Real-time Multimedia Computing on Off-The-Shelf Operating Systems:

From Timeliness Dataflow Models to Pattern Languages

Pau Arumí Albó

Director: Dr. Vicente Lopez

Co-director: Dr. Xavier Amatriain

Department of Technology

Universitat Pompeu Fabra

Doctorate in Computer Science and Digital Communication

Barcelona, 2009

Abstract

Software-based multimedia systems that deal with real-time audio, video and graphics processing are pervasive today, not only in desktop workstations but also in ultra-light devices such as smart-phones. The fact that most of the processing is done in software, using the high-level hardware abstractions and services offered by the underlying operating systems and library stacks, enables for quick application development. Added to this flexibility and immediacy (compared to hardware oriented platforms), such platforms also offer soft real-time capabilities with appropriate latency bounds. Nevertheless, experts in the multimedia domain face a serious challenge: the features and complexity of their applications are growing rapidly; meanwhile, real-time requirements (such as low latency) and reliability standards increase.

This thesis focus on providing multimedia domain experts with workbench of tools they can use to model and prototype multimedia processing systems. Such tools contain platforms and constructs that reflect the requirements of the domain and application, and not accidental properties of the implementation (such as thread synchronization and buffers management). In this context, we address two distinct but related problems: the lack of models of computation that can deal with continuous multimedia streams processing *in real-time*, and the lack of appropriate abstractions and systematic development methods that support such models.

Many actor-oriented models of computation exist and they offer better abstractions than prevailing software engineering techniques (such as object-orientation) for building real-time multimedia systems. The family of Process Networks and

Dataflow models —based on networks of connected processing actors— are the most suited for continuous stream processing. Such models allow to express designs close to the problem domain (instead of focusing in implementation details such as threads synchronization), and enable better modularization and hierarchical composition. This is possible because the model does not over-specify how the actors must run, but only imposes data dependencies in a declarative language fashion.

These models deal with multi-rate processing and hence complex periodic actor's execution schedulings. The problem is that the models do not incorporate the concept of time in a useful way and, hence, the periodic schedules do not guarantee real-time and low latency requirements. This dissertation overcomes this shortcoming by formally describing a new model that we named *Time-Triggered Synchronous Dataflow (TTSDF)*, whose periodic schedules can be interleaved by several time-triggered "activations" so that inputs and outputs of the processing graph are regularly serviced. The TTSDF model has the same expressiveness (or equivalent computability) than the Synchronous Dataflow (SDF) model, with the advantage that it guarantees minimum latency and absence of gaps and jitter in the output. Additionally, it enables run-time load balancing between callback activations and parallelization.

Actor-oriented models are not off-the-shelf solutions and do not suffice for building multimedia systems in a systematic and engineering approach. We address this problem by proposing a catalog of domain-specific *design patterns* organized in a *pattern language*. This pattern language provides design reuse paying special attention to the context in which a design solution is applicable, the competing forces it needs to balance and the implications of its application.

The proposed patterns focus on how to: organize different kinds of actors connections, transfer tokens between actors, enable human interaction with the dataflow engine, and finally, rapid prototype user interfaces on top of the dataflow engine, creating complete and extensible applications.

As a case study, we present an object-oriented framework (CLAM), and specific applications built upon it, that makes extensive use of the contributed TTSDF model and patterns.

Resum

Els sistemes multimèdia basats en programari capaços de processar àudio, vídeo i gràfics a temps-real són omnipresents avui en dia. Els trobem no només a les estacions de treball de sobre-taula sinó també als dispositius ultra-lleugers com els telèfons mòbils. Degut a que la majoria de processament es realitza mitjançant programari, usant abstraccions del maquinari i els serveis oferts pel sistema operatiu i les piles de llibreries que hi ha per sota, el desenvolupament ràpid d'aplicacions esdevé possible. A més d'aquesta immediatesa i flexibilitat (comparat amb les plataformes orientades al maquinari), aquests plataformes també ofereixen capacitats d'operar en temps-real amb uns límits de latència apropiats. Malgrat tot això, els experts en el domini dels multimèdia s'enfronten a un desafiament seriós: les funcionalitats i complexitat de les seves aplicacions creixen ràpidament; mentrestant, els requeriments de temps-real (com ara la baixa latència) i els estàndards de fiabilitat augmenten.

La present tesis es centra en l'objectiu de proporcionar una caixa d'eines als experts en el domini que els permeti modelar i prototipar sistemes de processament multimèdia. Aquestes eines contenen plataformes i construccions que reflecteixen els requeriments del domini i de l'aplicació, i no de propietats accidentals de la implementació (com ara la sincronització entre threads i manegament de buffers). En aquest context ataquem dos problemes diferents però relacionats: la manca de models de computació adequats pel processament de fluxos multimèdia *en temps-real*, i la manca d'abstraccions apropiades i mètodes sistemàtics de desenvolupament de programari que suportin els esmentats models.

Existeixen molts models de computació orientats-a-l'actor i ofereixen millors

abstraccions que les tècniques d'enginyeria del programari dominants, per construir sistemes multimèdia de temps-real. La família de les Process Networks i els models Dataflow —basades en xarxes d'actors de processat del senyal interconnectats— són els més adequats pel processament de fluxos continus. Aquests models permeten expressar els dissenys de forma propera al domini del problema (en comptes de centrar-se en detalls de la implementació), i possibiliten una millor modularització i composició jeràrquica del sistema. Això és possible perquè el model no sobreespecifica com els actors s'han d'executar, sinó que només imposa dependències de dades en un estil de llenguatge declaratiu.

Aquests models admeten el processat multi-freqüència i, per tant, planificacions complexes de les execucions dels actors. Però tenen un problema: els models no incorporen el concepte de temps d'una forma útil i, en conseqüència, les planificacions periòdiques no garanteixen un comportament de temps-real i de baixa latència. Aquesta dissertació soluciona aquesta limitació a base de descriure formalment un nou model que hem anomenat *Time-Triggered Synchronous Dataflow (TTSDF)*. En aquest nou model les planificacions periòdiques són intercalades per vàries "activacions" temporalment-disparades (time-triggered) de forma que les entrades i sortides de la xarxa de processat poden ser servides de forma regular. El model TTSDF té la mateixa expressivitat (o, en altres paraules, té computabilitat equivalent) que el model Synchronous Dataflow (SDF). Però a més, té l'avantatge que garanteix la operativitat en temps-real, amb mínima latència i absència de forats i des-sincronitzacions a la sortida. Finalment, permet el balancejat de la càrrega en temps d'execució entre diferents activacions de callbacks i la paralel·lització dels actors.

Els models orientats-a-l'actor no són solucions directament aplicables; no són suficients per construir sistemes multimèdia amb una metodologia sistemàtica i pròpia d'una enginyeria. També afrontem aquest problema i, per solucionar-lo, proposem un catàleg de *patrons de disseny* específics del domini organitzats en un *llenguatge de patrons*. Aquest llenguatge de patrons permet el re-ús del disseny, posant una especial atenció al context en el qual el disseny-solució és aplicable, les forces enfrontades que necessita balancejar i les implicacions de la seva aplicació.

Els patrons proposats es centren en com: organitzar diferents tipus de connexions entre els actors, transferir dades entre els actors, habilitar la comunicació dels humans amb l'enginy del dataflow, i finalment, prototipar de forma ràpida interfícies gràfiques d'usuari per sobre de l'enginy del dataflow, creant aplicacions

completes i extensibles.

Com a cas d'estudi, presentem un entorn de desenvolupament (framework) orientat-a-objectes (CLAM), i aplicacions específiques construïdes al seu damunt, que fan ús extensiu del model TTSDF i els patrons contribuïts en aquesta tesis.

Acknowledgements

I would like to thank my supervisor Vicente López for giving me the chance and support to work on the research topic of real-time multimedia, and providing the great environment of Fundació Barcelona Media.

I specially thank Xavier Amatriain who has been my advisor, mentor and friend. He has always been open-minded and consistent supporter. His vision and guidance helped me learn a lot, as well as find and focus on an exciting research topic.

I am very grateful to my friend and long-term co-worker David García for being always ready for a fruitful discussion. Indeed these uncountable discussions have been the source for many of the results presented in this thesis.

I also thanks to all the people who have contributed to the development of the CLAM framework. Beyond Xavier Amatriain and David García, a non-exhaustive list should at least include Maarten de Boer, Ismael Mosquera, Miguel Ramírez, Xavi Rubio, Xavier Oliver and Enrique Robledo. And of course, the students of the Google Summer of Code program: Hernan Hordiales, Natanael Olaiz, Greg Kellum, Andreas Calvo and Yushen Han.

I'd also like to thank our acoustics team in Fundació Barcelona Media, who make it such a great place to work. Including Toni Mateos, Adan Garriga, Jaume Durany, Jordi Arqués, Carles Spa and Giulio Cengarle.

I would like to thank Xavier Serra, for giving me the opportunity to start the doctorate program and work on audio technology in the very creative Music Technology Group. Working there with Maarten de Boer and Oscar Celma has been always fun and rewarding. The members of the Music Technology Group

have provided a first-class environment for discussing and exploring research ideas. Jordi Bonada, Alex Loscos, Emilia Gomez, Bram de Jong, Lars Fabig, Jordi Janer, and many other researches, they all provided positive feedback during the first part of my research.

Thanks to Dietmar Schetz, from Siemens, who was my "shepherd" during the months prior to presenting the patterns catalog to the PLoP 2006 conference. He revised early drafts and gave insights from his experience writing patterns, causing lots of rewritings; all of them worth. Dietmar comments provided a reality check for our ideas.

I am very grateful to Ralph Johnson, a member of the almost mythic "Gang of Four", It was absolutely great to discover that Ralph had many years of professional experience with dataflow-related software. During the three day's workshop sessions at PLoP 2006, Ralph provided insightful feedback and courage to keep our effort on dataflow patterns.

I'm also indebted to the Linux audio community, the Trolltech crew, and to all open-source developers in general to give me the chance to build upon their work. Examining existing code of many multimedia systems, and sharing insight on mailing-lists and IRC discussions was an invaluable source of ideas for this work.

The credit for some of the contributions of this thesis should be clearly shared. The CLAM framework has been a collaborative effort of many developers during the last 8 years. Among those, the credit for CLAM should be shared specially with the other two long-term developers David Garcia and Xavier Amatriain. The credit for the multimedia dataflow pattern language should be shared specially with David Garcia.

Thanks to the chairs of the 2006 ACM Multimedia Contest who gave CLAM the prize to the best Open-Source Multimedia software. This provided a fantastic boost of energy.

Special thanks goes to Martin Gasset, from OFAI, Austria. I provided Martin with a draft of the thesis and he gave me an awesome surprise weeks later, when he come back not only with very useful comments, but also with a finished implementation of the TTSDF model (one of the thesis contributions) for integration into "StreamCatcher", a system for real-time generative computer graphics based on live audio features. I cannot think on a better encouragement than this!

Thanks to George Tzanetakis who carefully reviewed drafts of the thesis and

provided many valuable insights that helped improving the thesis a great deal.

I'd also like to thank Josep Blat and his GTI group, who provided a great support to my research. The CLAM project has been supported by the Universitat Pompeu Fabra, the STSI division of the Catalan Government, the Fundació Barcelona Media and by Google —through the Summer of Code program. This research has been partially funded by a scholarship from Universitat Pompeu Fabra and by financial support from the STSI division of the Catalan Government, and by the European Union (FP7 project 2020 3D Media ICT 2007).

Lastly, my partner Matilda, provided tons of encouragement, patience, and love.

Table of Contents

List of Figures

CHAPTER 1

Introduction

Off-the-shelf operating systems with real-time multimedia processing capabilities are deployed today in all kinds of hardware, ranging from desktop workstations to ultra-light portable devices such as ipods and smart-phones. The fact that most processing is done in software, added to the high-level services offered by such operating systems and the availability of rich reusable libraries stacks enables for relatively quick application development. The software based applications developed for these platforms can fulfill the *soft real-time*[1] with relatively *low-latency*[2] requirements needed in tasks such as interactive video, audio and graphics streams processing. Because of the flexibility and immediacy of these solutions, software-based architectures are often preferred to system-on-chip or embedded software written for a specific hardware.

Nevertheless, experts in the domain of multimedia processing systems face a serious challenge. The features and complexity of their applications are growing rapidly. Meanwhile, real-time requirements (such as low latency) and reliability

[1]Soft real-time refers to a system that acts deterministically and guarantees that is able to execute its workload on time. "Soft" means that the consequences of not meeting a deadline are undesirable but not critical. These concepts are discussed in section 2.1.2.

[2]Latency means the time span between the instant when data is ready to the system's input and the instant the processed data is presented to the user. Refer to section 2.1.2 for details.

standards increase.

The General Goal

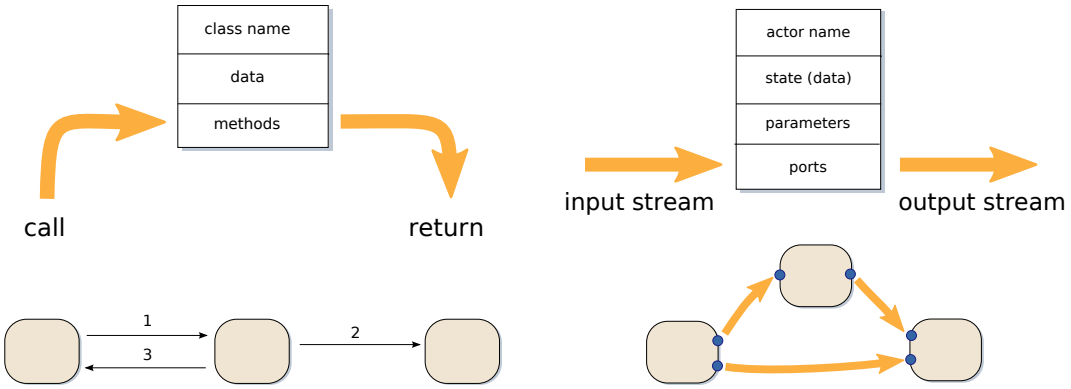
Unlike most software computing domains, multimedia computing heavily relies in the concept of time and concurrency. The problem is that prevailing software engineering techniques like object-orientation say little or nothing about time, and concurrency is typically expressed in low-level constructs like threads and mutexes which do not scale well [Lee, 2002]. And worst, many high-level languages tend to make timing of operations totally unpredictable because of services such as implicit memory allocation and garbage collection. Therefore, today, application developers in this field need to be experts in unrelated disciplines: multimedia signal processing in the one hand, and system-level design and real-time techniques on the other.

The general goal of this thesis is to provide multimedia domain experts with a workbench of tools they can use to describe and communicate multimedia processing problems in languages and constructs close to their domain, while hiding the details referring to the implementation. Most importantly, such tools should not only allow to "model" the problem, but to "run" it efficiently, guaranteeing soft real-time requirements.

The artifacts generated with rapid prototyping tools should not be toy prototypes but robust applications allowing for iterative enrichments until they are ready to ship to the final users. Specifically, they should be designed to fulfill the requirements of *soft real-time* applications (see section 2.1.2 for an analysis of soft and hard real-time) and the interaction with humans or networked systems during the real-time processing of multimedia streams.

Useful Technologies

To reach the previous goal we need platforms with modeling properties that reflect the domain and application requirements, not accidental properties of the implementation. Fortunately, we can take advantage of the emergence of technologies that synergistically combine domain-specific languages, design patterns, actor-oriented models of computation and frameworks into an agile software development process.



(a) Object Orientation (b) Actor Orientation

Figure 1.1: Object versus Actor Orientation. In Object Orientation what flows is sequential control, whereas in Actor Orientation what flows through an object is streams of data, leaving the decision of when (and where) to execute them to the Model of Computation.

Before outlining the concrete issues this thesis addresses we need to briefly introduce the aforementioned concepts.

- A *Domain-specific language* (described in section 3.2) is a corpus of a grammar, syntax and semantics that instead of being focused on a particular computation problem (such as those of procedural programming or data storage and interchange) is designed so that can more directly represent the problem domain which is being addressed. Domain-specific languages often have an explicit *domain-specific meta-model* that gives the semantics of the language defining its concepts and their relations.

- In *actor-oriented design* (described in section 2.2) the components define data dependencies without over-specifying implementation details like the execution control. In this paradigm, the building blocks do not transforms a body of input data into a body of output data and then return, but continuously operate on potentially infinite stream of data. See figure 1.1 for a comparison between *object-orientation* and actor-orientation. An *actor-oriented model of computation* defines how components interact, how they run, and the exact semantics of components composition.

- A *framework* (described in section 2.3.1) is the body of code that implements the aspects that are common across an entire domain, and exposes as extension points those elements in the domain that vary between one application

and another. More or less explicitly, multimedia processing frameworks implement one model of computation, and may have domain-specific languages implemented on top.

- A *design pattern* (described in section 2.4) is a general reusable solution to a commonly occurring problem in software design. Design patterns do not provide code, but a description of how to solve the (design) problem. Design patterns can have many different scopes ranging from architecture to implementation.

Figure 1.2 disposes these technologies in a stack layout indicating their relative level of abstraction and their relations of "use". It also shows how design patterns are useful to leverage one level of abstraction to the next, using a given set of tools and mechanisms.

This figure represents the "big picture" of our goal, and it is worth noting that the maturity or completion of the layers varies heavily. Specifically, the three upper layers (namely: actor-oriented models of computation, frameworks, and domain-specific languages and meta-models) and their associated patterns, are in permanent evolution and offer good research opportunities.

Outline of the Problem and the Proposed Solution

The goal of providing multimedia domain experts with useful workbench of tools is very general. In this thesis we do some steps in this direction narrowing the scope and addressing issues related to actor-oriented models, design patterns and frameworks.

Unfortunately, among all the existing actor-oriented models of computation, those most suited for continuous streams (such as video and audio) processing do not deal well with the concept of time. This is problematic because, with those models, real-time processing with fixed and predictable latencies is not possible. We propose a new actor-oriented model of computation that falls in the family of the *Static Schedulable Dataflows* (see section 2.2.4) that overcomes this problem, while retaining the same expressiveness and decidability properties of the original models.

