakly connected component $P' \in \Pi_{F'}$ and each two ports $x_1, x_2 \in P' \cap P_F$ it holds $m_{F'}(x_1) - m_{F'}(x_2) = m_F(x_1) - m_F(x_2)$, and for each port $x \in P_{F'} \cap P_F$ such that $l_{F'}(x) = 1$ it holds $l_F(x) = 1$ and $m_F(x) = m_{F'}(x)$.*

Proof. We first prove properties 4 and 5 of the refinement relation definition if all conditions of the lemma are satisfied. If for each input port $i \in I_{F'}$ we have $d_F(i) \geq d_{F'}(i)$ and $D_F(i) \leq D_{F'}(i)$, i.e., $[d_F(i), D_F(i)] \subseteq [d_{F'}(i), D_{F'}(i)]$, we would directly have $\mathcal{S}_{F|I_{F'}} \subseteq \mathcal{S}_{F'|I_{F'}}$ if for each port $x \in P_{F'}$ it holds $l_F(x) = l_{F'}(x)$ and $m_F(x) = m_{F'}(x)$. According to the lemma assumption these conditions are satisfied for locked ports x ($l_F(x) = 1$). If $\rho_{F'} \subseteq \rho_F$ we have $\Pi_{F'} \leq \Pi_F$, i.e., for each weakly component $P' \in \Pi_{F'}$ of $G_{F'}$ there exists a weakly connected component $P \in \Pi_F$ of G_F such that $P' \subseteq P$. Thus for all unlocked ports $x_1, x_2 \in P'$, i.e., $l(x_1) = l(x_2) = 0$, $\mathcal{S}_{F|I_{F'}} \subseteq \mathcal{S}_{F'|I_{F'}}$ follows from the interface semantics definition and lemma assumption $m_{F'}(x_1) - m_{F'}(x_2) = m_F(x_1) - m_F(x_2)$. Similar argument holds for property 5 if for each output port $o \in O_F$ we have $d_F(o) \leq d_{F'}(o)$ and $D_F(o) \geq D_{F'}(o)$, i.e., $[d_F(o), D_F(o)] \supseteq [d_{F'}(o), D_{F'}(o)]$.

For the opposite direction we assume $F' \preceq F$ and prove constraints of the lemma. From the refinement constraint $\mathcal{S}_{F|I_{F'}} \subseteq \mathcal{S}_{F'|I_{F'}}$ and with similar reasoning as above it follows that for each input port $i \in I_{F'}$ it holds $d_F(i) \geq d_{F'}(i)$ and $D_F(i) \leq D_{F'}(i)$. Also, if $F' \preceq F$ then from the requirement 5 we have $d_F(o) \leq d_{F'}(o)$ and $D_F(o) \geq D_{F'}(o)$ for each output port $o \in O_F$.

From the constraint $\mathcal{S}_{F|I_{F'}} \subseteq \mathcal{S}_{F'|I_{F'}}$ it also follows that for each input port $i \in I_{F'}$ such that $l_{F'}(i) = 1$ it is also $l_F(i) = 1$, otherwise interface F will allow for more input behaviors. Remember, for each output port $o \in O_{F'}$ there exists an input port $i \in I_{F'}$ such that there exists a path in $G_{F'}$ from i to o. If port o is locked ($l_{F'}(o) = 1$) than also $l_{F'}(i) = 1$ and, as explained above, $l_F(i) = 1$. Since $\rho_{F'} \subseteq \rho_F$ there exists a path from i to o also in G_F, and, thus $l_F(o) = 1$. Therefore, for each $x \in P_{F'} \cap P_F$ such that $l_{F'}(x) = 1$ it is also $l_F(x) = 1$. In addition, $m_F(x) = m_{F'}(x)$ holds, otherwise either requirement 4 or 5 of the refinement relation $F' \preceq F$ would not hold depending on whether $x \in I_{F'}$ or $x \in O_F$. Let $x_1, x_2 \in P_{F'} \cap P_F$ be two ports from the same weakly connected component of $G_{F'}$. If $m_{F'}(x_1) - m_{F'}(x_2) = m_F(x_1) - m_F(x_2)$ does not hold then either requirement 4 or 5 would not hold depending on the input/output character of ports x_1 and x_2. Note that ports x_1 and x_2 can be locked in F even though they are unlocked in F'. $\square$

Proposition 16 (Independent Refinement) *Let $i_1 \in I_F$ and $o_1, o_2 \in O_F$. For all interfaces F, F', and E*

1. *If $F' \preceq F$, $F \| E$ and $F' \| E$ are defined, then $(F' \| E) \preceq (F \| E)$,*

2. *If $F' \preceq F$, $F \to (o_1, i_1)$ and $F' \to (o_1, i_1)$ are defined, then $(F' \to (o_1, i_1)) \preceq (F \to (o_1, i_1))$, and*

3. *If $F' \preceq F$, $F \succ (o_1, o_2, o)$ and $F' \succ (o_1, o_2, o)$ are defined, then $(F' \succ (o_1, o_2, o)) \preceq (F \succ (o_1, o_2, o))$.*

Proof.