But having proper models of computation is not enough. Building complete applications on top of the proposed (or similar) model of computation involves taking many design and implementation decisions, each of which leads to non-trivial

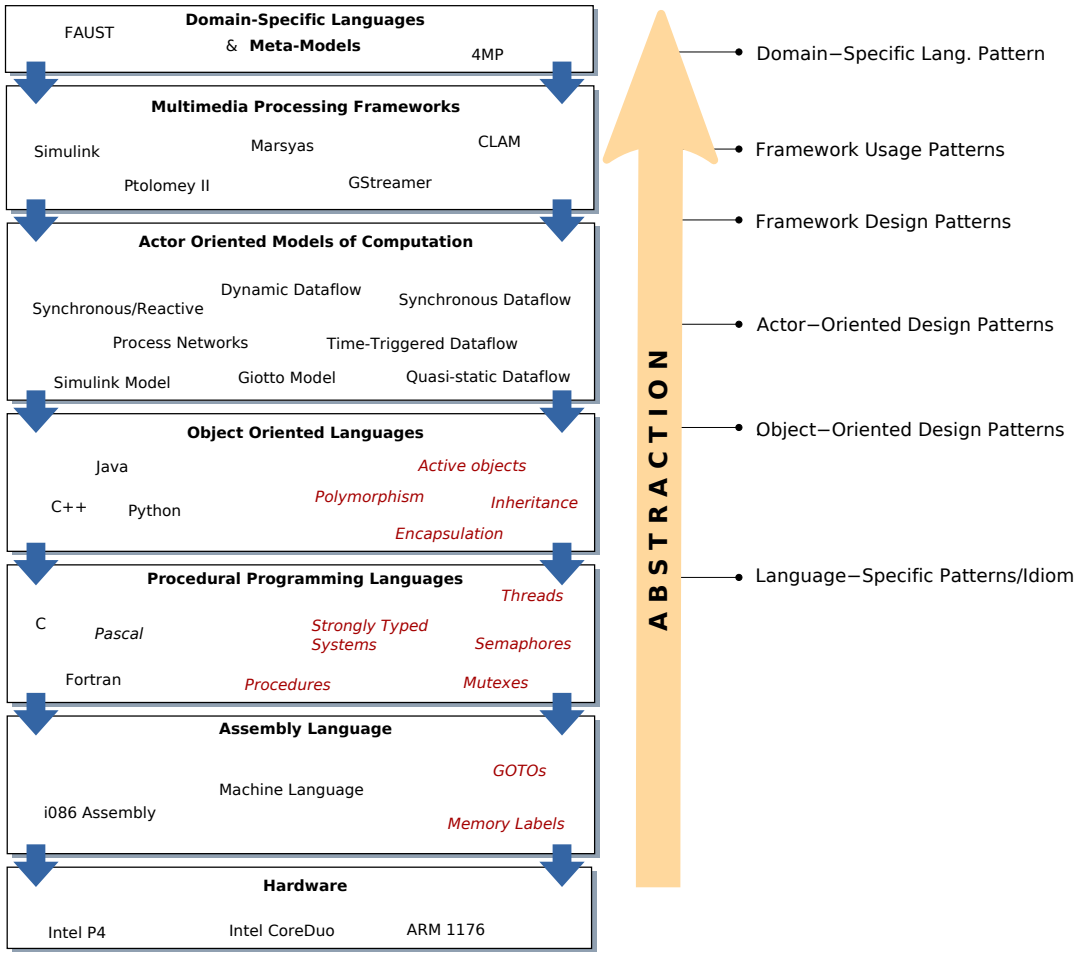


Figure 1.2: The "big picture" of technologies and tools for developing multimedia processing systems, organized in increasing level of abstraction. The upper layers use elements of the layer below. Inside the layers, normal font denotes tools, and italic denotes mechanisms contained in those tools. In the right side of the diagram, different kinds of patterns are listed, which provide means for reaching higher abstraction levels based on specific layers.

consequences. Thus, the potential of failing is huge. We provide a systematic, predictable, engineering approach to such design and implementation using a *design patterns language* (introduced in section 2.4.5). Finally, we show a (software) framework that implements the model and the patterns, and show how it is useful to rapid-prototype our target multimedia processing systems.

In the next sections of the introduction we expose the problem in more detail and explain why it is worth solving it (in section 1.1), outline the given solution (in section 1.2) and the used research method (in section 1.3), and list the main contributions of the thesis (in section 1.4).

1.1 The Problem

As outlined above, this thesis addresses two related problems: the lack of appropriate models of computation in the family of dataflow models that supports real-time operation (and thus, can be *time-triggered*); and the lack of abstractions and systematic development methods that enable both the implementation of such models of computation, and the enrichment of the models with domain-specific requirements.

In the next two sections we discuss both problems.

1.1.1 The Problem of Timeliness and Synchronous Dataflow

The Mutimedia Processing Systems Domain

In this thesis we are interested in the domain of *Multimedia Processing Systems* as defined in [Amatriain, 2007a]. Applications in this domain are *signal processing*

intensive (leaving out, for instance, static multimedia authoring systems), are *software based*; *stream oriented*, in that they work on continuous data, but also support *events* to control the processing carried out; allow *multiple data types* (e.g. audio frames, video, graphics, features): and are *interactive*, in that the user can monitor and modify the streams processing reacting to their content.

Timeliness in Process Networks and Dataflow Models

All actor-oriented models of computation offer modularity, composition and handle parallelism. Although these properties are very desirable, not all available models of computations are useful for continuous streams processing in software; only *Process Networks* (discussed in section 2.2.7), *Dataflow* models (discussed in section 2.2.2), and all their multiple variants are. Such models are very well suited for our domain because they provide excellent abstractions for expressing a signal-processing system in a natural way for humans.

Process Networks and Dataflow models have an important problem related with real-time requirements: The models cannot guarantee operating with optimum *latency* while avoiding *jitter* [3] (section 2.1.2 discusses these two concepts)

The underlying problem is that the computation semantics of such models lacks the notion of time. They only deal with sequentiality and causality of consumed and produced data. Ideally we'd like to specify that certain actors are to be triggered by timed hardware interrupts, and therefore, they should be able to consume and produce their data before given deadlines.

To overcome this problem we are interested in new models that guarantee a given operational latency, which ideally should be optimum. Because of this requirement, models based on Process Networks are discarded. In Process Networks actors are not restricted in any way about the number of tokens they can produce on each execution. Thus, the model provides no means for dealing with latency.

As we further explain in section 2.2, the main difference between Process Networks and Dataflow models is that Dataflow have explicit *firing rules* while Process Networks do not. Such explicit firing rules specify how many tokens an actor needs in each of its inputs in order to execute. This kind of rules are a useful mechanism to define a computational model with bounded latency.

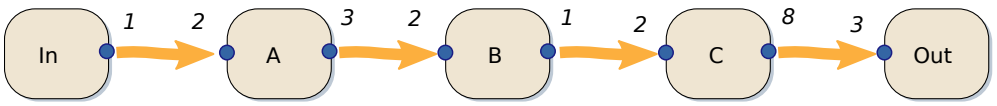
[3] *Latency* is the difference between the time a piece of input enters into the system and the time when the corresponding processed output is presented to the user. *Jitter* is the variation of latency. Jitter is allowed only before the data reaches its presentation deadline not after. [Steinmetz, 1995]

However, within the family of Dataflow models, only the *Synchronous Dataflow* enforces firing rules to remain constant during the model execution. This is why we take this model as the more promising one in respect to the timeliness problem —though it still remains unsuited for the wanted real-time use. Having fixed firing rules enables this model to be a *statically schedulable dataflow* and to have strong formal properties that makes questions, like computability and needed buffer sizes, decidable.

To be specific about the problem with Synchronous Dataflow and real-time we have to refer to their scheduling algorithm. In section 2.2.4 we review the static scheduling algorithm. And in section 4.1 we detail how Synchronous Dataflow periodic schedules do not cope well with real-time. Simply stated, this is because actors that collect input and send output, from and to the outside world are not executed with the needed regularity.

Schedules of Multi-Rate Graphs

While simple Synchronous Dataflow graphs with actors running with the same rate have trivial schedulings —each scheduling period consisting in a sorted, unrepeated list of the actors— and hence do not suffer of the mentioned problem. Graphs that exhibit multi-rate can potentially have long execution periods, with many instances of the same actor. The (multiple) rates in which actors run depends on the defined firing rules or *ports rates*, the number of tokens consumed or produced in each actor execution. In figure 1.3 we show a multi-rate graph with 5 actors, connected in pipeline, and a possible periodic scheduling. We can see that "input" and "output" nodes do not execute regularly, and thus, buffering in the connection pipes is necessary.



Periodic schedule:

In, In, A, B, In, In, A, B, C, In, Out, B, In, Out, A, B, C, In, Out, Out, In, A, B, Out, B, C, Out, Out, Out

Figure 1.3: An example of Synchronous Dataflow graph exhibiting multi-rate, with a possible scheduling period. Numbers at each end of an arc indicates the port rate, or number of tokens consumed or produced by the port in each actor execution.

At this point is important to underline that multi-rate capability is a requirement in our domain. We deal with mixed types of streaming data and they typically correspond to different time lapses. Thus, they need to be processed by the graph at different rates. Examples of data types the domain deals with are: frames of audio samples, audio spectra, video frames, audio/video features associated to a time windows, 3D graphics representations, and so on.

Interestingly, this problem was already made explicit in the article that originally formalized the Synchronous Dataflow model, by Lee and Messerschmitt :

> The SDF model does not adequately address the possible real-time nature of connections to the outside world. [...]. It may be desirable to schedule a node that collects inputs as regularly as possible, to minimize the amount of buffering required on the inputs. As it stands now, the model cannot reflect this, so buffering of input data is likely to be required. [Lee and Messerschmitt, 1987a]

In the state-of-the-art chapter of the thesis (chapter 3) we show that the current state of the art have not given a proper solution to this problem yet.

As a conclusion, we believe that there is a need in the multimedia processing systems domain for new dataflow models that combine the capability of multi-rate computation with time-triggered hardware interrupts, without introducing jitter, extra latency or run-time inefficiencies.

1.1.2 The Lack of Systematic Design Methods for Actor-Oriented Systems

The availability of appropriate actor-oriented models of computation for multimedia processing systems is fundamental for building reliable, efficient and flexible applications in an agile process. Actor-oriented models provide also the core building blocks for high-level tools such as frameworks, or domain-specific languages. Therefore, models of computation are key abstractions that can be thought as the "law of physics" that govern how component interacts and gives the semantics of components composition. They allow engineers to handle the computing complexity because many details —such as: how components can be parallelized, how they synchronize or when they can run— are abstracted away from the engineer. Lee argues [Lee, 2002] that without such abstractions, relying only to low-level mecha-

nisms such as threads, mutexes and semaphores, non-trivial real-time systems are impossible for humans to understand and implement.

However, actor-oriented models are not off-the-shelf solutions and do not suffice for building multimedia processing systems in a systematic and engineering driven approach. We also address this problem in the present thesis.

The problem lies on the fact that there is still a big gap between an instance of an actor-oriented model and the actual realization of a full-featured multimedia application using the model. Actor-oriented models of computation say nothing about how they are to be designed and implemented, or how complete applications can be designed on top of them. Therefore we deal with two related problems:

- The problem of translating the actor-oriented model of computation into an actual design and implementation on a specific software-based platform.

- The problem of enriching the processing core (defined by an instance of an actor-oriented model) to obtain a full-featured application that allows users to interact with the processing core, while keeping consistency with the underlying actor-oriented model.

The Patterns Technology

Software engineering technologies exists that address design reuse in a systematic way. Such technologies are domain-specific *design patterns* organized in *pattern languages*. Pattern languages, are able to guide the development process in a specific domain, answering the following questions for each significant design problem found in the development path:

- What are the *forces* that restrict the present design problem? What is the *solution*?

- Which is the solution that optimizes the tension between forces?

- What are the *consequences* of the solution?

The problem, therefore, can be rephrased as the lack of suited pattern languages in our domain. In section 3.4 we support this affirmation by reviewing the literature on multimedia related design patterns.

Limitations of Frameworks and Code Reuse

The above-mentioned need for design reuse starts from the observation that reusing existing domain-specific function libraries and programming frameworks is not possible, or insufficient, most of the times.

Reusable functions available in libraries leverages specific tasks such as disc access, mathematical operations or audio/video driver access. While useful, alone they do not solve the design problems presented by actor-oriented based designs.

Frameworks have a much broader scope than function libraries. They allow the creation (or instantiation) of complete applications with little effort. Frameworks also provide mechanisms for extension and parametrisation. Actually, frameworks provide not only code reuse but a both code and design reuse. Frameworks are further discussed in section 2.4.3.

In the multimedia processing systems domain, frameworks provide the run-time environment to actor-oriented models of computation and ease the task of developing specific applications. Therefore, frameworks play a key role in the general goal of the thesis, which is providing domain experts with a workbench of tools that allows them to express their systems using constructs close to their domain.

However, a great disadvantage of frameworks is that, in order to be easily instantiable, they also must be narrowly tailored. Thus, the set of design decisions that make a framework useful also limits the framework. Such design decisions, for example, might impose a certain balance between generality and efficiency, preventing to optimize certain functionalities or extending the framework in specific ways. Other decisions impose limitations on the platforms it can run, its programming language, the availability of its code, its license conditions, and so on.

Additionally, developing high quality frameworks is a hard task that needs extensive design experience, and this problem is basically equivalent to the problem stated-above about bridging the gap between models and implementations.

Compared to code reuse (e.g. by using frameworks), design patterns languages encompasses a much broader land, and often provide alternative solutions depending on the desired requirements, or a given *tension of forces*, to put it in patterns terminology.

The Lack of a Common Design Vocabulary

A related problem to the lack of design reuse is the lack of a common vocabulary to express designs ideas contained in frameworks. A framework can be viewed as the implementation of a system of design patterns, thus, design patterns can be easily used to boost frameworks documentation [Appleton, 1997], and high-level design ideas among developers.

In consequence, if appropriate pattern languages should exist for the multimedia processing systems domain, existing frameworks (e.g., Supercollider, CSL, GStreamer) [McCartney, 2002, Pope and Ramakrishnan, 2003, Taymans et al., 2008] would be easy to compare in terms of their design decisions and its consequences. Today, this it is a hard task.

Interestingly, most of these frameworks already incorporate "general, object-oriented" patterns (e.g., the ones found in the "Gang of Four" book [Gamma et al., 1995]) in their documentation, but the fundamental domain-specific aspects remain undocumented.

1.2 The Proposed Solution

This thesis addresses two specific and related problems: First, the lack of appropriate dataflow models of computation with real-time capabilities (that is, that incorporates the concept of time in a useful way). And second, the lack of abstractions for systematic development that enables the implementation of such models to realize full-featured applications.

The first problem is addressed with a new actor-oriented model in the family of *synchronous dataflows*, that schedules its actors in a way that input and output data from and to the outside world is collected and produced in a time-regular

basis. The second problem is addressed with a new catalog of domain-specific *design patterns* organized in a *pattern language*. Next sections gives an overview of both solutions.

1.2.1 The Time-Triggered Synchronous Dataflow

We propose a new actor-oriented model of computation, which we have named *Time-Triggered Synchronous Dataflow (TTSDF)* that belongs the category of *statically schedulable datatflows*. In section 4.2 we formalize the model, give a scheduling algorithm and prove their properties.

Unlike its inspiration source —the *Synchronous Dataflow (SDF)* model— our TTSDF model is suited for real-time processing because it ensures a fixed and optimal latency. At the same time, the TTSDF keeps the fundamental features of SDF's, namely:

- It is static schedulable, which enables run-time optimizations. This is possible because ports rates are fixed, or specified *a priori*;
- Its computability is decidable; that is: it is possible to know whether or not a given dataflow graph can process arbitrarily long streams in bounded memory.
- And it is is equally powerful in the sense that all graphs runnable by the SDF model also run by the TTSDF model.

As the "time-triggered" name suggests, the execution of the dataflow graph is driven by regular time interrupts (e.g., from audio or video hardware). The fundamental characteristic of TTSDF is that it distinguishes *timed actors* —the ones that collects input from the outside, and the ones that sends data out; and are linked to the driver clock— from *untimed actors*, which are all the rest.

Compared to SDF, the periodic schedule of TTSDF imposes new restrictions about the valid orderings of timed and untimed actors. The main one is the following: a valid periodic schedule sequence should always be partitioned in one or more time-triggered callback-activated subsequences, while maintaining the token causality restrictions of dataflows. See figure 1.4 and 1.4.

Taking a dynamic point of view on the model, the run-time first sets the needed delays (in the ports buffers), thus, setting the operative latency, and then executes inputs and outputs regularly placed in each timed trigger.

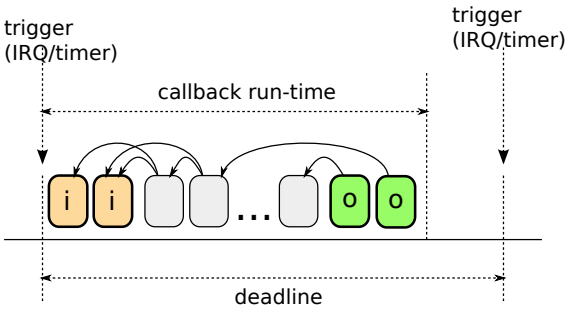


Figure 1.4: A callback activation with two inputs and two outputs. Arrows represents execution dependencies between nodes imposed by the dataflow graph.

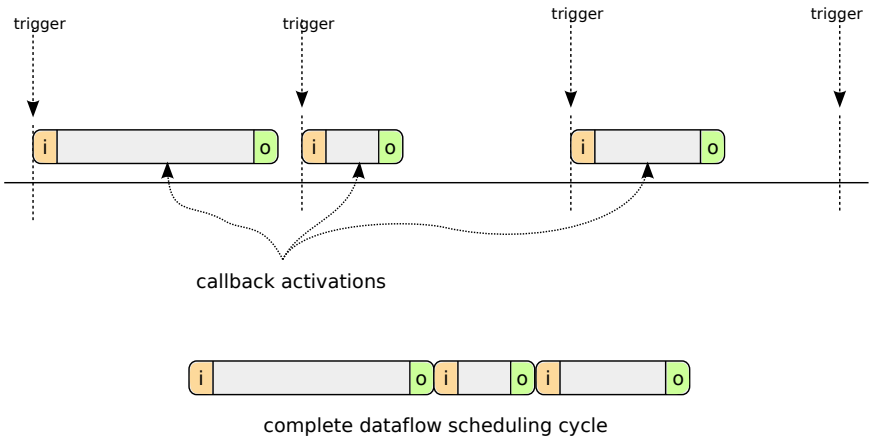


Figure 1.5: Multiple callback activations forming a dataflow cycle scheduling. For clarity, here **i** and **o** represents a sequence of all inputs and outputs.

1.2.2 A Pattern Language for Dataflow-Based Multimedia Processing Systems

Our contribution to the problem of the lack of systematic development methods to implement dataflow-based systems is a catalog of domain-specific design patterns that form a pattern language.

A *pattern* is a proven solution to a recurring generic design problem. A pattern is a literary construct (as opposed to a "code" construct) and it pays special attention to the context in which is applicable, to the competing *forces* it needs to balance, and to the positive and negative *consequences* of its application.

A *pattern language*, as described in [Borchers, 2000], is a comprehensive collection of patterns organized in a hierarchical structure. Each pattern references higher-level patterns describing the context in which it can be applied, and lower-level patterns that could be used after the current one to further refine the solution. Patterns and pattern languages are discussed in sections 2.4 and 2.4.5 respectively. Our proposed pattern language is presented in chapter 5.

All the patterns presented in this thesis fit within the generic actor-oriented design paradigm (see section 2.2), in which programs are defined in a declarative style by "connecting" components that operate on infinite data streams.

This paradigm can be easily translated to a *software architecture*. A software architecture is defined by a configuration of architectural elements —components, connectors, and data— constrained in their relationships in order to achieve a desired set of architectural properties [Fielding, 2000]. In many cases, architectural style descriptions have been recast as architectural patterns [Shaw, 1996], because patterns can describe relatively complex protocols of interactions between components (or objects) as a single abstraction, thus including both constraints on behavior and specifics of the implementation.

This is precisely the case of the Dataflow Architecture pattern formalized by Manolescu [Manolescu, 1997] (refer to section 3.4 for a summary of the pattern). The key point of the Dataflow Architecture pattern is that it addresses the problem of designing systems with components that perform a number of sorted operations on similar data elements in a flexible way so they can dynamically change the overall functionality without compromising performance. Figure 1.6 represents a simple example of such architecture.

The pattern language presented in this thesis is related to design decisions

on actor-oriented systems. Therefore, it fit well with the Dataflow Architecture pattern, which acts as the higher-level pattern in the pattern language.

The word "dataflow" (also written "data-flow" or "data flow") is often used with loose semantics. In many occasions, like in the Manolescu's pattern, it is used as a synonym for the general actor-oriented paradigm. In this thesis, unless explicitly stated, we will adhere to the precise definition given in section 2.2.2.

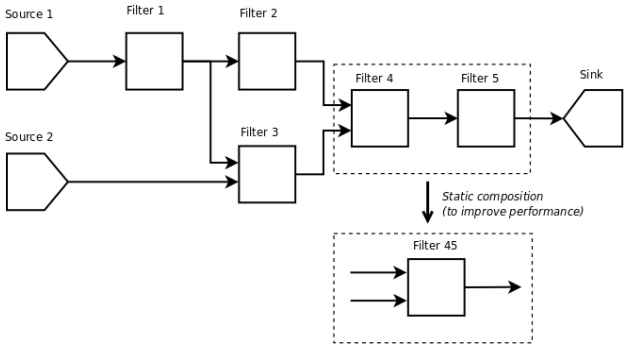


Figure 1.6: Dataflow Architecture

The Dataflow Architecture pattern does not impose any restrictions on issues such as message passing protocol, actors execution scheduling or data management. All these aspects should be addressed in other lower-level and orthogonal design patterns.