1. The condition demands $F' \| E$ to be defined because $F \| E$ can be defined, but not $F' \| E$, e.g. if $P_E \cap (O_F' \setminus O_F) \neq \emptyset$.

   From the definition of the composition operation, $P_{F\|E} = P_F \cup P_E$ and $\rho_{F\|E} = \rho_F \cup \rho_E$, it directly follows $I_{F'\|E} = I_{F'} \cup I_E \subseteq I_F \cup I_E = I_{F\|E}$, $O_{F'\|E} = O_{F'} \cup O_E \supseteq O_F \cup O_E = O_{F\|E}$, and $\rho_{F'\|E} = \rho_{F'} \cup \rho_E \subseteq \rho_F \cup \rho_E = \rho_{F\|E}$.

Since $I_{F'} \subseteq P_F$ and $I_E \subseteq P_E$ we have $\mathcal{S}_{F\|E|I_{F'}} = \mathcal{S}_{F|I_{F'}} \subseteq \mathcal{S}_{F'|I_{F'}} = \mathcal{S}_{F'\|E|I_{F'}}$, and $\mathcal{S}_{F\|E|I_E} = \mathcal{S}_{E|I_E} = \mathcal{S}_{F'\|E|I_E}$. Consequently, $\mathcal{S}_{F\|E|I_{F'\|E}} = \mathcal{S}_{F\|E|I_{F'}\cup I_E} \subseteq \mathcal{S}_{F'\|E|I_{F'}\cup I_E} = \mathcal{S}_{F'\|E|I_{F'\|E}}$. Finally, for each $s \in \mathcal{S}_{F'\|E}$, if $s_{|I_{F'\|E}} = s_{|I_{F'}\cup I_E} \in \mathcal{S}_{F\|E|I_{F'\|E}}$ then $s_{|I_{F'}} \in \mathcal{S}_{F|I_{F'}}$ and according to the definition of the refinement relation $s_{|O_F} \in \mathcal{S}_{F|O_F}$. In addition, we have $s_{|O_E} \in \mathcal{S}_{E|O_E}$. Therefore, we have $s_{|O_{F\|E}} = s_{|O_F\cup O_E} \in \mathcal{S}_{F\|E|O_F\cup O_E} = \mathcal{S}_{F\|E|O_{F\|E}}$.

2. According to the definition of the connection operation, $I_{F\to(o_1,i_1)} = I_F$ (resp. $I_{F'\to(o_1,i_1)} = I_{F'}$) if $path_F(i_1, o_1) = 1$ (resp. $path_{F'}(i_1, o_1) = 1$), and $I_{F\to(o_1,i_1)} = I_F \setminus \{i_1\}$ (resp. $I_{F'\to(o_1,i_1)} = I_{F'} \setminus \{i_1\}$) otherwise. Note that if $path_{F'}(i_1, o_1) = 1$, such a path also exists in G_F, because of the refinement condition $\rho_{F'} \subseteq \rho_F$. In that case, $I_{F'\to(o_1,i_1)} = I_{F'} \subseteq I_F = I_{F\to(o_1,i_1)}$, also by a condition of the relation $F' \preceq F$. Other cases follow similarly.

By the definition of connection operation and refinement relation we have
$$O_{F'\to(o_1,i_1)} = O_{F'} \supseteq O_F = O_{F\to(o_1,i_1)}.$$

The argument for the dependency relation ρ is similar to the one made for inputs. In the case when both $path_F(i_1, o_1) = 1$ and $path_{F'}(i_1, o_1) = 1$, we have $\rho_{F'\to(o_1,i_1)} = \rho_{F'}\cup(\{o_1\}\times Succ_{F'}(i_1)) \subseteq \rho_F\cup(\{o_1\}\times Succ_F(i_1)) = \rho_{F\to(o_1,i_1)}$, because $\rho_{F'} \subseteq \rho_F$ and $Succ_{F'}(i_1) \subseteq Succ_F(i_1)$.

We use Lemma 5 to prove requirements 4 and 5 of the refinement relation $(F'\to(o_1, i_1)) \preceq (F\to(o_1, i_1))$. The connection operation does not modify functions d and D, so from $F' \preceq F$ and Lemma 5 we have for each input port $i \in I_{F'\to(o_1,i_1)}$ it holds $d_{F\to(o_1,i_1)}(i) \geq d_{F'\to(o_1,i_1)}(i)$ and $D_{F\to(o_1,i_1)}(i) \leq D_{F'\to(o_1,i_1)}(i)$, and for each output port $o \in O_{F\to(o_1,i_1)}$ it holds $d_{F\to(o_1,i_1)}(o) \leq d_{F'\to(o_1,i_1)}(o)$ and $D_{F\to(o_1,i_1)}(o) \geq D_{F'\to(o_1,i_1)}(o)$.