The proposed pattern language in this thesis is not the first effort in the field. Manolescu collected and reshaped existing patterns and proposed new ones that fit under the umbrella of the Dataflow Architecture pattern (which can be regarded as the actor-oriented paradigm captured as a software architectural pattern) [Manolescu, 1997]. Some of those patterns —the ones more related to our work— are described in section 3.4. Manolescu's patterns are very high-level compared to ours, and does not address the issues related with the design and implementation of real-time capable models of computation. Both catalogs integrate synergistically and we explicitly relate them.

The patterns proposed in our pattern language (see section 2.4) focus on the four following aspects:

1. How to organize different kinds of actors connections.

2. How to transfer tokens between actors allowing in a flexible and convenient way.

3. How to enable human interaction with the dataflow engine while it is running.

4. How to rapid prototype user interfaces on top of a dataflow engine, creating complete applications without the need for coding —while allowing extensibility by coding.

The patterns are organized hierarchically, specifying their "use" relationships and making explicit the order in which they can be "instantiated" in the code. See figure 1.7 for an illustration of the instantiation process.

The main limitation of our patten language is that it is not comprehensive, in that it does not cover all the design space in the Multimedia Processing Systems domain. However, this is an expected limitation because only small (i.e. very specific) domains have a well-defined design space and allow comprehensive pattern languages. In such ideal conditions, all applications belonging to the domain could be completely derived from instantiations of patterns in the pattern language. Luckily, such incompleteness by no means implies little utility.

We regard the pattern language as something in permanent evolution, expecting new patterns being incorporated to match new trends and requirements in the domain.

Related technologies

The proposed pattern language uses and relates to well established technologies such as Object-orientation (see section 2.3), actor-oriented design and system engineering (see section 2.2); and also to newer technologies such as domain-specific meta-models and languages (see section 3.2) such as the *Meta-Model for Multimedia Processin Systems (4MPS)* [Amatriain, 2004]. 4MPS facilitates the translation of models into object-oriented designs, by giving names and semantics to those design elements that are common in many actor-oriented applications and frameworks. This nomenclature and semantics is also valuable when writing design patterns. Finally, the metamodel explicitly relates actor-oriented design with object-oriented design.

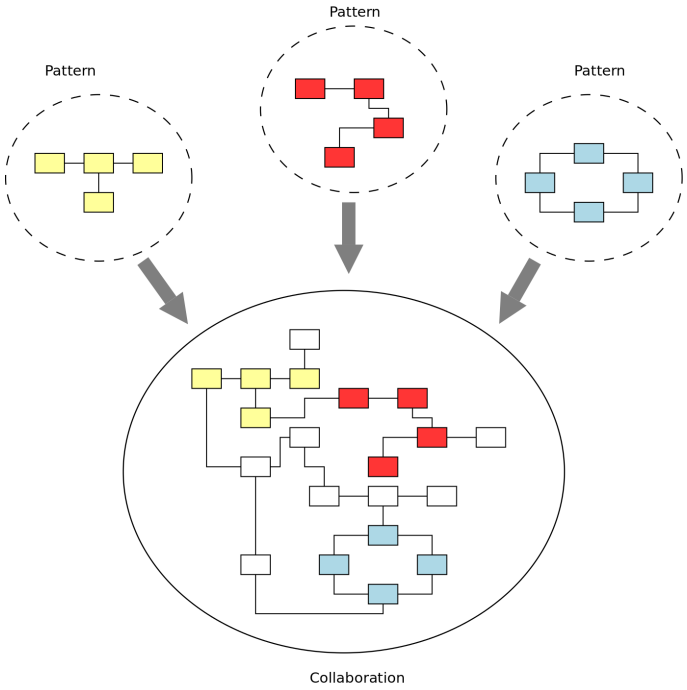


Figure 1.7: Pattern Instantiation: applying generic design solutions to concrete systems under development. The canonical solution given by the pattern must be adapted to concrete situation by relating existing classes with the roles of the solution classes.

1.3 The Method

The two problems addressed by this thesis, though related, need different approaches. The first problem —the real-time static schedulable computation model— is addressed formally, using algebraic methods and formal algorithms to prove all the significant properties of the model. The second problem —the systematic methodology to develop real-time interactive systems on top of dataflow models— cannot be addressed in a similar way. Instead of "calculating" the solution, we capture design experience and insight from many analyzed examples, abstract the commonalities and encapsulate this insight in a proper formalism.

These two methods are commonly used within different disciplines as table 1.1 shows.

Problem	Methodology	Discipline
Real-time statically schedulable model.	Formal methods (linear algebra, formal languages and algorithms).	Systems Engineering (also Embedded Systems).
Systematic methodology to develop dataflow-based systems.	Successful examples analysis, commonality abstraction and formal encapsulation of reusable designs.	Software Engineering.

Table 1.1: The two problems addressed by this thesis, the methodology used in each case, and the discipline where the methodology belongs.

1.3.1 The Systems Engineering Approach

In systems engineering, formal, mathematical-oriented methods are used to model a system and then derive interesting properties from that model.

We formalize our Time-Triggered Synchronous Dataflow model using a directed graph with some added information. Each arc has an associated queue of tokens.

Each node represents an actor which consumes tokens from the ingoing arcs and produces tokens to the outgoing arcs. The model is not concerned with the actual processing that an actor does —this is why a computation model is an *abstractions*, we disregard some aspects of the system. But it is interested in the semantics of the coordination language, therefore we define exactly how actors interact among them.

The most important properties of the model are its computability and its input-to-output latency. These properties depend on how the graph is executed. Therefore we need a proper notation (using formal languages, to describe execution sequences) and techniques to reason about it's schedulability. These techniques include linear algebra and computing algorithms. We define a class of scheduling algorithms and prove that they have the desired properties —namely, they can run statically or before run-time, and have a fixed and minimum latency while retaining the expressiveness that Static Dataflows have.

1.3.2 The Software Engineering Approach

Software engineering methodologies are quite different to other engineering and scientific disciplines in that many of their abstractions and constructs cannot be formally proved, and empirical proofs are often impossible. In such cases, the software engineering methods emphasise on evaluating, capturing and communicating insight. Szypersky speculates [Szyperski, 1998] that software engineers don't "calculate" their designs because software engineering has a much shorter history than other engineering fields that have large bodies of theory accumulated behind them. Software engineers, instead, follow guidelines and good examples of working designs and architectures that help to make successful decisions. Therefore in the context of software engineering, communicating experience, insight, and providing good examples are central activities.

Vlissides, it his "Pattern Hatching" book [Vlissides, 1998] points out that the methodology for growing a collection of highly-related patterns, while keeping them independent, involves many iterations and rewritings. He also identifies the most important part of a pattern: "The difficulties of creating new patterns should not be underestimated. The teaching component of patterns —mostly corresponding to the description and resolution of forces and the consequences of application— is the most important and also the hardest part. "

Beck remarks [Beck et al., 1996] that is their experience that the time-

consuming effort of writing patterns pays off because "the availability of a catalog of design patterns can help both the experienced and the novice designer recognize situations in which design reuse could or should occur." The pattern community is sufficiently enthused about the prospective advantages to be gained by making this design knowledge explicit in the form of patterns, that hundreds of patterns have been written, discussed and distributed.

Patten Mining

Our research provides elements of reusable design for building object-oriented real-time multimedia systems. Consequently we have taken an approach that is appropriate for this objective. The process of creating new design patterns starts by obtaining in-depth experience on a particular domain. The trade-off of forces (quality-of-services) to be optimized in a particular area have to be understood. But also sufficient breadth is needed to understand the general aspects of the solutions and to abstract them into a generalized solution. This process is called *pattern mining*. See figure 1.8 for an illustration of that process. The diagram shows how pattern mining can be regarded as the inverse of pattern instantiation (depicted in figure 1.7).

As Douglass states in his "Real-Time Patterns" book [Douglass, 2003] "pattern mining is not so much a matter of invention as it is of discovery —seeing that this solution in some context is similar to that solution in another context and abstracting away the specifics of the solutions." He also remarks that to be considered a useful pattern it must occur in different contexts and perform a useful optimization of one or more qualities-of-service.

Our patterns have been mined studying several open-source multimedia (mostly audio and/or video) frameworks or environments for application building. The list includes: Aura, SndObj, OSW, STK, CSL, Supercollider, Marsyas, PureData, Max/MSP and GStreamer [Dannenberg and Brandt, 1996a, Lazzarini, 2001, Chaudhary et al., 1999, Cook and Scavone, 1999, Pope and Ramakrishnan, 2003, McCartney, 2002, Tzanetakis and Cook, 2002, Puckette, 1997, Taymans et al., 2008]

However, the main source of experience comes from building and evolving the CLAM framework during the last 8 years. The best methodology to develop a frameworks is by "application-driven development" which means that the framework emerges as the commonalities of the developed applications in

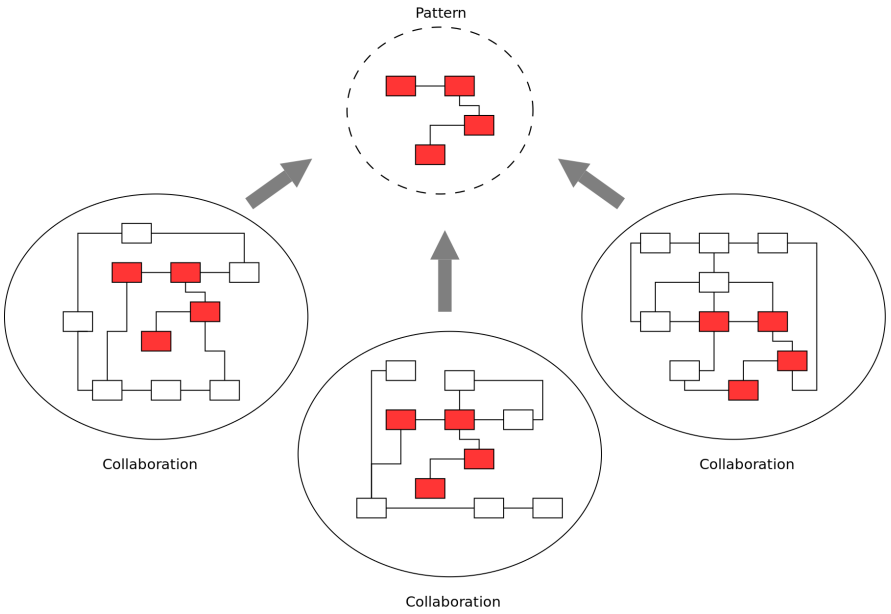


Figure 1.8: Pattern Mining: abstracting a general solution out of different cases, capturing the trade-offs to be optimized.

a domain.[Roberts and Johnson, 1996]. The process of adapting new applications into the framework commonalities, includes many design refactorings [Fowler et al., 1999] and a thoughtful analysis of trade-offs for each design decision. CLAM development has gone though this path, and therefore has been a fruitful source of design experience for pattern mining.

Patterns Evaluation

The patterns are evaluated assessing their usefulness in several use cases in different real-world applications. We show that most of the patterns can be found in different systems, while few other patterns are only found in the CLAM framework. However, we show that the CLAM framework, with it's related applications, has demonstrated its adaptability to several scenarios within the multimedia domain such as real-time spectral audio processing, and 3D-audio rendering based on animated 3D audio scenes and camera tracking metadata.

1.4 Contributions

In the first part of this introduction we have shown a diagram (figure 1.2) depicting the technologies involved in developing multimedia processing systems, in which our research is circumscribed. Now that we have outlined the concrete problems this thesis addresses and the proposed solutions, we recap and show how the addressed problems fit in this general picture of tools and technologies. Figure 1.9 shows the thesis contributions in the big picture of our domain. Main contributions are: the new TTSDF model in the actor-oriented models of computation; the design patterns —which are organized in two category: Actor-Oriented Design Patterns (to implement actor-oriented models) and Framework Design Patterns (to implement dataflow-based frameworks and applications); and the CLAM pattern which uses the aforementioned technologies.

1.4.1 List of Specific Contributions

1. **An actor-oriented model of computation**, in the family of dataflow, that we have named *Time-Triggered Synchronous Dataflow (TTSDF)*. This model overcomes the timeliness limitations of existing dataflow models, and hence, adds real-time capabilities.

   The TTSDF model have the following properties:

   (a) It combines actors associated with time with untimed actors

   (b) It retains token causality and dataflow semantics.

   (c) It is statically (before run-time) schedulable.

   (d) It avoid jitter and uses optimum amount of latency. A superior bound for the latency is given by the model.

   (e) It naturally fits in callback-based architectures.

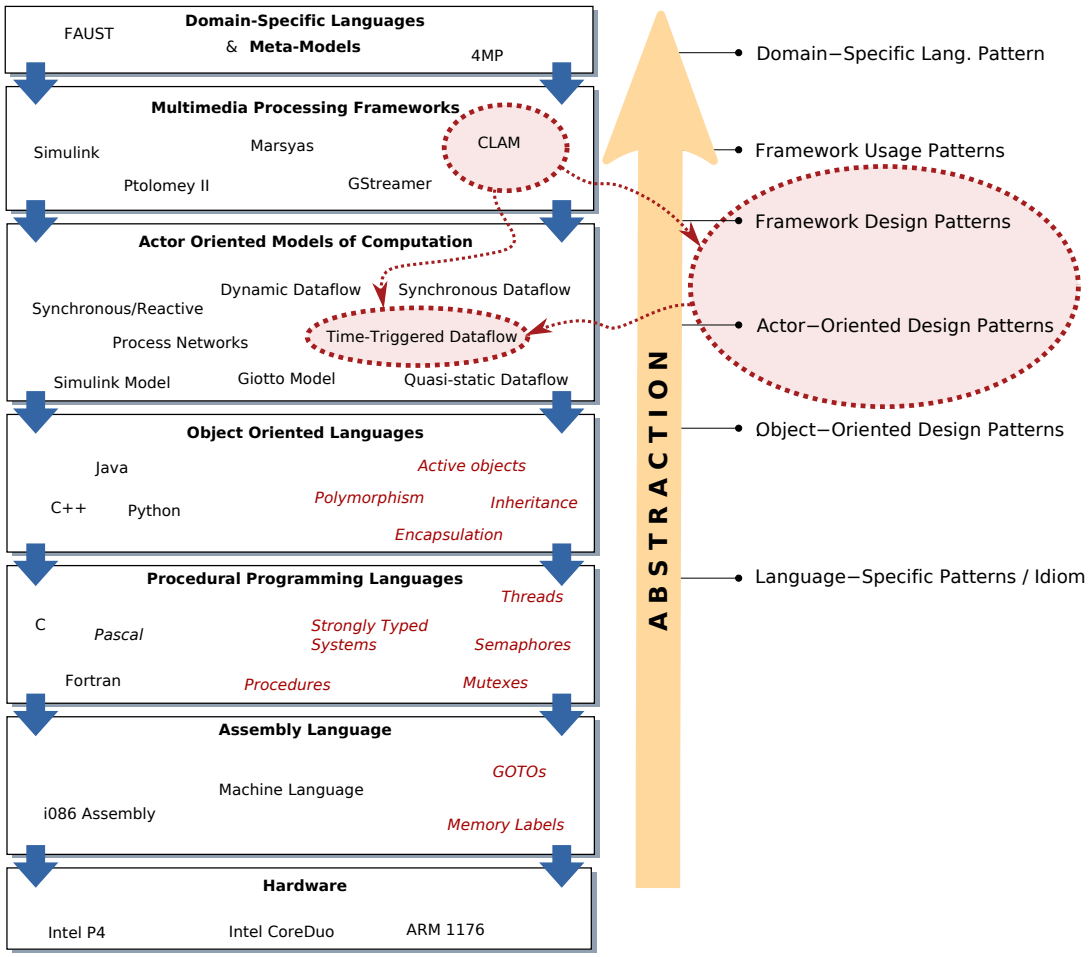


Figure 1.9: The "big picture" of technologies and tools for developing multimedia processing systems, presented before (figure 1.2), now underlining the contributed solutions in dashed circles: *The Time-Triggered Synchronous Dataflow* model, Actor-oriented and Framework-level *design pattern language*, and the *CLAM* framework that provides an example of implementation of the previous technologies. Arrows in dashes denote a relation of "use".

(f) It enables static analysis for optimum distribution of run-time load among callbacks

(g) The scheduling is parallelizable in multiple processors using well known techniques.

(h) The callback-based scheduling algorithm is easily adaptable to other dataflow models (BDF, DDF)

2. Precise semantics for dataflow schedulings in callback-triggered architectures, which can be used in different models of computation.

3. A formal notation for the callback-triggered semantics, based on formal languages. This notation is useful to reason about schedulability of dataflow models.

4. **A design pattern language** for dataflow-based multimedia processing systems. It enables software developers to reuse tested design solutions into their systems, and thus, applying systematic solutions to domain problems with predictable trade-offs to efficiently communicate and document design ideas.

   These characteristics represent a significant departure from other approaches that focus on "code" reuse (such as libraries and frameworks).

   The pattern language, though not comprehensive in all the multimedia processing domain, focuses and is organized in the following four aspects :

   (a) *General Dataflow Patterns:* they address problems about how to organize high-level aspects of the dataflow architecture, by having different types of modules connections.

   (b) *Flow Implementation Patterns:* they address how to physically transfer tokens from one module to another, according to the types of flow defined by the *general dataflow patterns*. Tokens life-cycle, ownership and memory management are recurrent issues in those patterns.

   (c) *Network Usability Patterns:* they address how humans can interact with dataflow networks without compromising the network processing efficiency.

(d) *Visual Prototyping Patterns:* they address how domain experts can generate applications on top of a dataflow network, with interactive GUI, without needing programming skills.

5. Showing that design patterns provide useful design reuse in the domain of multimedia computing.

6. Showing that all the patterns can be found in different applications and contexts.

7. Showing how design patterns are useful to communicate, document and compare designs of audio systems.

8. **An open-source framework** —CLAM— that implements all the concepts and constructs presented in the thesis: the TTSDF model of computation and all the proposed design patterns. CLAM enables multimedia domain experts to quickly prototype (that is: with visual tools, without coding) real-time applications. CLAM is not only a test-bed for theoretic concepts of this thesis, but a tool actually used for real-world multimedia systems (especially in the music and audio sub-domain).

1.5 Thesis Organization

This thesis is structured as follows. Chapter 2 gives the necessary background knowledge on the areas of multimedia-capable operating systems, real-time, actor-oriented design, and object-orientation and design patterns. Chapter 3 introduces the state-of-the-art found in the literature related to the addressed problems. This

chapter discusses extensions the synchronous dataflow model and existing actor-oriented models that support timeliness; it reviews a meta-model for multimedia processing systems that provides semantics for our pattern language; and presents existing patterns and frameworks in our target domain. For each of these state-of-the-art technologies we explain how this thesis innovates upon them or how they address a slightly different problem. Chapter 4 contributes a new actor-oriented model of computation, strongly based on Synchronous Dataflow but suited for real-time processing; its validity is proved and it is related to similar models. Chapter 5 contributes a design pattern language for dataflow-based multimedia processing systems; with use-case studies for qualitative evaluation of the patterns. Chapter 6 presents case studies using the contributed model and pattern language; including the CLAM framework with its rapid prototyping of multimedia applications capabilities. Finally, Chapter 7 draws conclusions and discusses open issues and future lines.

CHAPTER 2

Background

This chapter sets the ground for all the techniques we will use and develop to address the goals and problems before-mentioned in the introduction. Specifically, the work in this thesis is based on the following well-established —though evolving— technologies: operating systems, system-level design and real-time programming, actor-oriented design, design patterns and frameworks.

To start, section 2.1 explains the services offered by *off-the-shelf operating systems*, and how they enable multimedia requirements. We show that these requirements are basically those to process multimedia streams in soft real-time. We also discuss how operating systems and programs deal with such real-time requirements. Key concepts like hard and soft real-time are also presented in this section.