Let $P' \in \Pi_{F'\to(o_1,i_1)}$ and let $x \in P'$ be such that $l_{F'\to(o_1,i_1)}(x) = 1$. We first prove $l_{F\to(o_1,i_1)}(x) = 1$ and $m_{F'\to(o_1,i_1)}(x) = m_{F\to(o_1,i_1)}(x)$.

Let $o_1 \notin P'$. Thus, the connection operation $\to(o_1, i_1)$ applied on F' does not modify P' including x, i.e., $l_{F'}(x) = 1$. According to Lemma 5 it holds $l_F(x) = 1$ and $m_{F'}(x) = m_F(x)$. The connection operation does not change the value of l and m functions on locked ports, i.e., $l_{F\to(o_1, i_1)}(x) = 1$ and $m_{F'\to(o_1, i_1)}(x) = m_{F\to(o_1, i_1)}(x)$.

Let $o_1 \in P'$. Since P' is locked, port i_1 or port o_1 or both are locked in F', i.e., $l_{F'}(i_1) = 1$ or $l_{F'}(o_1) = 1$. If $l_{F'}(x) = 1$, then the argument follows as above. Thus, let $l_{F'}(x) = 0$, and assume $l_{F'}(o_1) = 1$ and $l_{F'}(i_1) = 0$, i.e., ports x and i_1 are in the same weakly connected component of $G_{F'}$. Other cases follow analogously. Due to the property of connection for unlocked ports we have $m_{F'\to(o_1, i_1)}(x) - m_{F'\to(o_1, i_1)}(o_1) = m_{F'}(x) - m_{F'}(i_1)$ and $m_{F\to(o_1, i_1)}(x) - m_{F\to(o_1, i_1)}(o_1) = m_F(x) - m_F(i_1)$. Since $l_{F'}(o_1) = 1$, we also have $l_F(o_1) = 1$ and therefore $m_{F'\to(o_1, i_1)}(o_1) = m_{F'}(o_1) = m_F(o_1) = m_{F\to(o_1, i_1)}(o_1)$. Due to Lemma 5 property for unlocked ports we have $m_{F'}(x) - m_{F'}(i_1) = m_F(x) - m_F(i_1)$. Consequently, $m_{F'\to(o_1, i_1)}(x) = m_{F\to(o_1, i_1)}(x)$. Since $\rho_{F'} \subseteq \rho_F$ ports x and i_1 are also in the same weakly connected component of G_F. Thus, from $l_F(o_1) = 1$ it follows $l_{F\to(o_1, i_1)}(x) = 1$.

Let $x_1, x_2 \in P' \cap P_{F\to(o_1, i_1)}$. Since x_1 and x_2 are elements of the same weakly connected component P' there is an undirected path from x_1 to x_2, both in $G_{F'\to(o_1, i_1)}$ and $G_{F\to(o_1, i_1)}$ (due to $\rho_{F'\to(o_1, i_1)} \subseteq \rho_{F\to(o_1, i_1)}$). Assume $o_1 \in P'$. We have $m_{F'\to(o_1, i_1)}(x_1) - m_{F'\to(o_1, i_1)}(x_2) = m_{F\to(o_1, i_1)}(x_1) - m_{F\to(o_1, i_1)}(x_2)$ because connection operation does not modify difference between m function values of already connected ports. For instance, if in $G_{F'}$ there exists a path from x_1 to o_1 then $m_{F'\to(o_1, i_1)}(x_1) - m_{F'\to(o_1, i_1)}(o_1) = m_{F'}(x_1) - m_{F'}(o_1) = m_F(x_1) - m_F(o_1) = m_{F\to(o_1, i_1)}(x_1) - m_{F\to(o_1, i_1)}(o_1)$. The middle equation of the previous expression is a consequence of Lemma 5. If $o_1 \notin P'$ we can directly apply Lemma 5 since the connection operation does not affect ports x_1 and x_2.

From all above arguments and Lemma 5 follows $(F' \to (o_1, i_1)) \preceq (F \to (o_1, i_1))$.

3. According to the definition of the join operation and refinement relation,
$I_{F' \succ (o_1, o_2, o)} = I_{F'} \subseteq I_F = I_{F \succ (o_1, o_2, o)}$, $O_{F' \succ (o_1, o_2, o)} = O_{F'} \cup \{o\} \supseteq O_F \cup \{o\} = O_{F \succ (o_1, o_2, o)}$, and $\rho_{F' \succ (o_1, o_2, o)} = \rho_{F'} \cup \{(o_1, o), (o_2, o)\} \subseteq \rho_F \cup \{(o_1, o), (o_2, o)\} = \rho_{F \rightarrow (o_1, i_1)}$.