In section 2.2 we present *actor-oriented design*, which can be thought as the "laws of physics" that govern the components (actors) interaction. Actor-oriented design is the conceptual framework within which larger designs are constructed by composing elements. Actor-oriented models abstract (or hide) the complexity of communication, data management, scheduling and hardware resource assignation (such as multiprocessors).

Section 2.3 is about *object-oriented design*; a mainstream programming paradigm in which we later base our design patterns, design examples and code framework. In this section we also introduce the related concept of *frameworks* (see section 2.4.3).

In the following section 2.4, we present *design patterns*, a technique that allows reusing design in an effective way. Design patterns describe an infinite set of similar problems with it's solution. We emphasise the delineation of what is a pattern and what is not. This section also introduces *pattern languages* (see section 2.4.5), a technique to compose individual patterns into a path of design decisions that, used effectively, enables development of complex software systems in a predictable engineering discipline.

2.1 The Multimedia Operating System

Today's operating systems installed in workstations and personal computers (being desktops or laptops) must cope with continuous-media processing. Fortunately, such multimedia systems are outside of traditional (hard) real-time scenarios and have more favorable (soft) real-time requirements.

Though it is fair to say that roughly all modern off-the-shelf operating systems cope with such kind of processing, different operating systems differ greatly on the soft real-time capabilities they offer. It is interesting to note that multimedia processing capabilities are relatively new in off-the-shelf operating systems. The concepts employed by such operating systems were initially developed for embedded real-time systems.

A multimedia processing system (defined in section 1.1.1) reacts to the real world through its hardware components. In the context of an operating system, the

interaction between hardware and software happens through hardware interrupts. Therefore, *interrupt requests (IRQ's)* handling, and process scheduling concerns are paramount in multimedia systems. Other aspects such as resources, files and memory also have a great impact on the ability to deliver continuous media on time.

2.1.1 Multimedia Processing Systems Requirements

The main characteristic of real-time systems is the need for correctness, including both the *result* of the computation and the *time* at which the result is presented to the user. Thus, a real-time system can fail because the system is unable to execute its critical work in time [Stankovic, 1988].

A real-time system adheres to pre-defined time spans for processed data response times. Speed and efficiency are not, as often assumed, the main characteristics of a real-time system. Deterministically timed computation is.

Multimedia must consider timing and logical dependencies among different tasks processed at the same time. In processing of audio and video data the timing relation between the two media has to be considered.

Audio and video streams consist in periodically changing values of continuous media-data such as audio samples or video frames. Each data unit must be presented at a specific deadline. *Jitter* —which is the variation on the time interval at which a periodic task takes place— is allowed only before, not during the final presentation [Steinmetz, 1995]. For example, a piece of music must be played with constant speed or it will be noticed by the human ear. However, users may not perceive a slight jitter at some media presentation, depending on the medium and the application. The human eye is less sensitive to video jitter that the ear is to audio jitter [Steinmetz, 1996].

Latency Bounds

Latency or *delay* is the time span between the instant when a piece of data is made ready to the system's input and the instant when the system makes the result available. Latency can be given in time or in number of sample data or periodic events, provided that the sampling or period rate are known.

When human users are involved just in the input or output of continuous media, delay bounds are more flexible. Consider the playback of video streamed

(a) A laptop-based music performance setup

(b) The Apple's iPhone

(c) A video jockey performance

Figure 2.1: Examples of multimedia processing systems.

from a remote Internet server. The delay of a single video frame transferred is unimportant if all frames arrive in a regular fashion. Users will only notice an initial delay in response to their "start play" commands.

On the other hand, when humans are involved in both the input and the output, the initial delay or latency is important. One example is a software synthesizer in live music performance. Music played by one musician must be made available to all other members of the band within a few milliseconds, or the underlying knowledge of a global unique time is disturbed.

Real-time Requirements

To fulfill the timing requirements of multimedia processing systems, the operating system must use real-time scheduling techniques. Traditional hard real-time for command and control systems used, for example, in areas such as aircraft piloting, demand high security and fault tolerance. Fortunately, the real-time demands of multimedia processing systems are more favorable than hard real-time requirements:

- *Fault-tolerance* is usually less strict: A short-time failure of a continuous-media system, such as a delay in delivering video-on-demand will not directly lead to critical consequences.

- *Missing a deadline*, for many applications in multimedia systems is, though regrettable, not a severe failure. It may even go unnoticed. If a video frame is not prepared on time, it can simply be omitted, assuming this does not

happen for a contiguous sequence of frames. Audio requirements are more stringent.

- *Most of the time-critical operations are periodic*: A sequence of digital continuous-media data results from periodically sampling sound or image signal. Hence, in processing the data units of such a data sequence, all time-critical operations are periodic. Scheduling periodic tasks is much easier than scheduling sporadic ones [Mok, 1983].

2.1.2 Soft vs. Hard Real-Time

Several definitions for real-time systems can be found in the literature. Here we will assume the following definition, which is accordant to the IEEE POSIX Standard [Walli, 1995] :

> A real-time system is one in which the correctness of a result not only depend on the logical correctness of the calculation but also upon the time at which the result is made available.

Therefore there are timing constrains associated to system tasks. Such tasks have normally to control or react to events that take place in the outside world, happening in "real-time".

It is important to note that *real-time computing* is not equivalent to *fast computing*. Fast computing aims at getting the results as quick as possible, while real-time computing aims at getting the results at a prescribed point of time within defined time tolerances. Thus, a *deadline* can be associated with the task that has to satisfy this timing constrain.

If the task has to meet the deadline because otherwise it will cause fatal errors or undesirable consequences, the task is said to be *hard real-time*. On the contrary, if the meeting of the deadline is desirable but not mandatory, the task is *soft real-time* [Steinmetz, 1995].

Given the aforementioned requirements of multimedia processing systems, in the context of such systems we are interested in soft real-time.

Expanding on the question of whether hard real-time is needed, the important point is to ask what is the impact of failure in the case at hand. Will somebody die? Will costly machines or products be destroyed? Will something just require human maintenance? Will some mass-produced product have a slightly higher

defect rate? In other words, we deal with levels of criticality. "Hard" is not an absolute term, just an indication that, unlike "soft", somebody thinks "late" means "unacceptable" as opposed to just "undesirable". But that means that "hard" and "soft" do not express absolute values.

It is interesting to note that time is not a central part of today's computing science and, thus, it is hard to exactly predict how long an operation will take. Lee and Zhao argue in [Lee and Zhao, 2007] that prevailing modern hardware and software techniques play against the very tight constraints required for hard real-time:

> Performance gain in modern processors comes from the statistical speedups such as elaborate caching schemes, speculative instruction execution, dynamic dispatch, and branch prediction. These techniques compromise the reliability of embedded systems. In fact, most embedded processors such as programmable DSP's and microcontrollers do not use these techniques.
>
> [...] Despite considerable progress in software and hardware techniques, when embedded computing systems absolutely must meet tight timing constraints, many of the advances in computing become part of the problem, not part of the solution. Although synchronous digital logic delivers precise timing determinacy, advances in computer architecture and software have made it difficult or impossible to estimate or predict the execution time of software. Moreover, networking techniques introduce variability and stochastic behavior, and operating systems rely on best effort techniques. Worse, programming languages lack time in their semantics, so timing requirements are only specified indirectly.

Real-Time and Multi-Threading Concepts

The following table defines concepts related to real-time operating systems and multi-threading, that are used throughout this chapter.

Term	Definition
Real-time system	A system that can fail not only because of hardware or software failure, but because the system is unable to execute its workload in time. The system must act deterministically, adhering to previously defined time span for data manipulation: that is, it guarantees a response time.
Jitter	Variation on the time interval at which a periodic task takes place.
Latency	Time span between the instant when a piece of data is made ready to the system's input and the instant when the processed data is presented to the user. Latency can be given in time or in number of sample data or periodic events, provided that the sampling or period rate are known.
Race condition	Situation where simultaneous manipulation of a resource by two or more threads causes inconsistent results.
Critical section	Segment of code that coordinates access to a shared resource.
Mutual exclusion	Property of software that ensures exclusive access to a shared resource.
Deadlock	Special condition created by two or more processes and two or more resource locks that keep processes from doing productive work.
Preemption	The act of temporarily interrupting a task being carried out by an operating system, without requiring its cooperation, and with the intention of resuming the task at a later time.
Priority inversion	The scenario where a low priority task holds a shared resource that is required by a high priority task. This causes the execution of the high priority task to be blocked until the low priority task has released the resource.
Interrupt response time	The time between the arrival of the interrupt and the dispatching of the required task, assuming it is the highest-priority task to be dispatched.

2.1.3 Operating Systems Real-Time Facilities

Real-time systems and real-time operating systems are not equivalent concepts. A real-time operating system provides facilities like multitasking scheduling, inter process communication mechanism, etc., for implementing real-time systems. Therefore, although relying on a real-time operating system, there is still a huge potential for applications to fail on fulfilling real-time restrictions.

Such applications have a real-time thread that is typically programmed in a callback scheme. The real-time thread must satisfy the imposed timing constrains. Therefore, real-time programming techniques and specific operating system configurations should be used. The prominent rule is to not allow operations more expensive than $O(n)$, where n is the size of the input data. Hence, threads must be synchronized using *lock-free* operations, most system calls —such as requesting for memory allocation– should be avoided, memory page faults should be avoided by locking enough RAM memory on initialization, and last, task priority must be risen over all non real-time tasks.

Modern multimedia capable operating systems include in general the following real-time features: fast switch context, small size, preemptive scheduling based on priorities, multitasking and multi-threading, real-time timers, and intertask communication and synchronization mechanisms (such as semaphores, signals, events, shared memory, etc.) [Gambier, 2004].

The active community around Usenet's "comp/realtime" newsgoup define the following requirements that make an OS a real-time OS (RTOS)[1] :

1. A RTOS (Real-Time Operating System) has to be multi-threaded and preemptable.

2. The notion of thread priority has to exist as there is for the moment no deadline driven OS.

3. The OS has to support predictable thread synchronisation mechanisms

4. A system of priority inheritance has to exist

5. OS Behaviour should be known

So the following figures should be clearly given by the RTOS manufacturer:

[1] http://www.faqs.org/faqs/realtime-computing/faq/

1. the interrupt latency (i.e. time from interrupt to task run) : this has to be compatible with application requirements and has to be predictable. This value depends on the number of simultaneous pending interrupts.

2. for every system call, the maximum time it takes. It should be predictable and independent from the number of objects in the system;

3. the maximum time the OS and drivers mask the interrupts.

The following points should also be known by the developer:

1. System Interrupt Levels.

2. Device driver IRQ Levels, maximum time they take, etc.

Real-time scheduling

A fundamental part of a multimedia operating system is the task manager. It is composed by the *dispatcher* and the *scheduler*. The dispatcher carries out the *context switch*, that is, the context saving for the outgoing task and the context loading for the incoming task, and the CPU handling to change the active task. The scheduler selects the next task that will obtain the CPU. This choice is given by means of scheduling algorithms, and this is the point where real-time OS's and non-real-time OS's are mostly distinguished. Such algorithms are an active area of research (see [Liu, 2000] and [Stallings, 1998] for an overview of the field).

The multimedia systems in which we are interested must cope with unpredictable workload. That is, the complete information about the scheduling problem (number of tasks, deadlines, priorities, periods, etc.) is not known *a priori*. Therefore, the scheduling must be *dynamic* —done at run-time.

The guarantee that all deadlines are met can be taken as measure of the effectiveness of a real-time scheduling algorithm . If any deadline is not met, the system is said to be *overloaded*. The *total processor utilization* for a set of n tasks is given by

$$U = \sum_{j=1}^{n} \frac{C_i}{min(D_i, T_j)}$$

can be used as schedulability test [Liu and Layland, 1973]. C is the execution time, D the deadline and T the task period. If the task is aperiodic or the deadline is smaller than the period, then the deadline is used in the equation.

Among the most popular algorithms for scheduling tasks with real-time requirements we find the *Rate Monotonic Scheduling (RMS)* and *Earliest Deadline First (EDF)*.

In the RMS approach [Liu and Layland, 1973], each task has a fixed static priority which is computed pre-runtime. The runnable tasks are executed in order determined by their priorities. If all tasks are periodic, a simple priority assignment is done as follows: *the shorter the period, the higher the priority*. This scheduling assumes that all tasks are pre-emptive, periodic, with deadlines equal to the period and independent (that is, there is no task precedence restriction). In this case the total CPU utilization has an upper bound given by

$$U \leq n(2^{\frac{1}{n}} - 1)$$

The RMS algorithm is easy to implement and if the system becomes overloaded, deadlines are missed predictably. The most important drawbacks are its low CPU utilization (under 70%) and the fixed priorities, which can lead to starvation and deadlocks.

The EDF algorithm is based on assigning priorities according to their deadline. The task with the earliest deadline has the highest priority. Thus, the resulting priorities are naturally dynamic. This algorithm was also presented by Liu and Layland and they showed that if all tasks are periodic and preemptive, then the algorithm is optimal [Liu and Layland, 1973]. A disadvantage of this algorithm is that the execution time is not taken into account in the priority assignment.

2.1.4 Operating-System Scheduling vs. Dataflow Scheduling

In this thesis we address the problem of how to schedule *Dataflow* models of computation in a timed manner. Though this problem resembles and relates to the problem of scheduling OS tasks, they differ considerably. In Dataflows models (see section 2.2.2), actors do not need to run separately in individual threads. In fact, it is typical to run a whole model of computation in a single thread. Or partition the model in as many threads as processors available.

The reason behind this is that dataflow actors are not defined with an associated periodic time, but with data precedences declared in the dataflow graph. In order to cope with data dependencies and multi-rate ports, specific schedul-

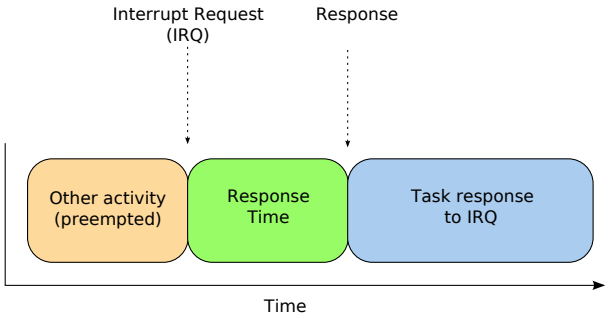


Figure 2.2: Interrupt response time in a preemptable OS.

ing algorithms are needed (see section 2.2.4). Unlike OS scheduling algorithms, dataflow scheduling does not deal with task priorities because equal priority is assumed to all the graph.

In some models, such as the *Synchronous Dataflow* (see section 2.2.3), the scheduling can be done statically before runtime. Another reason to group multiple actors in a single (or few) thread is efficiency since context switching, inter-process communication, and process synchronization take CPU time.

In the proposed *Time-Triggered Synchronous Dataflow* model (in section 4) we relate two different concepts: a periodic real-time task with a deadline (driven by a hardware device IRQ's), with the dataflow scheduling. As a result, the new model provides real-time capabilities and optimal latency to the dataflow execution. In this model, the dataflow scheduling period is interleaved by multiple real-time task executions.

Unlike Dataflow models, the *Process Network* model (see section 2.2.7) uses a more dynamic, nondeterministic approach. Thus, it can benefit form relying on a normal OS scheduler to run each actor as a separate task. In such case, Process Network model copes with its data dependencies (specified by its graph) by means of blocking reads on queues shared between actors.

2.1.5 From Hardware Interrupt to User-Space Response

Off-the-shelf operating systems such as *Mac OS X/BSD, standard Linux or Windows Vista*, clearly separates the *kernel space* and *user space*. Hardware interrupts requests (IRQ's) generated, for example, by audio and video devices and high precision clocks, are first handled by the kernel, but the actual processing must happen in the user space —thus, ensuring the reliability of the system in case of

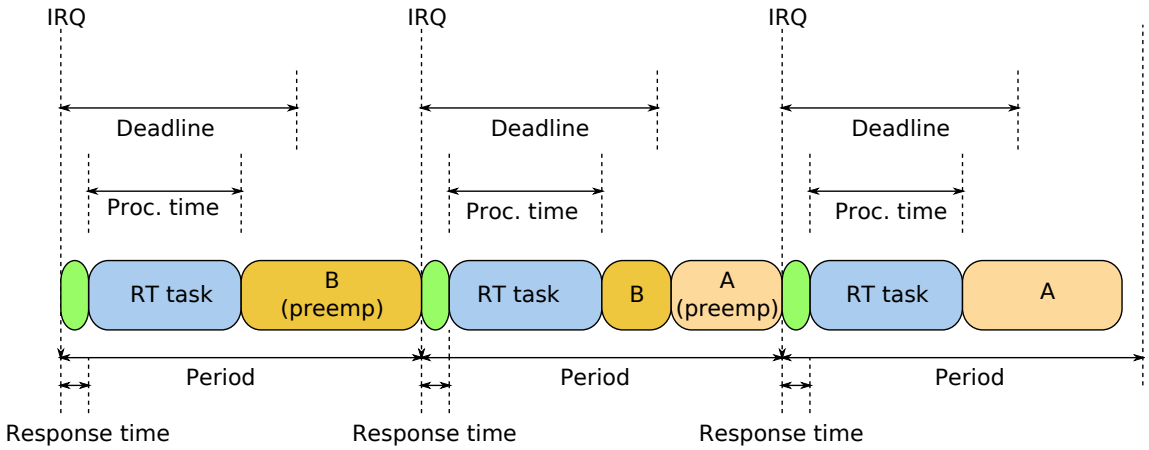


Figure 2.3: Periodic task scheduling of a real-time task on a preemptable OS in heavy load.

application failure.

Figure 2.2 shows that when a IRQ arrives, the CPU is interrupted and enters interrupt handling in the kernel. Some small amount of work is done to determine what event occurred and, after that, the required task in the user space is dispatched (via a *context switch*). The time between the arrival of the interrupt and the dispatching of the required task, assuming it is the highest-priority task to be dispatch, is called *interrupt response time*. For real-time, the response time should be deterministic and operate within a known worst-case time.

In addition to deterministic interrupt processing, task scheduling supporting periodic intervals is also needed for real-time processing. Figure 2.3 shows a periodic task scheduling. Real-time audio and video require periodic sampling and processing. Consider a low-latency audio system that must process samples in periods of 5 ms. Assume that processing those 5 ms of samples takes 3 ms of CPU. The deadline for the processing task is 4 ms (since the audio hardware needs 1 ms to service the internal buffers). For this system to work, the audio processing task must be performed at periodic intervals in the desired deadlines. This means that other tasks must be preempted and the interrupt latency cannot exceed 1 ms.

2.1.6 The Standard 2.6 Linux Kernel

The open-source Linux operating system offers many versions and alternatives on how to achieve real-time. Linux is also attractive because it is easily ported

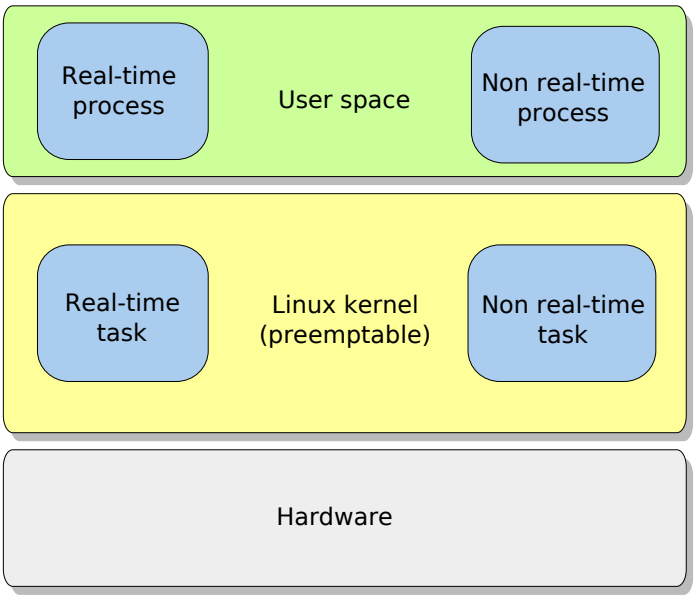


Figure 2.4: The standard 2.6 Linux kernel with preemption

to many architectures including ultra-portable devices such as *ipods* and smartphones. This section explores the solution available today in the standard off-the-shelf 2.6 kernel.