We use Lemma 5 to prove requirements 4 and 5 of the refinement relation $(F' \succ (o_1, o_2, o)) \preceq (F \succ (o_1, o_2, o))$. The join operation sets only the values for port o. From the definition of $F' \succ (o_1, o_2, o)$ we have $l_{F'}(o_1) = l_{F'}(o_2) = 1$ and $l_{F' \succ (o_1, o_2, o)}(o) = l_{F \succ (o_1, o_2, o)}(o) = 1$. From Lemma 5 we have $m_{F'}(o_1) = m_F(o_1)$ and $m_{F'}(o_2) = m_F(o_2)$. Thus $m_{F' \succ (o_1, o_2, o)}(o) = \max\{m_{F'}(o_1), m_{F'}(o_2)\} = \max\{m_F(o_1), m_F(o_2)\} = m_{F \succ (o_1, o_2, o)}(o)$. In addition, since $d_F(o_1) \leq d_{F'}(o_1)$ and $d_F(o_2) \leq d_{F'}(o_2)$, we have $d_{F' \succ (o_1, o_2, o)}(o) = \max\{m_{F'}(o_1) \cdot p + d_{F'}(o_1), m_{F'}(o_2) \cdot p + d_{F'}(o_2)\} - m_{F' \succ (o_1, o_2, o)}(o) \cdot p \geq \max\{m_F(o_1) \cdot p + d_F(o_1), m_F(o_2) \cdot p + d_F(o_2)\} - m_{F \succ (o_1, o_2, o)}(o) \cdot p = d_{F \succ (o_1, o_2, o)}(o)$. Similarly, since $D_F(o_1) \geq D_{F'}(o_1)$ and $D_F(o_2) \geq D_{F'}(o_2)$, we have $D_{F \succ (o_1, o_2, o)}(o) \geq D_{F' \succ (o_1, o_2, o)}(o)$. From all above arguments and Lemma 5 follows $(F' \succ (o_1, o_2, o)) \preceq (F \succ (o_1, o_2, o))$.

$\square$

So, we proved that even in case of task graphs, in order to refine a given composition of interfaces, it suffices to independently refine each interface and to compose the obtained refinements.

Example. We demonstrate the independent refinement property on the task graph shown in Fig. 4.22. Assume tasks t_1, t_2 and t_3 are allocated to processor r_1 and t_4, t_5 and t_6 to processor r_2. Let r_1 and r_2 have the same processing power and let all tasks have the same execution time requirements. In particular, let for each task the uncertainty interval of execution time, i.e., the time needed to process the task if the task had a dedicated resource, be $[e_j, E_j] = [2, 3]$ for each $1 \leq j \leq 6$. We also assume that both processors schedule tasks according to the preemptive fixed-priority mechanism where lower task in-
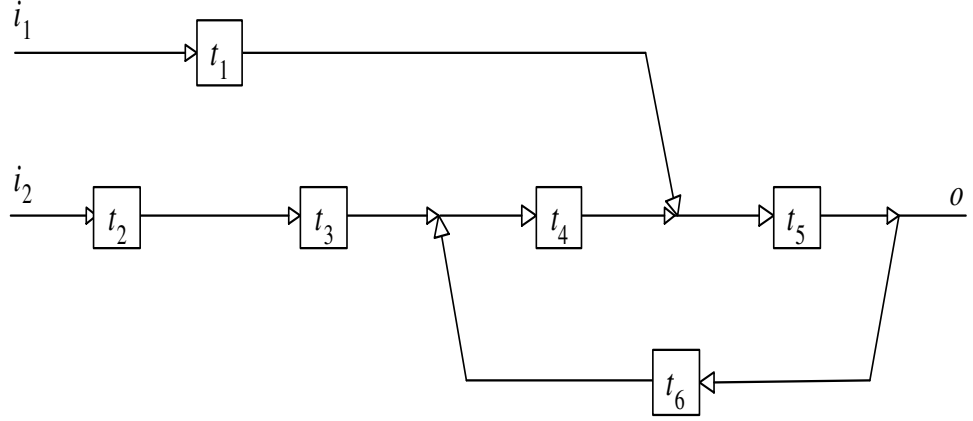
Figure 4.22. Example task graph

dex corresponds to higher priority. Finally, the period of the system is $p = 20$ and event models for inputs are $(m(i_1), d(i_1), D(i_1)) = (0, 0, 1) = (m(i_2), d(i_2), D(i_2))$, i.e., both input events occur within interval [0,1] every period. Fig. 4.23 a) shows F_1, the interface of the component that implements task graph from Fig. 4.22, constructed from interfaces of the individual tasks. If the tasks are considered separately, i.e., if task dependencies are ignored, then taking into account priorities of the tasks on processor r_1, for the intervals of task response times we have $[w_1, W_1] = [e_1, E_1] = [2, 3]$, $[w_2, W_2] = [e_2, E_1 + E_2] = [2, 6]$ and $[w_3, W_3] = [e_3, E_1 + E_2 + E_3] = [2, 9]$. Thus, when composition, connection and join operations are applied (shown with full arrows in Fig. 4.23 a), for the output of task t_3 we have $d_{F_1}(o_3) = d(i_1) + e_2 + e_3 = 4$ and $D_{F_1}(o_3) = D_{F_1}(i_1) + E_2 + E_3 = 16$. Similarly, for output port o we obtain $d_{F_1}(o) = 8$ and $D_{F_1}(o) = 25$. Fig. 4.23 b) shows F_2, the interface obtained if dependencies between tasks on processor r_1 are known before interface operations are applied. The facts that task t_1 can preempt either t_2 or t_3 but not both, and that t_2 cannot preempt t_3, can be used in this case to compute stronger guarantee, i.e., smaller uncertainty interval for the output of task t_3. In particular, $d_{F_2}(o_3) = d(i_1) + e_1 + e_2 + e_3 = 6$ and $D_{F_2}(o_3) = D_{F_1}(i_1) + E_1 + E_2 + E_3 = 10$, and $d_{F_2}(o) = 10$ and $D_{F_1}(o) = 19$. Thus, as a consequence of $[d_{F_2}(o_3), D_{F_2}(o_3)] \subseteq [d_{F_1}(o_3), D_{F_1}(o_3)]$ we have $[d_{F_2}(o), D_{F_2}(o)] \subseteq [d_{F_1}(o), D_{F_1}(o)]$, i.e., $F_2 \preceq F_1$. Finally, if task graph on processor

137

r_2 is considered from the beginning as shown in Fig. 4.23 c), we obtain $d_{F_2}(o) = 10$ and $D_{F_1}(o) = 16$, i.e., $F_3 \preceq F_2 \preceq F_1$.

4.6 Conclusion

We started this chapter by showing how a group of tasks, each defined with an arrival rate curve, a delay, and a worst-case execution requirement, can be abstracted into a bounded-delay resource model. In order to use such abstracted components in a larger real-time system comprising of multiple task sequences we introduced component interfaces. A formal interface algebra allows for automatic procedures that enable component integration. We motivated and proved two properties of such a framework, incremental design and independent refinement. Next we formalized a similar interface theory, but for richer task models, those in which the underlying task precedence graph can be an arbitrary graph. This was studied for periodic event model with jitter and phase. Even such a less general event stream representation results in an algebra that is not as flexible as the one in case of task sequences. However, we proved that if pertinent interface compositions are defined both associativity and independent refinement properties hold even for components that implement general task graphs including cyclic graphs. In this context, it remains unclear how to address more complicated event stream specifications, such as event streams specified with general lower and upper arrival rate functions. The composition with abstracted components inevitably incurs higher resource utilization and, therefore, effectiveness of composition can be compromised. We leave the question of how tight the entire framework is for some future work. In addition, interesting problems for future investigations arise when more complex, temporally adaptive interactions between real-time components require extension to an automaton-based interface formalism.
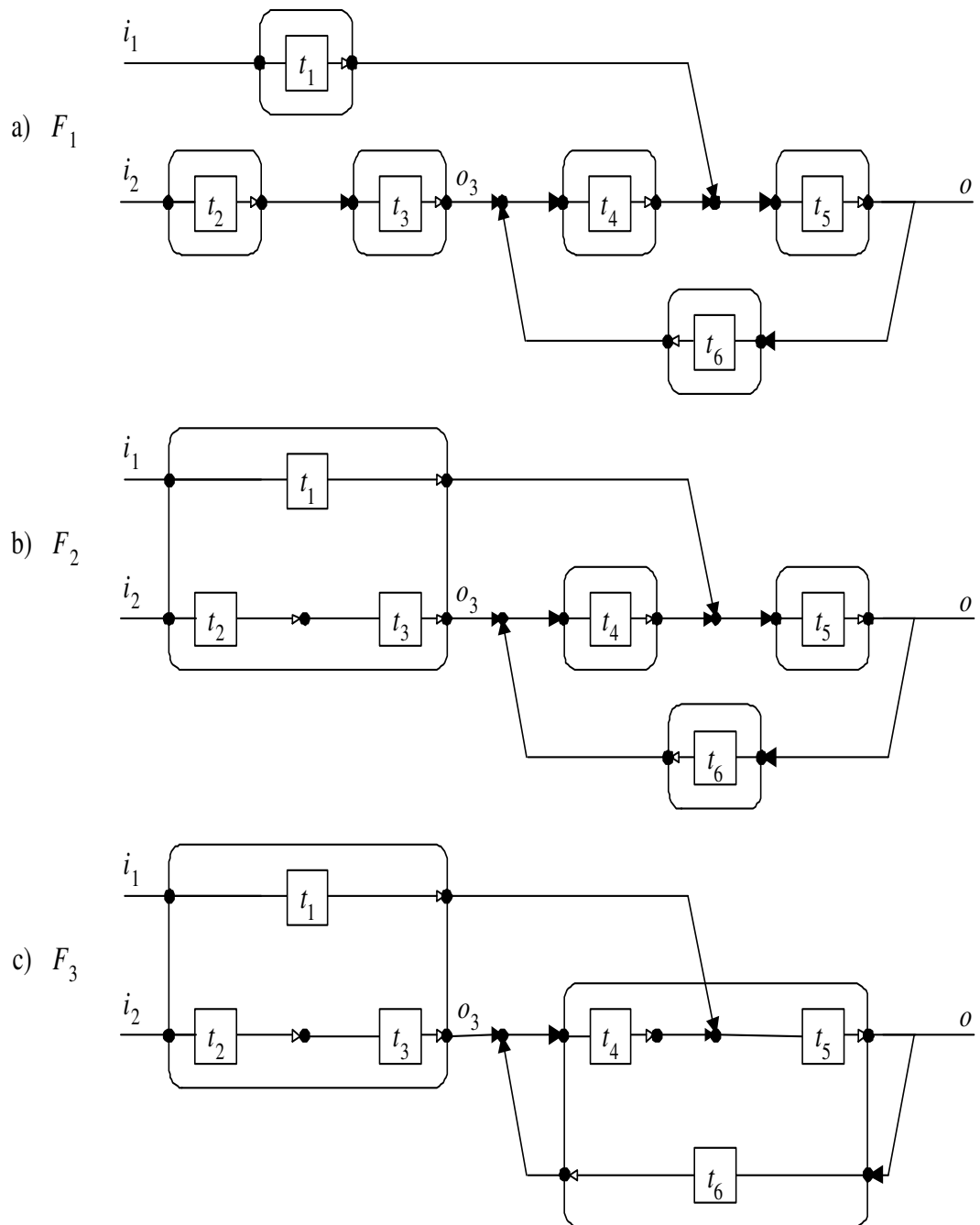
Figure 4.23. Independent refinement for the task graph in Fig. 4.22