The 2.6 Linux kernel is fully preemptable. In previous versions, when a user space process made a call into the kernel (though a system call), it could not be preempted. This means that if a low-priority process made a system call, a high-priority process had to wait until that call was complete before it could gain access to the CPU. Version 2.6 of the kernel changes this behavior by allowing kernel processes to be preempted if high-priority work is available (even if the process is in the middle of a system call). Figure 2.4 depicts the standard 2.6 Linux kernel with real-time and non-real-time process in both kernel and user space.

This preemptable kernel feature has a trade-off. It enables soft real-time performance even under heavy load, but it does so at a cost: slightly lower throughput and small reduction in kernel performance. Therefore, the kernel can be configured (at compile time) to better fit multimedia desktops or servers.

The 2.6 Linux kernel also provides high-resolution timers and a $O(1)$ scheduler. This scheduler operate in constant time regardless of the number of tasks to execute.

A hard real-time support is possible by applying the *PREEMPT_RT patch* —this version is actually also available off-the-shelf in some popular Linux distributions. The real-time patch provides reimplementation of some of the kernel locking primitives to be fully preemptable, implements priority inheritance for in-kernel mutexes, and converts interrupt handlers into kernel threads so that they are fully preemptable.

2.1.7 Real-time programming styles: Callbacks vs. Blocking I/O

In computer science a callback is executable code of a layer that is passed as an argument to another layer. It allows generic components such an operating-system or a device driver to trigger the needed specific behavior, in a decoupled way (the generic component does not know the specific component).

An interrupt handler, also known as an interrupt service routine (ISR), is a callback subroutine in an operating system or device driver whose execution is triggered by the reception of an interrupt. In a multimedia processing environment an interrupt is generated when new input data is ready to be processed, and output data needs to be serviced. The interrupt causes a call to a callback function that the user has previously registered, Such interrupt handlers are also useful for dealing with the transitions between protected modes of operation such as hardware interrupt calls and user-space processes.

In the blocking input/ouput style of programming adds an extra layer. The user real-time processing code is not triggered by a callback, but is organized in a main loop provided by the user. In this style, the communication with the input and output data is done through blocking read and write calls —that is, when such call is done the multimedia process will stall till the hardware is ready to service the input or output.

The callback style is preferred to the blocking IO style, for real-time applications [Bencina and Burk, 2001]. However, blocking IO might be simpler to understand, and so, useful for pedagogical purposes. The main benefit of callback versus blocking I/O is that the multimedia processing automatically takes place in a high-priority thread (an interrupt-triggered thread), and, in respect to the main application, this thread runs in background.

In the blocking I/O approach, on the contrary, strong multi-tasking support

is needed in order to run the multimedia process in high-priority. Even when this support is available, the run-time performance is penalized by the extra context-switching among threads, inter-process communication and I/O buffers management. Thus, it is not strange that most of the multimedia API's are callback-based only.

2.2

Actor-Oriented Design

The term "actors" was introduced in the 1970's by Carl Hewitt and others at MIT to describe autonomous reasoning agents. They developed basic techniques for constructing systems based on *asynchronous message passing*, instead of applicative evaluation, as in lambda calculus [Hewit, 1977, Hewitt and Baker, 1977].

The term evolved through the work of Gul Agha and others to refer to a family of *concurrent models of computation*, independently of whether they were being used to realize autonomous reasoning agents [Agha, 1986].

The term "actor" has also been used since 1974 in the dataflow community in the same way, to represent a concurrent model of computation. More recent work by Eker, Lee and others at UC Berkeley [Eker et al., 2003] focuses on the use of patterns of message passing between actors. These are called *models of computation*, and provide interesting modeling properties.

The term "actor" also has some pitfalls: the most prominent is that actor collides with the Unified Modeling Language (UML) "actor", with a totally different meaning (a prototypical user in UML use cases) [Larman, 2002].

2.2.1 Actor-Oriented Models of Computation

An *actor* is a unit of functionality. Actors have a well-defined interface that abstracts internal state and execution, and restricts how an actor interacts with its environment. This interface includes *ports* that represent points of data communication between actors, and *configuration parameters* which are used to configure the behavior of an actor.

An important concept in actor-oriented design is that internal behavior and state are hidden behind the actor interface and not visible externally. This *strong encapsulation* separates the behavior of a component from the interaction of that component with other components.

Connections between actor ports represent communication *channels* that pass data *tokens* from one port to another. Actors are composed with other actors to form *composite actors* or *models*. The exact semantics of the composition and the communication style is determined by a *model of computation*. Since the processing system can be modeled with a mathematical graph —with nodes being actors and arcs communication channels—, actor-oriented models of computation are also known as *graphical models of computations*.

One of the most flexible ways to specify actor behavior is to embed the specification within a traditional programming language, such as C or C++, and use special purpose object-oriented programming interfaces for specifying ports and sending and receiving data. The C++ code in listing 2.1 gives an example of such specification in an object-oriented language. The external interface is often drawn as a box with inlets and outlets. Figure 2.5 is the visual representation of the same interface. This technique (i.e. actor behavior defined with a standard programming language) has been widely used in actor-oriented design [Buck and Vaidyanathan, 2000], since it allows for existing code to be reused, and for programmers to quickly start using actor-oriented methodologies.

Many actor-oriented models of computation exist for different purposes and with different features. Examples of such models are: Queuing Models, Finite State Machines, State Charts, Petri Nets, Processing Networks and Dataflow Networks. Each of these has many variants. The Ptolemy project[2] is interesting in this context because it defines and implements many of these actor-oriented

[2]The Ptolemy project include the following actor-oriented models of computation: Component Interaction, Communicating Sequential Processes, Continuous Time, Discrete Events, Distributed Discrete Events, Discrete Time, Synchronous Reactive, and Timed Multitasking.

```
class MyProcessing : CLAM::Processing
{
  InPort<TokenType> _input1;
  InPort<TokenType> _input2;
  OutPort<TokenType> _output;

public:
  MyProcessing() : _input1("Input 1",this), _input2("Input 2", this),
            _output("Output", this)
  {
  }
  void Do() // just add inputs
  {
      _output.produce( _input1.consume() + _input2.consume() );
  }
  ...
};
```

Listing 2.1: Actor interface and functionality specification in C++, using the CLAM framework (simplified for clarity sake)

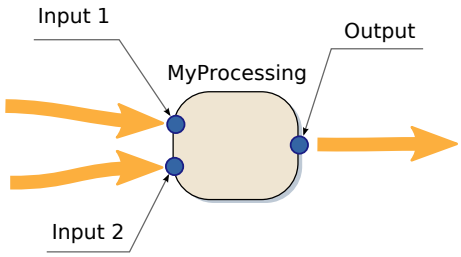


Figure 2.5: A visual representation of the actor interface specified in C++ in listing 2.1

models [Hylands et al., 2003].

The selection of an actor-oriented model of computation (or set of heterogeneous models) depends on the purpose and requirements of the system to be developed. This requirements are generally constrained by the application domain. The multimedia processing systems domain, for example, will generally benefit from Dataflow and Process Network models — while, for example, control-intensive applications will benefit from Finite State Machine models.

An essential difference between models of computation is their modeling of time. Some are very explicit by taking time to be a real number that advances uniformly. Others are more abstract and take time to be discrete. Others take time to be merely a constraint imposed by causality (or data dependency).

Many actor-oriented frameworks and description languages exist. Examples include hardware design languages (like VHDL [Perry, 1993] and Verilog [Lawrence, 2003]), coordination languages [Papadopoulos and Arbab, 1998], synchronous languages [Benveniste et al., 2003] and frameworks like Giotto [Henzinger et al., 2003]; the system description language SystemC [Aynsley and Long, 2005]; SHIM [Edwards and Tardieu, 2005]; Simulink [Dabney and Harman, 2001] and Real-Time Workshop by MathWorks; the LabView [Johnson and Jennings, 2001] graphical development platform from National Instruments; Ptolemy II [Hylands et al., 2003] heterogeneous model framework from University of California, Berkeley; the Generic Modeling Environment [Ledeczi et al., 2001]; Lucid, a dataflow programming language [Wadge and Ashcroft, 1985], where variables and expressions denotes streams; Erlan, a general purpose concurrent and functional programming language developed in Ericsson [Armstrong et al., 1996] to support soft real-time applications; and many more.

Examples of actor-oriented specific to multimedia, that allows developers to (sometimes visually) build audio and/or video processing systems are: in the proprietary arena: Microsoft's Filter Graph [Gray, 2003], and the Java Media Framework [Gordon and Talley, 1999]; and in the open-source arent: GStreamer [Taymans et al., 2008], Marsyas [Tzanetakis, 2008], CLAM (presented in section 6.1), CSound, developed originally from MIT, that uses the *orchestra* and *score* paradigm for defining both the sound synthesis and the score of a music piece [Boulanger et al., 2000], and Faust, a functional programming language for audio [Orlarey et al., 2004].

Many actor-oriented frameworks offer a visual syntax and tools to manipulate models. Such visual tools are not necessary to specify a model, since textual based approaches exist, but they have the benefit of clearly showing the departure from sequential control (or imperative) programming paradigm. In actor-oriented diagrams concurrency, for instance, is made explicit, which does not happen in textual specifications. Figure 2.6 shows screen-shots of a couple of such visual tools.

For the purpose of this thesis, we now focus on Dataflow models of computations, and the closely related Process Networks model.

2.2.2 Dataflow Models of Computation

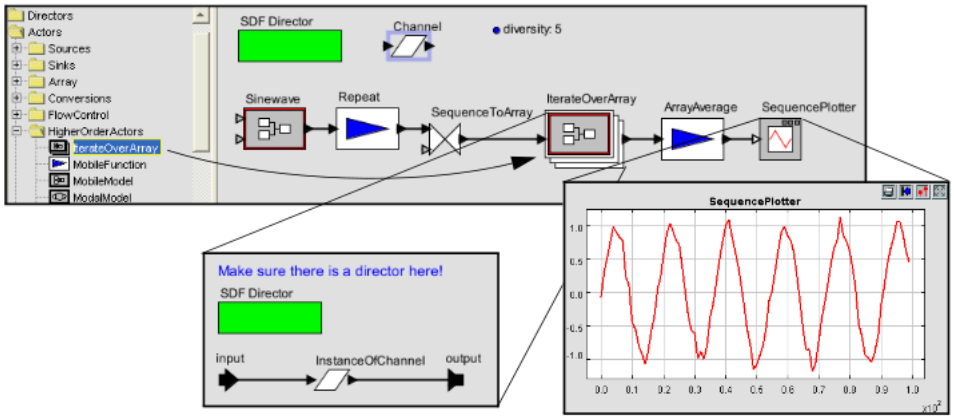
The term *dataflow* —equivalent to *dataflow network*— has been and still is often used with loose semantics. We will adhere to the precise definition given by Lee and Parks [Lee and Parks, 1995], who formalized the semantics outlined by Dennis in [Dennis, 1974].

Dataflow denotes not a single model of computation but a family of models. They all share the following properties: the model consists in a directed graph whose nodes are actors that communicate exclusively through the channels represented by the graph arcs. Conceptually, each channel consist of a *first-in-first-out (FIFO)* queue of tokens. Actors expose how many tokens they need to consume and produce to each channel on the next execution. This number of tokens is represented by an integer value associated to each port and is known as *port rate* (but also *token rate* in some literature). A dataflow actor is only allowed to run when tokens stored in the input channels queues are sufficient.

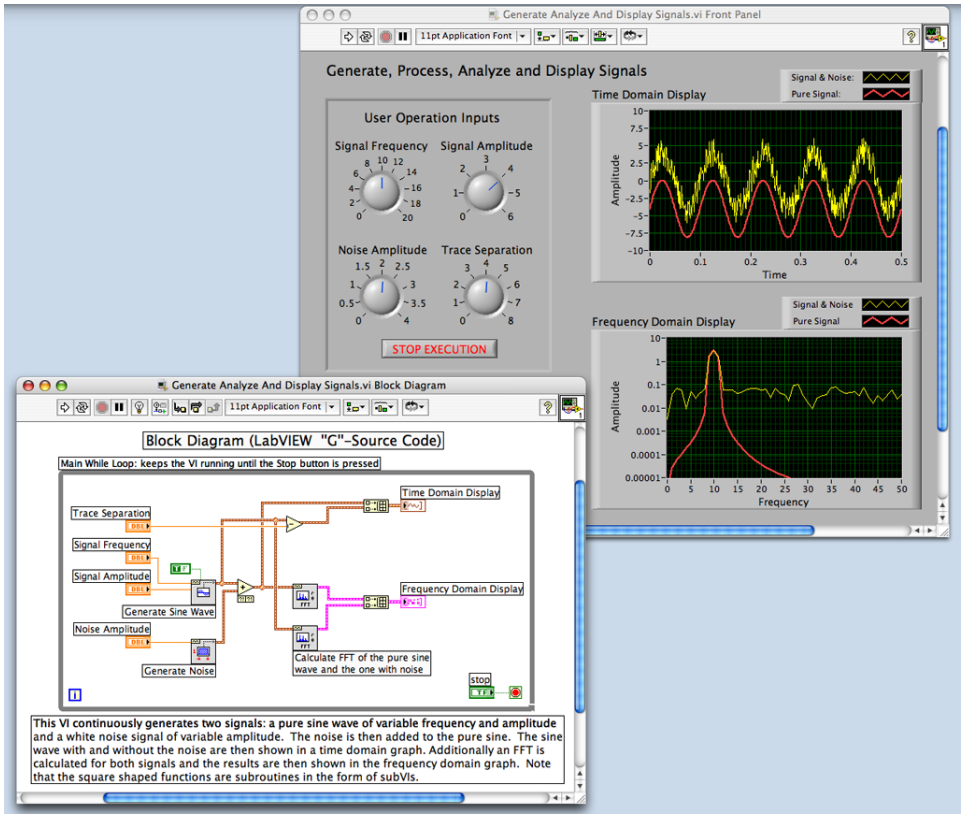
These message queues desynchronize the communication between actors, allowing the sending actor to continue concurrently without waiting for the message to be received. At the same time, message queues ensures that messages are received in order of transmission with no message loss.

As a consequence, dataflow actors can be executed in any order —provided that it does not result in negative buffer sizes— and the produced result will not be affected.

Dataflow models of computation are appealing since they closely match a designer's conceptualization of a multimedia processing system as a block diagram. Additionally, they offer opportunities for efficient implementation both in concurrent or sequential schedulings. This is specially true for statically schedulable



(a) The Ptolemy II framework supporting heterogeneous models of computation, from UC Berkeley



(b) The LabView development platform, from National Instruments

Figure 2.6: Examples of actor-oriented frameworks with visual syntax and tools

dataflows, which allow efficient code generation or *synthesis* for a whole scheduling cycle.

Given that actors only communicate through ports and do not share state, system parallelism is explicitly exposed by the graph. However, parallel implementation is not required.

Firing Rules

In general, a dataflow actor may have more than one firing rule. The evaluation of the firing rules is sequential in the sense that rules are sequentially evaluated until at least one of them is satisfied. Thus, an actor can only fire if one or more of its firing rules are satisfied. In the most typical models, though, actors have a single firing rule of the same kind: a number of tokens that must be available at each of the inputs. For example, an adder with two inputs has a single firing rule saying that each input must at least have one token.

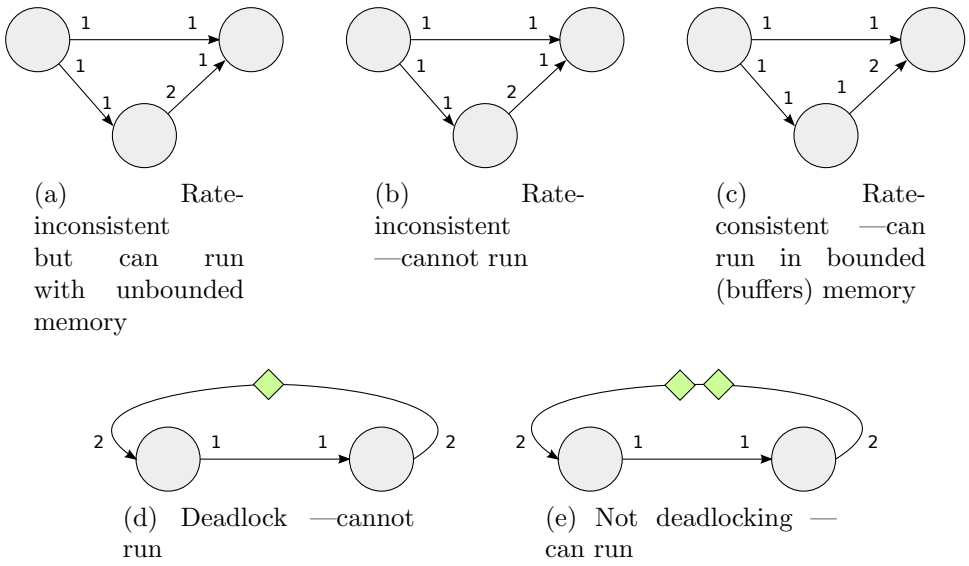
The constrains on the firing rules is what differentiates different dataflow models. For example, in *Dynamic Dataflows* (DDF), token rates related to an actor are allowed to change after each execution. In *Static Dataflows* token rates are specified *a priori* and cannot change during run-time. *Boolean-controlled Dataflows* take a middle ground and allow changing certain token rates in special actors.

As pointed out by [Parks, 1995] breaking down processes into smaller units such as dataflow actor firings, makes efficient implementations possible, allowing better scheduling and parallelization. Moreover, restricting the type of dataflow actors to those that have a predictable, or fixed, token rates makes it possible to perform static, off-line analysis to bound the memory usage.

Dataflow graphs have data-driven semantics. The availability of operands enables the operator and hence sequencing constraints follow only from data availability. The principal strength of dataflow networks is that they do not over-specify an algorithm by imposing unnecessary sequencing constraints between operators [Buck and Lee, 1994].

2.2.3 Synchronous Dataflow Networks

Synchronous Dataflow Networks (SDF) is a special case of Dataflow Network in which the number of tokens consumed and produced by an actor (token rates) is known *a priori*, before the execution begins. Therefore, the consuming and



(a) Rate-inconsistent but can run with unbounded memory

(b) Rate-inconsistent —cannot run

(c) Rate-consistent —can run in bounded (buffers) memory

(d) Deadlock —cannot run

(e) Not deadlocking — can run

Figure 2.7: Runnability of Synchronous Dataflows. Rate-inconsistency can lead to non runnable or runnable with unbounded memory. Feedback loops can lead to deadlock if not enough initial delays.

produced rates of each actor repeat every time the actor is fired.

Since token rates are known a priori, the SDF model is statically (before runtime) schedulable. For that reason the SDF is also known as *Static Dataflow*. Here we will adhere to the "Synchronous Dataflow" term given by Lee in its original formalization [Lee and Messerschmitt, 1987b]. "Synchronous" refers to the fact that each node firing in a fixed rate.

Formally, an SDF graph has its arcs tagged with two values: the consuming and producing token rates. Arcs can have initial tokens. Every initial token represents an offset between the token produced and the token consumed at the other end. Each unit delay (or token) is represented by a diamond in the middle of the arc, or a similar notation.

The SDF model suits many multimedia and signal-processing domains, which are fixed-rate by nature. —where "fixed" rate does not imply "single" rate. Moreover, more dynamic and complex system can be achieved by composing efficient SDF subsystems using Process Networks or Dynamic Dataflow models as a coordinating model.

This fixed-rate restriction limits the model expressiveness but, on the other hand, many questions are decidable, like its calculability —that is, the ability

to run— and a bound for the memory usage —in the connections buffers. The periodic or *cyclic scheduling* can be found *statically*, before run-time; therefore, all the scheduling overhead evaporates. Furthermore, it enables optimized embedded software synthesis [Edwards et al., 2001].

It is important to note that we are interested in cyclic schedulings that operate with bounded memory. Therefore, the scheduling analysis should detect when the graph is defective. Defects are related either to being *rate-inconsistent*, causing an ever-increasing (unbounded) memory, or inability to run; or to deadlock (starvation) caused by lack of initial buffering. Examples of each defect type are given in figure 2.7.

A *static schedule* consists of a finite list of actors, in the case of a sequential schedule, and N finite lists —with additional synchronization information— in the case of a N parallel schedule. In any case the schedule is *periodic*. An important property of an admissible schedule is that the buffering state (that is, the size of each FIFO queue) after a whole period has been executed exactly matches the initial state.

2.2.4 Static Scheduling of Dataflow Graphs

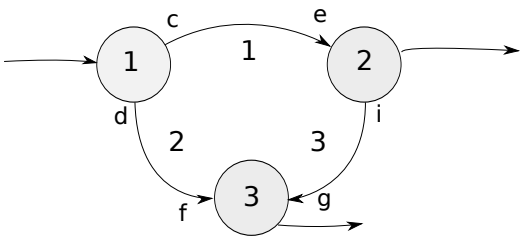


Figure 2.8: An SDF graph showing the amount of tokens consumed and produced for each node and the nodes and arcs numbering.

The static scheduler algorithm has two phases: the first one consists on finding how many times each actor must run in one period, and the second one consists on simulating an execution of a period.

The next paragraphs detail these two phases taking the graphs in Figure 2.8 and 2.9 as examples.

As noted above, buffer sizes must reach the initial state after executing a whole period. As a consequence we can write that, within a cycle, the total number of

tokens produced into any queue must balance the total number of tokens consumed from that queue; and this is precisely captured by the *balance equations*.

Let $\vec{q}$={x, y, z} be the number of executions per period of nodes in a three node graph like the one depicted in figure 2.8. We can write the following balance equations:

$$\begin{aligned} x * c &= y * e \\ x * d &= z * f \\ y * i &= z * g \end{aligned} \tag{2.1}$$

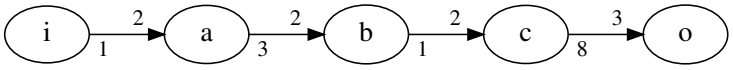
If a non-zero solution for x, y, z exists (that is, *none* of the variables is zero), we say that the SDF is *rate-consistent*. Else, it is *rate-inconsistent* and a schedule does not exist. Such inconsistency is easy to grasp intuitively, as we can think of it as the problem of balancing the incoming flow and outgoing flow for each queue. See examples of rate-inconsistent graphs in figure 2.7.

It can be easily proved that if a non-zero solution exists, an integer solution also exists (and not only one but an infinite number of them). Solving the balance equations is equivalent to finding a vector in the *null-space* of the graph *topology matrix*. The topology matrix is similar to the incidence matrix in graph theory and is constructed as follows: all node and arcs are first enumerated, the (i, j)th entry in the matrix is the amount of data produced by node j on arc i each time it is invoked. If node j *consumes* data from arc i, the number is negative, and if it is a connected to arc i, the number is zero. Therefore we assign a column to each node and a row to each arc. The SDF graph depicted in figure 1 has the following topology matrix:

$$\Gamma = \begin{bmatrix} c & -e & 0 \\ d & 0 & -f \\ 0 & i & -g \end{bmatrix} \tag{2.2}$$

The second phase of the static scheduling algorithm consists on simulating a cycle execution. Iteratively, all nodes are checked for their runnability (that is, whether they have enough tokens in their inputs), and, if runnable, they are scheduled. This goes on until the number of executions $\vec{q}$ found in the previous phase are completed. It is proved [Lee and Messerschmitt, 1987a] that if an admissible schedule exists, this straightforward strategy will find such a schedule. It is interesting to note that any rate-consistent graph will have a schedule unless the graph

has loops, and there is a lack of initial delays in the looping arcs.



$$\Gamma = \begin{bmatrix} -2 & 0 & 0 & 1 & 0 \\ 3 & -2 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 \\ 0 & 0 & 8 & 0 & -3 \end{bmatrix}$$

- Period executions: $\vec{q}$={8,4,6,3,8}. Nodes order: *i, a, b, c, o*
- SDF periodic schedule: $\mathbf{i}_0$, $\mathbf{i}_1$, a_0, b_0, $\mathbf{i}_2$, $\mathbf{i}_3$, a_1, b_1, c_0, $\mathbf{i}_4$, $\mathbf{o}_0$, b_2, $\mathbf{i}_5$, $\mathbf{o}_1$, a_2, b_3, c_1, $\mathbf{i}_6$, $\mathbf{o}_2$, $\mathbf{o}_3$, $\mathbf{i}_7$, a_3, b_4, $\mathbf{o}_4$, b_5, c_2, $\mathbf{o}_5$, $\mathbf{o}_6$, $\mathbf{o}_7$

Figure 2.9: A simple SDF graph performing sampling-rate conversions, its topology matrix and its non-trivial scheduling

Figure 2.9 shows the result of an SDF scheduling. It can be noted how using different port-rates implies having a longer cyclic scheduling. Also note how execution of inputs and outputs (nodes **i** and **o**) runs in bursts and not in constant frequency. This is problematic for real-time requirements, and we have solved this issue in the proposed *Time-Triggered SDF* model, described in chapter 4.

The SDF can be generalized with an additional parameter T (threshold) associated with each arc, that specifies the number of tokens needed in the arc before the node can be fired. Of course, $T \geq W$ where W is the port-rate. This type of SDF is also known as *Computation Graph*. Questions of termination and boundedness are solvable for Computation Graphs.

2.2.5 Boolean-controlled Dataflow

Although SDF is adequate for representing large parts of systems, sometimes it does not fit for representing an entire program. A more general model is needed to represent data-dependent iteration, conditionals and recursion. We can generalize synchronous dataflow to allow conditional, data-dependent execution and still use the balance equations. *Boolean-controlled Dataflow (BDF)* is an extension of Synchronous Dataflow that allows conditional token consumption and production.

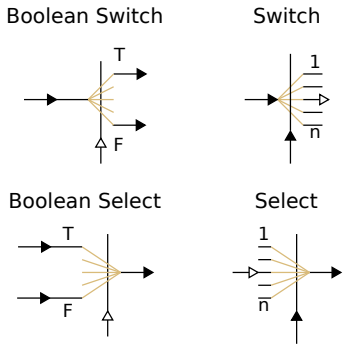


Figure 2.10: The "Switch" and "Select" actors of Boolean-controlled Dataflows

By adding two simple control actors —Switch and Select— we can build conditional constructs like if-then-else and do-while loops. In the Ptolemy II framework, the Switch and Select actors are represented as shown in figure 2.10. The Switch actor gets a control token and then copies a token from the input to the appropriate output, determined by the boolean value of the control token. The Select actor gets a control token and then copies a token from the appropriate input, determined by the boolean value of the control token, to the output. These actors are not SDF actors because the number of produced/consumed tokens is not fixed and depends on an input boolean control [Buck and Lee, 1994].

Switch and Select actors are usually used in well-behaved patterns or schemas, such as conditionals and loops. Using these patterns, finite execution schedules can be found and complete cycles are guaranteed to execute in bounded memory [Gao et al., 1992]. However, in general, models are not guaranteed to execute in bounded memory, or in finite time —that is, a cyclic scheduling cannot be found—[Buck, 1993]. For example, figure 2.11 depicts a model that might require unbounded memory and unbounded time.

It is interesting to note that the general problem of determining whether a BDF graph can be scheduled with bounded memory is undecidable (equivalent to the halting problem); this is because BDF graphs are Turing-equivalent [Buck, 1993].

2.2.6 Dynamic Dataflow

Dynamic Dataflows are Dataflows with actors for which the number of tokens consumed and produced by a firing cannot be specified statically (before run-

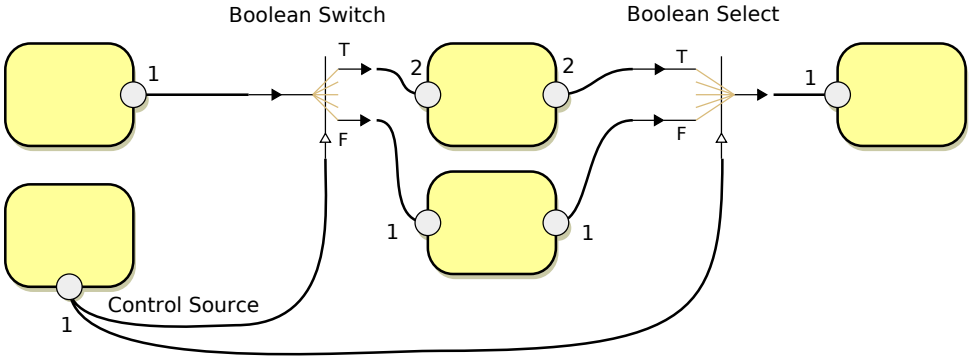


Figure 2.11: A Boolean-controlled Dataflow model where a complete cycle might require unbounded memory and unbounded time. In an execution where *ControlSource* produces a single *true* followed by many *false* tokens, the output of the middle-bottom actor will accumulate at the input to the Select actor.

time). This definition imply that Boolean-controlled Dataflows are a subclass of Dynamic Dataflows.

Unlike Boolean-controlled Dataflows, the general Dynamic Dataflow model of computation only uses run-time (dynamic) analysis. Thus, it makes no attempt to statically answer questions about deadlock and boundedness [Parks, 1995].

2.2.7 Process Networks

The *Process Networks (PN)* —also known as *Kahn Process Networks*— [Kahn and MacQueen, 1977, Geilen and Basten, 2003] is very much related to Dataflows. Process Networks, however, do not need to expose the token rate of their processing blocks. Each actor (in this context also known as "process") runs in a separate process (at least conceptually) which is orchestrated by a scheduler similar to those found in general purpose operating systems. Each process communicates through unidirectional FIFO channels, where writes to the channel are non-blocking, and reads are blocking.

Dataflows are a special case of PN where explicit fire rules exists. Compared to PN, Dynamic Dataflows consists of repeated "firings" of dataflow actors. An actor defines a (often functional) quantum of computation. By dividing processes into actor firings, the considerable overhead of context switching incurred in most implementations of Process Networks is avoided [Lee and Parks, 1995]. This suggests that the granularity of the processes in PN's should be relatively large. For

Dataflows, on the contrary, the cost can be much lower since no context switch is required in general, and hence the granularity can be smaller.

Kahn and MacQueen propose an implementation of Process Networks using multitasking with a primarily demand-driven style [Kahn and MacQueen, 1977]. A single "driver" process (one with no outputs) demands inputs. When it suspends due to an input being unavailable, the input channel is marked "hungry" and the source process is activated. It may in turn suspend, if its inputs are not available. Any process that issues a "put" command to a hungry channel will be suspended and the destination process restarted where it left off, thus injecting also a data-driven phase to the computation. If a "get" operation suspends a process, and the source process is already suspended waiting for an input, then deadlock has been detected.

Scheduling can be classified as *data-driven* (*eager* execution), *demand-driven* (*lazy* execution) or a combination of the two. In eager execution a process is activated as soon as it has enough data as required by any of its firing rules. In lazy execution a process is activated only if the consumer process does not have enough data tokens. When using bounded scheduling (see [Parks, 1995]) three rules must be applied: (a) a process is suspended when trying to read from an empty input, (b) a process is suspended when trying to write onto a full queue and (c) on artificial deadlock, increase the capacity of the smallest full queue until its producer can fire.

2.2.8 Context-aware Process Networks

A special kind of Process Network introduced as an extension to the basic model that is interesting for our purposes is that of Context-aware Process Networks [van Dijk et al., 2002]. This new model emerges from the addition of asynchronous coordination to basic Kahn Process Networks so processes can immediately respond to changes in their context. This situation is very common in embedded systems.

In Context-aware Process Networks, stream oriented communication of data is done through regular channels but context information is sent through unidirectional register links (REG). These links have destructive and replicative behavior: writing to a full register overwrites the previous value and reading from a register returns the last value regardless if it has been read before or not. Thus, register links are an event-driven asynchronous mechanism. As a consequence, the be-

havior of a Context-aware Process Network depends on the applied schedule or context.

A simple example of a system that can be effectively modeled by a Context-aware Network is a transmitter/receiver scheme in which the receiver needs to send information about its consumption rate to the transmitter so transmission speed can be optimized. The basic transmitter/receiver scheme can be implemented with a Kahn Process Network but in order to implement feedback coordination we need to use the register link provided by Context-ware process networks.

Context-aware Network systems are indeterminate by nature. Unless the indeterminate behavior can be isolated, a composition of indeterminate components becomes a non-deterministic system, which is possible but not practical. Nevertheless as mentioned in [van Dijk et al., 2002] some techniques can be used in order to limit indetermination.

The "Context-aware" property is not only applicable to Process Networks, but also applicable to Dataflow models.

2.2.9 Petri Nets

Prior to the development of the actor-oriented models, *Petri Nets* were widely used to model concurrent computation [Murata, 1989]. However, they were acknowledged to have an important limitation: they modeled control flow but not data flow. Consequently they were not readily composable, and hence, limiting their modularity.

Hewitt pointed out another difficulty with Petri Nets: simultaneous action. That is, the atomic step of computation in Petri Nets is a transition in which tokens simultaneously disappear from the input places of a transition and appear in the output places. The physical basis of using a primitive with this kind of simultaneity seemed questionable to him. Despite these apparent difficulties, Petri Nets continue to be a popular approach to modelling concurrency, and are still the subject of active research.

2.3

Object Oriented Technologies

Booch defines Object-oriented programming as "a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships." [Booch, 1994]

Objects act on each other, as opposed to a traditional view in which a program may be seen as a collection of functions, or a list of instructions. Each object is able to receive messages, process data and send messages to other objects. It is interesting to note that this definition of object is similar with the given definition of actor. However, a fundamental difference exist: in an object-orientated collaboration what flows is sequential control (thus the sequence of executed code is make explicit by the program), whereas in an actor-oriented collaboration what flows is streams of data, leaving the decision of when to execute actors to the (actor-oriented) model of computation.

An *object* is a real-world or abstract entity made up of an identity, a state, and a behavior. A *class* is an abstraction of a set of objects that have the same behavior and represent the same kind of instances. The object-oriented paradigm can be deployed in the different phases of a software life-cycle and the *UML* language supports most of the activities contained in them.

An object-oriented language supports two characteristic features: encapsulation and inheritance. Abstraction is the process of identifying relevant objects in the application and ignoring the irrelevant background. Abstraction delivers reusability and information hiding through encapsulation. Encapsulation consists in hiding the implementation of objects and declaring publicly the specification of their behavior through a set of attributes and operations. The data structures and methods that implements these are private to the objects.

Object types or classes are similar to data types and to entity types with encapsulated methods. Data and methods are encapsulated and hidden by objects.

Classes may have concrete instances, also known as objects.

Inheritance is the ability to deal with generalization and specialization or classification. Subclasses inherit attributes and methods from their super-classes and may add others of their own or override those inherited. In most object-oriented programming languages, instances inherit all and only the properties of their base class. Inheritance delivers extensibility, but can compromise re-usability.

Polymorphism —having many forms— is a fundamental capability of object-orientation. In dynamic polymorphism —the more common form of polymorphism— variables can refer to instances of different classes during run-time.

The benefits arising from the use of objects technology are summarized by Graham in [Graham, 1991]

> Reusability, extensibility and semantic richness. Top-down decompositon can lead to application-specific modules and compromise reuse. The bottom-up approach and the principle of information hiding maximize reuse potential. Encapsulation delivers reuse.
>
> Polymorphism and inheritance make handling variation and exceptions easier and therefore lead to more extensible systems. The open-closed principle is supported by inheritance. Inheritance elivers extensibility but may compromise reuse.
>
> Semantic reichness is provided by inheritance and other natural structures, together with constraints and rules concerning the meaning of objects in context. This also compromises reuse and must be carefully managed.

2.3.1 Frameworks

Frameworks are reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate. A good framework can reduce the cost of developing an application by an order of magnitude because it lets the developer reuse both design and code. They do not require new technology, because they can be implemented with existing object-oriented programming languages.

Apart from implementing the aspects that are common across an entire domain, frameworks must expose those elements in the domain that vary between

one application and another, as extension points.

Developing good frameworks is expensive. A framework must be simple enough to be learned, yet must provide enough features so that it can be used quickly and hooks for features that are likely to change. It must embody a theory of the problem domain, and is always the result of domain analysis, whether explicit and formal, or hidden and informal [Roberts and Johnson, 1996].

Therefore, frameworks should be developed only when many applications are going to be developed within a specific problem domain, allowing the time savings of reuse to recoup the time invested to develop them.

In multimedia processing, frameworks typically implement a single actor-oriented model of computation. In some cases, they have domain-specific languages implemented on top.

2.4 Design Patterns

When code reuse is not feasible engineers have no choice but to fall back to ad-hoc or creative solutions. In such cases engineers tend to reuse similar solutions that worked well for them in the past, and, as they gain more experience, their repertoire of design experience grows and they become more proficient. Traditionally, this design reuse was usually restricted to personal experience and there was little sharing of design knowledge among developers [Beck et al., 1996].

Instead of reusing code, engineers might take advantage of reusing design. *Design patterns*, introduced by [Gamma et al., 1995] is a software engineering technique that allows effectively recording best design practices.

A popular definition for patterns is the one given by Christopher Alexander "A pattern is a solution to a problem in a context.". However, Vlissides points

out [Vlissides, 1998] three relevant things are missing from this definition:

1. *Recurrence*, which makes the solution relevant in situations outside the immediate one.

2. *Teaching*, which gives you the understanding to tailor the solution to a variant of the problem. (Most of the teaching in real patterns lies in the description and resolution of forces, and/or the consequences of its application.)

3. A *name* by which to refer to the pattern.

2.4.1 A Brief History of Design Patterns

In the 1960's, building architects were investigating automated, computerized building design. The mainstream of this movement was known as modular construction, which tries to transform requirements into a configuration of building modules using computerized rules and algorithms. The architect Christopher Alexander broke with this movement, noting that the great architectures of history where not made from rigorous, planned designs, but that their pieces were custom-fit to each other and to the building's surroundings. He also noted that some buildings were more aesthetically pleasing than others, and that these aesthetics were often attuned to human needs and comforts. He found recurring themes in architecture, and captured them into descriptions (and instructions) that he called patterns and pattern languages [Alexander, 1977]. The term "pattern" appeals to the replicated similarity in a design, and in particular to similarity that makes room for variability and customization in each of the elements. "Thus *Window on Two Sides of Every Room* is a pattern, yet it prescribes neither the size of the windows, the distance between them, their height from the floor, nor their framing (though there are other patterns that may refine these properties)." [Coplien, 1998]

Over the decade of the 1990's, software designers discovered analogies between Alexander patterns and software architectures. The first work on design pattern had their origin in the late 1980's when Ward Cunningham (the father of the wiki) and Kent Beck (best known for its extreme programming agile methodology) documented a set of patterns for developing elegant user interfaces in Smalltalk[Beck, 1988]. Few years later, Jim Coplien developed a catalog of language-specific C++ patterns called *idioms*. Meanwhile, Erich Gamma

collected recurring design structures while working on the ET++ framework [Weinand et al., 1989] and his doctoral dissertation on object-oriented sofware development. These people and others met at a series of OOPSLA workshops starting in 1991. Draft versions of the first pattern catalog were matured during 4 years and eventually formed the basis for the first book on design patterns called *Design Patterns* [Gamma et al., 1995] that appeared in 1995. It was received with enthusiasm and the authors were given the name of Gang-of-Four. In the summer of 1993, a small group of pattern enthusiasts formed the "Hillside Generative Patterns Group" and subsequently organized the first conference on patterns called the "Pattern Languages of Programming" (PLoP) in 1994.

Patterns have been used for many different domains: development processes and organizations, testing, architecture, etc. Apart from *Design Patterns*, other important pattern books include *Pattern-Oriented Software Architecture: A System of Patterns* [Buschman et al., 1996a] —also called the POSA book, authored by five engineers at Siemens; and the book series entitled *Pattern Languages of Program Design* with five volumes to the date.