Chapter 5

Conclusions and Outlook

This thesis presents several methods for compositional design and verification of real-time systems. We formally address higher layers of design, in particular, task and distribution layers. All studied models of computation include inter- or intra-component task communication. Some of the problems discussed include: code generation and scheduling with limited coordination, verification procedures for interface compliance checking, abstraction of component resource requirements, composition of resource requirements, composability tradeoffs, associativity of interface operators and independent component implementability. A common thread for all methods is an objective to work with a minimal but complete interface needed for a model or problem being considered.

Although the general goal for all compositional methods is the same, i.e., to simplify solution to the original large problem by solving several smaller problems, the problems and solutions in different chapters of the thesis are in several aspects considerably different. Chapter 2 and 3 study time-triggered, whereas Chapter 4 event-triggered models of computation. While the problems in the first two chapters try to estimate several composability measures, e.g. performance tradeoffs or complexity of composability checking and interface compliance, the problems in the last chapter are predominantly of qualitative nature.

The methods that involve LET semantics try to establish time predictability by reducing sensitivity to unknown system information, whereas interface-based methods try to match implementation concepts with analysis.

In principle, some of these methods can be integrated within the existing tools. As discussed in Chapter 3 the LET semantics is very similar to the RTW semantics used in Simulink. However, Simulink tools currently concentrate on code generation and not on formal performance analysis. On the other hand, scheduling analysis has found its way into industrial practice, e.g. [74]. The methods that are available are typically global holistic methods, and, as such, are largely ignored due to complexity. Attempts to use SymTA/S approach within the Autosar project are good signs for the compositional and hierarchical methods. The methods that tackle cyclic component dependencies, including the results in this thesis, are still very limited in reach.

The timing verification techniques should not only be reliable, but also precise. More complex models or requirement specifications ask for more complex analyses, a tradeoff that must be carefully considered. In particular, currently there are no research results that study tradeoffs between timing predictability (difference between estimated and measured timing) and computational complexity of the analysis. It is often argued that formal performance analysis results in conservative designs. However, this criticism is only partially true, since, as noted in [39], over-provisioning in simulation and test methods can often result in designs with much more conservative outcomes. On the other hand, formal analyses require far less computation time, which renders them very effective for rapid design-space exploration.

A drawback of formal performance methods discussed in this thesis is their bad correlation with lower layers of system design, such as single task timing analysis and compiler optimizations. Similarly, the timing consequences of interrupts occurring at unknown program states can hardly be estimated with high accuracy. So, often it is difficult to find

good event models with sufficient precision. In addition, many properties of distributed real-time systems cannot be modeled easily, so links to upper system layers may also be open problems.

To simplify integration, most real-time communication protocols are static time-division protocols. On the other hand, more and more applications are becoming dynamic in character. For instance, the application throughput varies during the execution and cannot be easily predicted in advance. For such applications static protocols lead to both large buffer sizes and long response times. Thus, the challenge here would be to have dynamic resource reservation and reclaiming together with performance guarantees.

Despite the current limitations of system-level formal timing analysis techniques, we believe that the increasing number of performance dependencies in complex systems will lead designers to accept performance analysis, probably by matching them with moderately restrictive implementation methods. This might be the way to achieve both good performance and productivity, and time predictability with specified precision.

Bibliography

[1] AbsInt. In *http://www.absint.com/ait/*.