2.4.2 Pattern Misconceptions

One of the most recurring misconceptions about patterns is trying to reduce them to something known, like rules, programming tricks, data structures...

John Vlissides, one of the Gang of Four, comments in his book *Pattern Hatching* [Vlissides, 1998]:

> Patterns are not rules you can apply mindlessly (the teaching component works against that) nor are they limited to programming tricks, even the "idioms" branch of the discipline focuses on patterns that are programming language-specific. "Tricks" is a tad pejorative to my ear as well, and it overemphasizes solution at the expense of problem, context, teaching, and naming.

Since software patterns grew inside the object-oriented community to record object-oriented design principles, they are often seen as limited to object-oriented design. However, patterns capture expertise and the nature of that expertise is left open to the pattern writer. Certainly there's expertise worth capturing in object-oriented design — and not just design but analysis, maintenance, testing, documentation, organizational structure, and on and on. As Vlissides recognises:

"the highly structured style the GoF used in *Design Patterns* is very biased to its domain (object technology), and it doesn't work for other areas of expertise. Clearly, one pattern format does not fit all. What does fit all is the general concept of pattern as a vehicle for capturing and conveying expertise, whatever the field."

Not every solution, algorithm, best practice, maxim, or heuristic constitutes a pattern; one or more key pattern ingredients may be absent. Even if something appears to have all the requisite pattern elements, it should not be considered a pattern until it has been verified to be a *recurring phenomenon*. Some feel it is inappropriate to call something a pattern until it has undergone some degree of scrutiny or review by others [Appleton, 1997].

Documenting good patterns can be an extremely difficult task. To quote Jim Coplien [Coplien, 1998], good patterns do the following:

- It solves a problem: Patterns capture solutions, not just abstract principles or strategies.
- It is a proven concept: Patterns capture solutions with a track record, not theories or speculation.
- The solution isn't obvious: Many problem-solving techniques (such as software design paradigms or methods) try to derive solution from first principles. The best patterns generate a solution to a problem indirectly — a necessary approach for the most difficult problems of design.
- It describes a relationship: Patterns don't just describe modules, but describe deeper system structures and mechanisms.
- The pattern has a significant human component: All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

2.4.3 Patterns, Frameworks and Architectures

The practical nature of patterns themselves should not be underestimated. Ralph Johnson published a famous critique for computer people "going meta" too often, and stating the need for obtaining design experience [Beck et al., 1996]:

> One of the distinguishing characteristics of computer people is the tendency to go "meta" at the slightest provocation. Instead of writing programs, we want to invent programming languages, we want to

> create systems for specifying programming languages. There are many good reasons for this tendency, since good theory makes it a lot easier to solve particular instances of the problem. But if you try to build a theory without having enough experience in the problem, you are unlikely to find a good solution. Moreover, much of the information in design is not derived from first principles, but obtained by experience.

Kent Beck and Ralph Johnson points to the reasons why patterns are powerful tools in the design process [Beck and Johnson, 1994]. Their argumentation goes like follows: Design is hard. One way to avoid the act of design is to reuse existing designs. But reusing designs requires learning them, or at least some parts of them, and communicating complex designs is hard too. One reason for this is that existing design notations focus on communicating the "what" of designs, but almost completely ignore the "why". However, the "why" of a design is crucial for customizing it to a particular problem. Therefore we need ways (and this is what design patterns do) of describing designs that communicate the reasons for our design decisions, not just the results.

A closely related idea inside the object-oriented community is that of *framework*. A framework is the reusable design of a system or a part of a system expressed as a set of abstract classes and a way instances of (subclasses of) those classes collaborate.

Beck and Johnson were pioneers of object-oriented frameworks. They observed that frameworks, could be explained as a set of interrelated patterns. Thus, frameworks are a good source for pattern mining [Beck and Johnson, 1994]. They exemplify this with the HotDraw framework: explaining its architecture in terms of generic "Gang of Four" patterns [Gamma et al., 1995] plus new domain specific patterns such as *Drawing*, *Figure*, *Tool* and *Handle*, puts each of its classes in perspective. It explains exactly why each was created and what problem it solves. Therefore, presented this way, HotDraw becomes much easier to re-implement, or to modify. This is a completely different approach to describing the design of a framework than more formal approaches like Contracts. The more formal results only explain what the design is, but a pattern-based derivation explains why. This is similar to the proof process in mathematics, where the presentation of a proof hides most of its history, and where advances in mathematics are often caused by break-downs in proofs. Catalogs of design patterns will mature as people try to explain designs in terms of patterns, and find patterns that are missing from the

catalogs.

In parallel with the object-oriented programming community research on design patterns and pattern languages, other software engineering communities have been exploring *architectural styles*.

A software architecture is an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture. A software architecture is defined by a configuration of architectural elements —components, connectors, and data— constrained in their relationships in order to achieve a desired set of architectural properties [Fielding, 2000]. The principle of abstraction via encapsulation is central in software architecture. A complex system contains many levels of abstraction, each one with its own architecture. The architecture represents an abstraction of the system behavior at that level.

We have seen how frameworks are well described using patterns (and are source for pattern mining). Frameworks are as well a way to "implement" architectures. Nevertheless, the ideas of patterns (and pattern languages) and architectures overlap. Both are attempts to reuse design, and examples of one are sometimes used as examples of the other. Moreover, the object-oriented programming community has taken the lead in producing catalogs of design patterns, as exemplified by the "Gang of Fou" book and the essays edited by Coplien and Schmidt [Coplien and Schmidt, 1995].

2.4.4 Empirical Studies

The need for reliable software has made software engineering an important aspect for industry in the last decades. The steady progress recently produced an enormous number of different approaches, concepts and techniques: the object oriented paradigm, agile software development, the open source movement, component based systems, frameworks and software patterns, just to name a few. All these approaches claim to be superior, more effective or more appropriate in some area than their predecessors. However, to prove that these claims indeed hold and generate benefits in a real-world setting is often very hard due to missing data and a lack of control over the environment conditions of the setting.

In the joint paper *Industrial Experiences with Design Patterns* [Beck et al., 1996] authored together by Kent Beck (First Class Software),

James O. Coplien (AT&T), Ron Crocker (Motorola), John Vlissides (IBM) and other 3 experts, authors describe the efforts and experiences they and their companies had with design patterns. The paper contains a table of the most important observations ordered by the number of experts who mentioned them. This can be interpreted as the results of interviewing experts. The top 3 observations mentioned by all experts where:

1. Patterns are a good communication medium.

2. Patterns are extracted from working designs.

3. Patterns capture design essentials.

The first observation is, indeed, the most prominent benefit of design patterns: in the Design Pattern book [Gamma et al., 1995] by the Gang of Four two of the expected benefits are made explicit: first, the design patterns provide "a common design vocabulary" and, second, design patterns provide a "documentation and learning aid", which also focus on the communication process.

In [Prechelt et al., 1998] two controlled experiments using deign patterns for maintenance exercises are presented. For one experiment students were used to compare the speed and correctness maintenance work with and without design patterns used for the documentation of the original program. The result of this experiment was that using patterns in the documentation increases either the speed or decreases the number of errors for the maintenance task and thus seems to improve communication between the original developer and the maintainer via the documentation.

Another quantitative experiment is presented in [Hahsler, 2004]. They analyzed historic data describing the software development process of over 1000 open source projects in Java. They found out that only a very small fraction of projects used design patterns for documenting changes in the source code. Though the study had many limitations, e.g., the information on the quality of the produced code is not included. the results show a correlation between use of patterns and project activity, and that design patterns are adopted for documenting changes and thus for communicating in practice by many of the most active open source developers.

2.4.5 Pattern Languages

Christopher Alexander, coined the term pattern language. He used it to refer to common problems of civil and architectural design, from how cities should be laid out to where windows should be placed in a room. The idea was initially popularized in his book "A Pattern Language"[Alexander, 1977]. The pattern language technique has been used in many fields of design such as software design, human computer interaction, architecture, education, etc.

When a software engineer is designing a system, he must make many decisions about how to solve design problems. A single problem, documented with a generic approach, with its best solution, is a single design pattern. Each pattern has a name, a descriptive entry, and some cross-references to other patterns. The allowed sequences of pattern applications form a pattern language.

Just as words must have grammatical and semantic relationships to each other in order to make a spoken language useful, design patterns must be related to each other in order to form a pattern language. Alexander suggests that patterns should be organized so that they make intuitive sense to the designer. The actual organizational structure (hierarchical, iterative, etc. is left to the discretion of the pattern author, depending on the topic. Each pattern should indicate its relationship to other patterns and to the language as a whole. This gives the designer using the patterns some guidance about the order in which problems should be solved.

2.5

Summary

In this chapter we have reviewed several technologies upon which real-time multimedia systems can be built. Multimedia systems do not run in isolation

but needs the support of an operating system. We reviewed how the interaction between a multimedia system and the real-world takes place. Operating systems must provide certain capabilities in order to process multimedia streams in real-time. For example, it must be multi-threaded and preemptable; threads must have priorities; and predictable thread synchronisation mechanisms must exist.

Some operating system process scheduling algorithms have been reviewed and compared to dataflow schedulings. In one hand, processes can be scheduled freely (and they can freely terminate the execution), On the other hand, dataflow actors must obey restrictions imposed by the data dependencies specified by its graph, and actor's executions are atomic. Also, dataflow models often allow static scheduling analysis which calculates the cyclic scheduling prior to the running time.

Time plays a fundamental role in correctness of real-time multimedia computing. Real-time is not fast computing. Each data unit must be presented at a specific deadline, and jitter should not be allowed because the human ear is very sensitive to it. The effects of latency depends on the type of multimedia application.

We have shown that multimedia systems need *soft real-time* because missing a deadline, though undesirable, is not totally unacceptable. We have also reviewed how an operating system transmits hardware stimuli to a user-space process using interrupts, and kernel-space processes.

Actor-oriented design is based on hiding the internal behavior and state of an actor behind the actor interface. This strong encapsulation separates the behavior of a component from the interaction of that component with other components. A *model of computation* provides the exact semantics of the composition and communication style of the components. Many actor-oriented frameworks and languages in different fields have been shown.

The selection of an actor-oriented model of computation depends on the purpose of the system. The multimedia processing domain generally benefits from Dataflow and Process Network models. Dataflow actors communicate exclusively through FIFO queues. Actors expose their *token rates* —the number of tokes to be consumed and produced in each execution. An actor is only allowed to run when tokens stored in the input channels are sufficient. The message queues desynchronize the communication between actors, and system parallelism is explicitly exposed by the graph. Dataflow models of computation are appealing since they

closely match a designer's conceptualization of a system as a block diagram. Additionally, they offer opportunities for efficient implementation both in concurrent or sequential schedulings. This is specially true for statically schedulable dataflows, like the *Synchronous Dataflow*, which allow efficient code generation or synthesis for a whole scheduling cycle.

The algorithms for static scheduling of dataflow models have also been reviewed. Such algorithms begin solving the *balance equations* of the graph which gives the number of executions of each actor to complete a scheduling cycle. Next, they simulate a cycle execution which builds the scheduling. In Synchronous Dataflows, the runnability of a graph using finite memory is a decidable question.

Other sub-types of Dataflow models have been reviewed: The *Boolean-controlled Dataflow* adds conditional constructs with varying token rates. This model of computation does not guarantee to execute in bounded memory or finite cycles. *Dynamic Dataflow* allow token rates to change after an actor execution. *Process Networks* are even more dynamic and do not need to expose token rates. Process Networks must be run with a scheduler similar to the ones offered by operating systems. *Context-aware* Process Networks or Dataflows adds asynchronous message passing to the previous models. Such systems are indeterminate by nature.

We are interested in object-oriented design because it allow implementing actor-oriented models and complete dataflow-based multimedia systems. Object-oriented languages supports three characteristics: encapsulation, inheritance and polymorphism. Proponents of object-oriented programming claim that is easier to learn, to develop and to maintain, lending itself to more direct analysis and coding of complex systems.

Frameworks implement the aspects that are common across an entire domain describing a set of abstract classes and the way instances collaborate. A good framework can reduce the cost of developing an application by an order of magnitude because it reuses both code and design.

Lastly, this chapter has reviewed *design patterns*, a software engineering technique that allows effectively recording best design practices. A set of related patterns that can be applied in sequence form a *pattern language*. The domain of real-time multimedia systems remains quite impermeable to these techniques and, hence, offers good opportunities for research.

Now that the ground knowledge of our field has been introduced, the next

chapter reviews the state-of-the art and side-knowledge *related* to the specific problems formulated in the introduction. For each technology we specify why it does not effectively address the formulated problems —which typically is because it addresses other specific problems.

CHAPTER 3

State of the Art

This chapter discusses the previous work found in the literature related to the problems this thesis addresses. We outline, for each related work, whether it focuses on a slightly different problem, or is a valid, though incomplete, approximation to the same problem. When it is the latter case, we explain how our thesis builds upon it.

As stated in the introduction, one of the concrete problems this thesis addresses is the lack of timeliness in dataflow models of computation, in order to allow the hardware to regularly trigger the dataflow actors execution. Therefore, in section 3.1 we explore existing extensions, and related models, of Synchronous Dataflow. Most of the extensions deal with enhancing expressiveness by allowing the ports rates (the number of tokens consumed or produced in each execution) to change (see for example the Boolean-controlled Synchronous Dataflow in section 2.2.5), but this kind of extensions do not address the real-time limitations of dataflows. Some other extensions add the concept of time, and so are more promising. However, they do not deal with the needed time-triggered semantics.

Meta-models have been defined that allows modeling any computing system in the multimedia domain. In section 3.2 we review the *Meta-model for Multime-*

dia Systems. It is relevant to our problem because it links actor-oriented design with object-oriented technology. It generalizes concepts found in many reviewed actor-oriented multimedia systems and specifies those concepts in object-oriented terminology. We will later use the names and semantics of the meta-model in our design pattern language, which allows to translate our dataflow model into actual (object-oriented) designs and runnable code.

We have introduced the concept of design pattern in chapter 2. Now, in section 3.4 we present existing patterns related to the problem of developing complex dataflow systems in a predicable way. Most of these existing patterns have been collected and harmonized in a *Dataflow Pattern Language* [Manolescu, 1997]. Though its name suggest that it already solves our stated problem we show that it does not. Some of its patterns are not suited for the real-time multimedia domain. Others are, but they all are very coarse level patterns, that are insufficient for deriving complete designs on top of actor-oriented models. Our proposed patterns (in chapter 5) combine well with those ones collected by Manolescu.

3.1 Timeliness Dataflow Models

Some extensions of the SDF model exist. Some use a less constrained model allowing rate parameters to change during run-time; and a few others, like our TTSDF model (see chapter 4, introduce the concept of time). Here, we are interested in reviewing the models belonging to the second category.

3.1.1 Timeliness extensions to Synchronous Dataflow

In the category of timeliness extensions to Synchronous Dataflow (SDF) we find *Discrete Time (DT)* [Fong, 2000], which adds the concept of time progression to

SDF. In the DT model all tokens have *uniform token flow* in the sense that tokens get produced and consumed at regular and unchanging time intervals. Compared to TTSDF, both models use added latency, thus slightly breaking the SDF semantics for the initial resulting tokens (e.g. silence audio samples). However, the DT model is not useful in time-triggered callback-based architectures. In such architectures inputs nodes need to be triggered in regular times. Though this is similar to DT, the actual processing following the inputs nodes (executed inside the callback) needs to happen as quickly as possible. Therefore, uniform token flow in each actor connection is not desirable.

In DT all tokens flow in regular intervals. In time-triggered callback-based systems, tokens needs to flow on regular bursts of tokens driven by a hardware interrupt. Each burst must be quick enough to finish before the real-time deadline.

A similar approach to DT is *SDF-for-VLSI (VSDF)* [Kerihuel et al., 1994]. The VSDF model eases the transition between a synchronous dataflow and a hardware based implementation using specialized VLSI synchronous circuits. It introduces a time-aware notation that allows to specify a model and statically verify correct synchronization at Register-transfer Level. Like DT, VLSI is not suited for callback-based systems either.

Other time-related extensions to SDF have been designed to improve the parallelization into multiprocessors using accurate predictions of the execution times for dynamic work-loads [Pastrnak et al., 2004]. However, this approach is very problem-specific —the case presented is specific for a MPEG-4 video shape-texture decoding system—, and it is restricted to homogeneous SDF's, that is: all port rates are one, and actors run at the same rate.

3.1.2 Related Timeliness Models of Computation

Other models and architectures match more closely the concepts of time-triggered callbacks and hardware interrupt that are the base of our TTSDF model. We will show how they do not completely match our stated problem —which is: adapting a dataflow model to real-time performance. However, the reasons are subtle; therefore we encourage the reader to come back again to this section once being familiar with the TTSDF model presented in chapter 4.

Giotto [Henzinger et al., 2001] and Simulink with Real-Time Workshop from MathWorks, define and implement time-triggered models that allow the combination of fast running components with slow running components. These components

can run at different rates and communicate data between them. The technique exploits an underlying multitasking operating system with preemptive priority-driven multitasking. The two components are executed in different threads, ensuring that the slow running code cannot block the fast running code. In order to allow the two components to pass data, the model needs explicit blocks at the point of the rate conversion or extra delays. Although these time-triggered models can specify different execution rates for their components they do not have dataflow semantics because they lack queues on each arc, and so, execution order independence.

Another related model is *Discrete Events (DE)* model [Lee, 1999]. In DE components interact with one another via events that are placed on a time line. We find an interesting example of DE in ChucK [Wang and Cook, 2004], a concurrent programming language for multimedia.

The *Discrete-Event Runtime Framework* [Lee and Zhao, 2007] implemented in the Ptolemy II framework [Eker et al., 2003] combines time-triggered models and architectures with Discrete Events on one hand, and dataflow untimed computation on the other hand. Therefore a time-triggered actor execution can activate further down-stream executions. This very much resembles our technique of assigning schedules to callback activations. However, there is an essential difference: in TTSDF (see chapter 4) the requirement is to have only one kind of time-triggered event, which corresponds to the execution of all input nodes at the same instant, and have to finish with the execution of all output nodes. In Discrete-Event Runtime Framework, on the other hand, this restriction does not exist: time-triggered actors can activate other untimed actors but these activations do not necessarily finish with output nodes execution. Therefore, this model does not address our problem of fitting a dataflow scheduling in a time-triggered, callback-based architecture. It lacks the concepts of callback activations and also the concept of optimum initial latency to avoid jitter.

3.2

Object-Oriented Meta-Model for Multimedia Processing Systems

The Object-Oriented Meta-Model[1] for Multimedia Processing Systems, 4MPS for short, provides the conceptual framework (meta-model) for a hierarchy of models of media processing systems in an effective and general way. The meta-model is not only an abstraction of many ideas found in the CLAM framework (see section 6.1) but also the result of an extensive review of similar frameworks (see [Amatriain, 2004]) and collaborations with their authors. Therefore the meta-model reflects ideas and concepts that are not only present in CLAM but in many similar environments. Although initially derived for the audio and music domains, it presents a comprehensive conceptual framework for media signal processing applications. In this section we provide a brief outline of the meta-model, see [Amatriain, 2007a] for a more detailed description.

The 4MPS meta-model is based on a classification of signal processing objects into two categories: *Processing* objects that operate on data and control, and *Data* objects that passively hold media content. Processing objects encapsulate a process or algorithm; they include support for synchronous data processing and asynchronous event-driven control as well as a configuration mechanism and an explicit life cycle state model. On the other hand, Data objects offer a homogeneous interface to media data, and support for metaobject-like facilities such as reflection and serialization.

Although the meta-model clearly distinguishes between two different kinds of objects the managing of Data constructs can be almost transparent for the user. Therefore, we can describe a 4MPS system as a set of Processing objects connected in graphs called *Networks* (see figure 3.1).

[1]The word *meta-model* is here understood as a "model of a family of related models", see [Amatriain, 2004] for a thorough discussion on the use of meta-models and how *frameworks* generate them.