[2] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: Response-time analysis and server design. In *EMSOFT*, pages 95–103. ACM Press, 2004.

[3] AUTOSAR. In *www.autosar.org/*.

[4] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling: capturing causality and the correctness of loosely time-triggered architectures (ltta). In *EMSOFT*, pages 220–229. ACM Press, 2004.

[5] A. Benveniste, B. Caillaud, and P. L. Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163:125–171, 2000.

[6] A. Benveniste, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In *EMSOFT*, pages 35–50, 2003.

[7] J.-Y. L. Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag New York, 2001.

[8] P. Brucker, S. Kravchenko, and Y. Sotskov. Preemptive job-shop scheduling problems with a fixed number of jobs. *Mathematical Methods of Operations Research*, 49:41–76, 1999.

[9] G. C. Buttazzo. Rate monotonic vs. edf: Judgment day. *Real-Time Systems*, 29:5–26, 2005.

[10] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to scade/lustre to tta: a layered approach for distributed embedded applications. In *LCTES*, pages 153–162, 2003.

[11] S. Chakraborty, Y. Liu, N. Stoimenov, L. Thiele, and E. Wandeler. Interface-based rate analysis of embedded systems. In *RTSS*, pages 25–34, 2006.

[12] S. Chatterjee and J. K. Strosnider. Distributed pipeline scheduling: A framework for distributed, heterogeneous real-time system design. *The Computer Journal*, 38:271–285, 1995.

[13] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[14] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2001.

[15] L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Timed interfaces. In *EMSOFT*, volume 2491 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2002.

[16] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *RTSS*, pages 308–319. IEEE Computer Society, 1997.

[17] B. P. Douglass. Real-time uml. In *FTRTFT*, pages 53–70, 2002.

[18] dSPACE. In *http://www.dspaceinc.com/ww/en/inc/home/products/hw /accessories/autobox.cfm*.

[19] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Incremental schedulability analysis of hierarchical real-time components. In *EMSOFT*, pages 272–281, 2006.

[20] S. A. Edwards and E. A. Lee. The case for the precision timed (pret) machine. In *DAC*, pages 264–265, 2007.

[21] Embedded-Market-Forecasters. In *http://www.embeddedforecast.com*.

[22] H.-G. Frischkorn. Automotive software systems. In *ASW*, pages 0–25, 2004.

[23] S. Goddard. Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application. In *IEEE Real Time Technology and Applications Symposium*, pages 60–71, 1997.

[24] S. Goddard and K. Jeffay. Managing latency and buffer requirements in processing graph chains. *The Computer Journal*, 44:486–503, 2001.

[25] G. Gößler and A. L. Sangiovanni-Vincentelli. Compositional modeling in metropolis. In *EMSOFT*, pages 93–107, 2002.

[26] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1992.

[27] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1):13–28, 1994.

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

[29] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the symta/s approach. *IEE Proceedings Computers and Digital Techniques*, 152(2):149–166, 2005.

[30] T. A. Henzinger, Christoph, M. Kirsch, M. A. Sanvido, and W. Pree. From control models to real-time code using giotto. In *EMSOFT*. Lecture Notes in Computer Science, Springer-Verlag, 2002.

[31] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

[32] T. A. Henzinger and C. M. Kirsch. The embedded machine: predictable, portable real-time code. *SIGPLAN Not.*, 37(5):315–326, 2002.

[33] T. A. Henzinger, C. M. Kirsch, R. Majumdar, and S. Matic. Time-safety checking for embedded programs. In *EMSOFT*, pages 76–92, 2002.

[34] T. A. Henzinger, C. M. Kirsch, and S. Matic. Schedule-carrying code. In *EMSOFT*, pages 241–256, 2003.

[35] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed giotto. In *LCTES*, pages 21–30, 2005.

[36] T. A. Henzinger and S. Matic. *Distributed Schedule-Carrying Code.* Tech. Report UCB//CSD-04-1360, University of California at Berkeley, EECS Department, 2004.

[37] T. A. Henzinger and S. Matic. An interface algebra for real-time components. In *IEEE Real Time Technology and Applications Symposium*, pages 253–266, 2006.

[38] N. J. H. Ip and S. A. Edwards. A processor extension for cycle-accurate real-time software. In *EUC*, pages 449–458, 2006.

[39] M. Jersak, K. Richter, and R. Ernst. Performance analysis for complex embedded applications. *International Journal of Embedded Systems*, 1(1-2):33–49, 2006.

[40] M. Jersak, K. Richter, R. Ernst, J.-C. Braam, Z.-Y. Jiang, and F. Wolf. Formal methods for integration of automotive software. In *DATE*, pages 20045–20050, 2003.

[41] K. Karplus and A. Strong. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 2(7):43–55, 1983.

[42] C. Kirsch, M. A. Sanvido, and T. A. Henzinger. A programmable microkernel for real-time systems. In *VEE*, pages 35–45. ACM Press, 2005.

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

[44] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The time-triggered ethernet (tte) design. In *ISORC*, pages 22–33, 2005.

[45] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1), 2003.

[46] H. Kopetz and N. Suri. Compositional design of rt systems: A conceptual basis for specification of linking interfaces. In *ISORC*, pages 51–60. IEEE Computer Society, 2003.

[47] S. Lankes, A. Jabs, and M. Reke. A time-triggered ethernet protocol for real-time corba. In *ISORC*, pages 215–222, 2002.

[48] E. A. Lee. *Overview of the Ptolemy Project*. Tech. Report UCB/ERL M01/11, University of California at Berkeley, EECS Department, 2001.

[49] E. A. Lee. Absolutely positively on time: What would it take? *IEEE Computer*, 38(7):85–87, 2005.

[50] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computing*, 36:24–35, 1987.

[51] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *ECRTS*, pages 151–158. IEEE Computer Society, 2003.

[52] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti. A hierarchical framework for component-based real-time systems. In *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 209–216. Springer, 2004.

[53] S. Matic and T. A. Henzinger. Trading end-to-end latency for composability. In *RTSS*, pages 99–110, 2005.

[54] S. Matic and T. A. Henzinger. An interface algebra for real-time graphs. In *FMCO*, 2007.

[55] Mentor-Graphics. In *http://www.mentor.com/products/fv/hwsw_coverification /seamless/index.cfm*.

[56] A. K. Mok and A. X. Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. In *RTSS*, pages 129–138. IEEE Computer Society, 2001.

[57] A. K. Mok and A. X. Feng. A model of hierarchical real-time virtual resources. In *RTSS*, pages 26–35. IEEE Computer Society, 2002.

[58] A. K. Mok and A. X. Feng. Real-time virtual resource: A timely abstraction for embedded systems. In *EMSOFT*, volume 2491 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2002.

[59] A. K. Mok, A. X. Feng, and D. Chen. Resource partition for real-time systems. In *RTAS*, pages 75–84. IEEE Computer Society, 2001.

[60] T. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *CODES*, pages 187–192, 2002.

[61] J. Regehr and J. A. Stankovic. Hls: A framework for composing soft real-time schedulers. In *RTSS*, pages 3–14. IEEE Computer Society, 2001.

[62] K. Richter, M. Jersak, and R. Ernst. A formal approach to mpsoc performance verification. *IEEE Computer*, 36(4):60–67, 2003.

[63] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *DAC*, pages 287–292, 2002.

[64] J. Rushby. *Partitioning for Avionics Architectures: Requirements, Mechanisms, and Assurance*. NASA Report CR-1999-209347, NASA Langley Research Center, 1999.

[65] S. Saewong, R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierar hical fixed-priority scheduling. In *ECRTS*, pages 173–181, 2002.

[66] Semiconductor-Industry-Association. The international technology roadmap for semiconductors. In *http://www.itrs.net/Links/2001ITRS/ExecSum.pdf*, 2001.

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

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

[69] I. Shin and I. Lee. Compositional real-time scheduling framework. In *RTSS*, pages 57–67. IEEE Computer Society, 2004.

[70] The-Flexray-Consortium. In *http://www.flexray-group.com*.

[71] The-MathWorks. Models with multiple sample rates. In *Real-Time Workshop User Guide*, pages 1–34, 2005.

[72] L. Thiele, E. Wandeler, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *EMSOFT*, pages 34–43, 2006.

[73] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[74] Tri-Pacific-Software. In *http://www.tripac.com/html/prod-fact-rrm.html*.

[75] E. Wandeler and L. Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *EMSOFT*, pages 80–89. ACM Press, 2005.

[76] E. Wandeler and L. Thiele. Interface-based design of real-time systems with hierarchical scheduling. In *IEEE Real Time Technology and Applications Symposium*, pages 243–252, 2006.

[77] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao. Real-time component-based systems. In *RTAS*, pages 428–437. IEEE Computer Society, 2005.

[78] T.-Y. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Trans. Parallel Distrib. Syst.*, 9(11):1125–1136, 1998.

[79] V. Yodaiken. Rtlinux manifesto. In *Linux Expo*, 1999.

[80] M. Zennaro and R. Sengupta. Distributing synchronous programs using bounded queues. In *EMSOFT*, pages 325–334, 2005.

[81] Y. Zhao, J. Liu, and E. A. Lee. A programming model for time-synchronized distributed real-time systems. In *RTAS*, pages 259–268, 2007.

[82] Y. Zhou and E. A. Lee. A causality interface for deadlock analysis in dataflow. In *EMSOFT*, pages 44–52, 2006.

[83] D. Ziegenbein, M. Jersak, K. Richter, and R. Ernst. Breaking down complexity for reliable system-level timing validation. In *EDP*, 2002.

[84] D. Ziegenbein, F. Wolf, K. Richter, M. Jersak, and R. Ernst. Interval-based analysis of software processes. In *LCTES/OM*, pages 94–101, 2001.