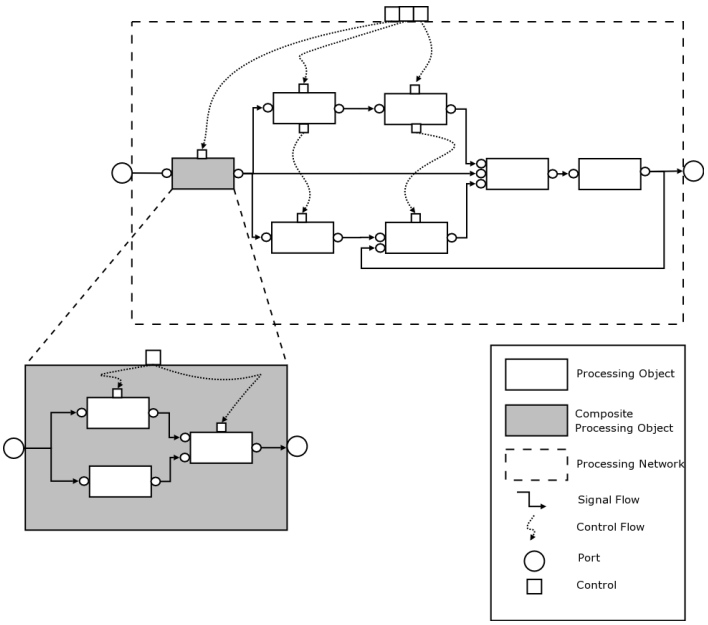


Figure 3.1: Graphical model of a 4MPS processing network. Processing objects are connected through ports and controls. Horizontal left-to-right connections represents the synchronous signal flow while vertical top-to-bottom connections represent asynchronous control connections. Diagram taken with permission from [Amatriain, 2004]

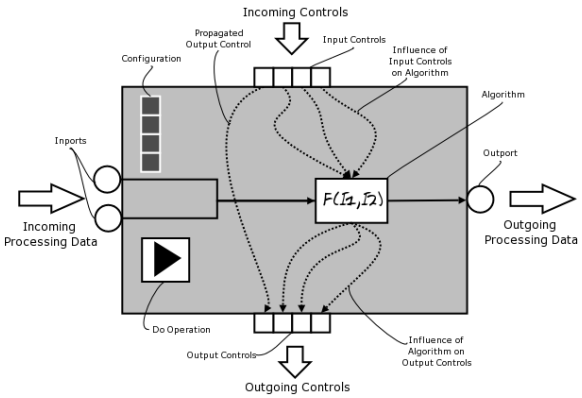


Figure 3.2: 4MPS Processing object detailed representation. A Processing object has input and output ports and incoming and outgoing controls. It receives/sends synchronous data to process through the ports and receives/sends control events that can influence the process through its controls. A Processing object also has a configuration that can be set when the object is not running. Diagram taken with permission from [Amatriain, 2004]

Because of this, the meta-model can be expressed in the language of actor-oriented (or graphical) models of computation as a *Context-aware Dataflow Network* (see section 2.2.8) and different properties of the systems can be derived in this way.

Figure 3.2 is a representation of a 4MPS processing object. Processing objects are connected through channels. Channels are usually transparent to the user that should manage Networks by simply connecting ports. However they are more than a simple communication mechanism as they act as FIFO queues in which messages are enqueued (produced) and dequeued (consumed).

The meta-model offers two kinds of connection mechanisms: *ports* and *controls*. Ports transmit data and have a synchronous dataflow nature while controls transmit events and have an asynchronous nature. By synchronous, we mean that messages are produced and consumed at a predictable —if not fixed— rate.

A processing object could, for example, perform a low frequency cut-off on an audio stream. The object will have an in-port and an out-port for receiving and delivering the audio stream. To make it useful, a user might want to control the cut-off frequency using a GUI slider. Unlike the audio stream, control events arrive sparsely or in bursts. A processing object receives that kind of events through controls.

The data flows through the ports when a processing is triggered (by receiving a *Do()* message). Processing objects can consume and produce at different rates and consume an arbitrary number of tokens at each firing. Connecting these processing objects is not a problem as long as the ports are of the same data type (see the *Typed Connections* pattern in section 5.2.4). Connections are handled by the *FlowControl*. This entity is also responsible for scheduling the processing firings in a way that avoids firing a processing with not enough data in its input ports or not enough space into its output ports. Minimizing latency and securing performance conditions that guarantee correct output (avoiding underruns or deadlocks, for instance) are other responsibilities of the FlowControl.

Life-cycle and Configurations

A 4MPS Processing object has an explicit lifecycle made of the following states: *unconfigured*, *ready*, and *running*. The processing object can receive controls and data only when running. Before getting to that state though, it needs to go through the *ready*, having received a valid *configuration*.

Configurations are another kind of parameters that can be input to Processing objects and that, unlike controls, produce expensive or structural changes in the processing object. For instance, a configuration parameter may include the number of ports that a processing will have or the numbers of tokens that will be produced in each firing. Therefore, and as opposed to controls that can be received at any time, configurations can only be set into a processing object when this is not in running state.

Static vs. Dynamic processing compositions

When working with large systems we need to be able to group a number of independent processing objects into a larger functional unit that may be treated as a new processing object in itself.

This process, known as composition, can be done in two different ways: *statically* at compile time, and *dynamically* at run-time (see [Dannenberg, 2004]). Static compositions in the 4MPS meta-model are called Processing Composites while dynamic compositions are called Networks.

Choosing between Processing Composites and Networks is a trade-off between efficiency versus understandability and flexibility. In Processing Composites the developer is in charge of deciding the behavior of the objects at compile time

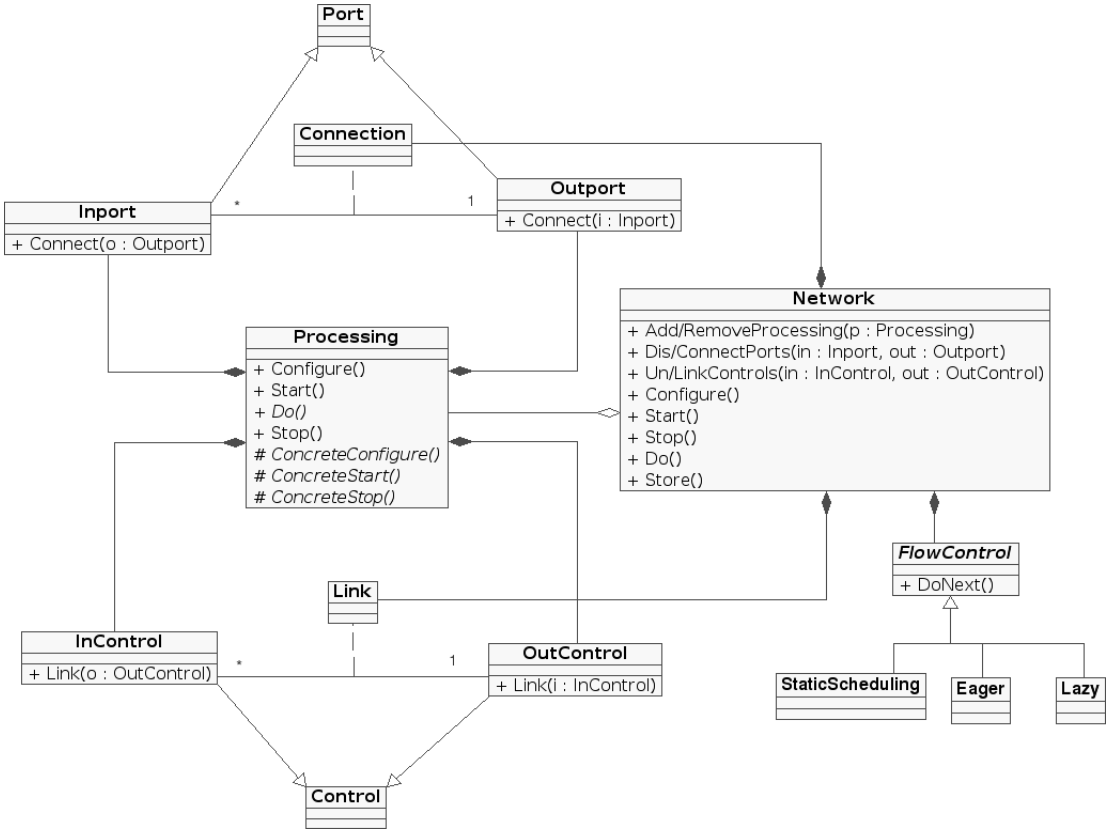


Figure 3.3: Participant classes in a 4MPS Network. Note that a 4MPS Network is a dynamic run-time composition of Processing objects that contains not only Processing instances but also a list of connected Ports and Controls and a Flow Control. Diagram taken with permission from [Amatriain, 2004]

and can therefore fine-tune their efficiency. On the other hand Networks offer an automatic flow and data management that is much more convenient but might result in reduced efficiency in some particular cases.

Processing Networks

Nevertheless, Processing Networks in 4MPS are in fact much more than a composition strategy. The Network metaclass acts as the glue that holds the meta-model together. Figure 3.3 depicts a simplified diagram of the main 4MPS metaclasses.

Networks offer an interface to instantiate new processing objects given a string with its class name using a processing object *factory* and a plug-in loader. They also offer interface for connecting the processing objects and, most important, they

automatically control their firing.

This firing scheduling can follow different strategies by either having a static scheduling decided before run-time (if the model of computation is a static schedulable dataflows. See section 2.2.4 for static scheduling algorithms) or implementing a dynamic scheduling policy (see dynamic dataflows and process networks in section 2.2.2) such as a *push strategy* starting firing the up-source processings, or a *pull strategy* where we start querying for data to the most down-stream processings. As a matter of fact, these different strategies depend on the given actor-oriented model of computation for dataflow processing. In any case, to accommodate all this variability the meta-model provides for different FlowControl sub-classes which are in charge of the firing strategy, and are pluggable to the Network processing container.

3.3 Previous Efforts in Multimedia Design Patterns

General design patterns —like the ones from the Gang of Four [Gamma et al., 1995], and the POSA [Buschman et al., 1996b] catalogs— are being more and more widely used in multimedia computing. This can be appreciated, for instance, in academic papers describing multimedia systems, where there is a growing tendency of documenting the overall system design in terms of *general* design patterns. But also in open-source projects, both in discussions on projects mailing-lists and in code documentation.

Nevertheless, the fact is that there is not any published catalog of design pattern in the multimedia computing domain. The present work is an attempt to change this situation and goes in the same direction as other very recent ef-

forts: Aucouturier presented several patterns for Music Information Retrieval in his thesis [Aucouturier, 2006]. Roger B. Dannenberg and Ross Bencina presented several patterns on audio and real-time in a ICMC 2005 workshop[2]. In the border line of the domain we find music composition patterns [Borchers, 2000] and, finally, a pattern language for designing patches for modular digital synthesisers [Judkins and Gill, 2000].

Apart from spreading design *best practices*, collecting multimedia patterns can serve to another goal: record *innovative* software designs for critical examination which might, eventually, become new *best practices*.

3.4

General Dataflow Patterns

Many pattern catalogs have been written on different domains. Some of them relates to general aspects of actor-oriented models of computation, dataflow and process network models —though not restricted to the multimedia processing domain— [Buschman et al., 1996b, Shaw, 1996], others covers specific aspects of dataflow [Meunier, 1995, Edwards, 1995]; and some of them are specialized patterns for a particular domain [Posnak and M., 1996].

Most of the state of the art work on dataflow-oriented design patterns has been harmonized and cataloged by Manolescu in a single well crafted catalog [Manolescu, 1997], which can also be seen as a pattern language. It is important to note here that the definition of "dataflow" in that work does not adhere to the definition given in section 2.2.2, but is more generic and encompasses any type of dataflows and process networks (see section 2.2.7).

[2]This very interesting pattern catalog made of 6 patterns can be found in the web: http://www.cs.cmu.edu/~rbd/doc/icmc2005workshop/. But we are not aware that they have been published in a formal academic publication.

Compared to the systems engineering approach (see 2.2), these patterns are not a theoretical approach to dataflow models but rather the result of an exhaustive analysis of existing software solutions. Therefore, they represent a key element to translate the model requirements into the software domain.

The dataflow paradigm (in Manolescu's generic definition) is a very broad area, thus it is not strange that we find different systems with conflicting quality-of-service requirements. The applicability of the pattern language to the multimedia processing domain varies depending on the pattern. The most architectural and high-level ones apply well, but other more specific patterns have forces that are in conflict with those in the multimedia processing domain, and are more oriented to offline or batch processing. The pattern language is composed by the following four patterns: *Dataflow architecture*, *Payloads*, *Module data protocol*, and *Out-of-band and in-band partitions*. Though some of them offer several variants, the pattern language cannot be considered a complete language (it does not cover a domain). Our contributed pattern language (see chapter 5) extends this initial pattern language focusing on the multimedia processing domain. Therefore, we summarise here these four patterns to give the necessary context.

3.4.1 Pattern: Data flow architecture

A variety of applications apply a series of transformations to a data stream. The architectures emphasize data flow, and control flow is not represented explicitly. These applications consist of a set of *modules* that interconnect forming a new module or *network*. The modules are self-contained entities that perform generic operations that can be used in a variety of contexts. A module is a computational unit while a network is an operational unit. The application functionality is determined by: types of modules and interconnections between modules. The application could also be required to adapt dynamically to new requirements.

In this context, sometimes a high-performance toolkit applicable to a wide range of problems is required. The application may need to adapt dynamically or at run-time. In complex applications it is not possible to construct a set of components that cover all potential combinations. The loose coupling associated with the black-box paradigm usually has performance penalties: generic context-free efficient algorithms are difficult to obtain. Software modules could have different incompatible interfaces, share state, or need global variables.

The Solution is to highlight the data flow such that the application's architec-

ture can be seen as a network of modules. Inter-module communication is done by passing messages (sometimes called tokens) through unidirectional input and output ports (replacing direct calls). Depending on the number and types of ports, modules can be classified into *sources* (only have output ports and interface with an input device), *sinks* (only have input ports and interface with output devices), and *filters* (have both input and output ports).

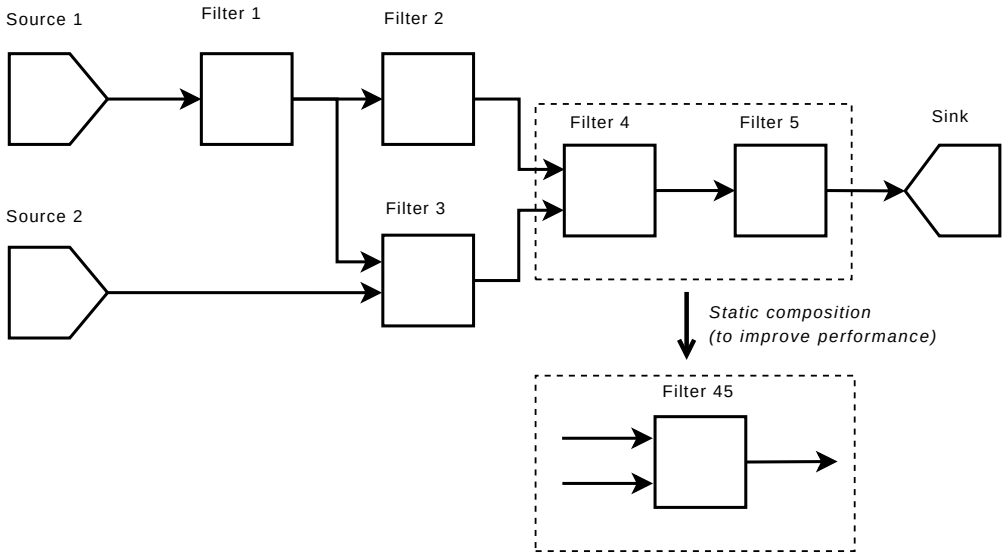


Figure 3.4: Dataflow architecture

Unidirectional input and output ports are not a limitation. Rather, they increase a component's autonomy, such that, provided that there are no feed-back loops, processing of a component is unaffected by the presence or absence of connections at the output ports. For two modules to be connected the output port of the upstream module and the input port of the downstream module must be plug-compatible. Having more than one data type means that some modules perform specialized processing. Filters that do not have internal state could be replaced while the system is running. The network usually triggers re-computations whenever a filter output changes.

In a network, adjacent performance-critical modules could be regarded as a larger filter and replaced with an optimized version, using the *Adaptive Pipeline* pattern [Posnak and M., 1996] which trades flexibility for performance. Modules that use static composition cannot be dynamically configured.

3.4.2 Pattern: Payloads

In dataflow-oriented software systems separate components need to exchange information either by sending messages (payloads) through a communication channel or with direct calls. If it is restricted to message passing, payloads will encapsulate all kinds of information but components need a way to distinguish the type as well as other message attributes such as asynchronousity, priority... Some overhead is associated with every message transfer. Depending on the kind of communication, the mechanism must be optimized.

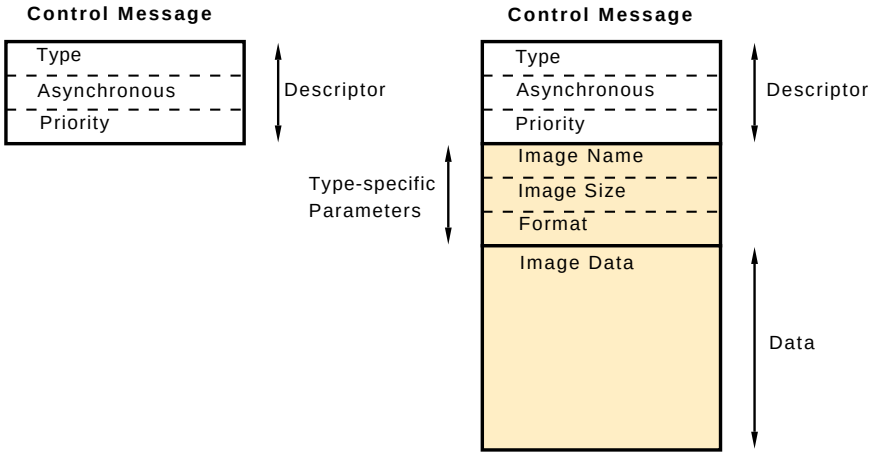


Figure 3.5: Different payloads and their components

Payloads give a solution to this problem. Payloads are self-identifying, dynamically typed objects such that the type of information can be easily identified. Payloads have two components: a descriptor component and a data component. In the case where different components are on different machines, payloads need to offer serialization in order to be transmitted over the channel.

Payload copying should be avoided as much as possible using references whenever possible. If the fan out is larger than one, the payload has to be cloned. In order to reduce copies even in that case, the cloned copies can be references of the same entities and only perform the actual copy if a downstream receiver has to modify its input. If it is not possible to avoid copying there are two possibilities: shallow copy (copy just the descriptor and share the data component) and deep copy (copy the data component as well maybe implementing copy-on-write).

The greatest disadvantage of the payload pattern compared to direct call is its inefficiency, associated with the message passing mechanism. One way to minimize

it is by grouping different messages and sending them in a single package.

A consequence of this pattern is that new message types can be added without having to modify existing entities. If a component receives an unknown token, it just passes it downstream.

3.4.3 Pattern: Module data protocol

Collaborating modules pass data-blocks (payloads) but depending on the application, the requirements for these payloads could be very different: some may need asynchronous user events, some may have different priority levels, some may contain large amounts of data. On the other hand, sometimes the receiving module operates at a slower rate than the transmitter, to avoid data loss the receiver must be able to determine the flow control.

Besides, we must take into account a number of possible problems. Large payloads make buffering very difficult. Payloads with time-sensitive data have to be transferred in such a way that no deadlines are violated. Asynchronous or priorized events are sent from one module to another- Shared resources for inter-module communication might not be available or the synchronization overhead not acceptable. And flow control has to be determined by receiving module.

There are three basic ways to assign flow control among modules that exchange Payloads:

- Pull (functional): The downstream module requests information from the upstream module with a method call that returns the values as result. This mechanism can be implemented via a sequential protocol, may be multi-threaded and may process in-place. The receiving module determines flow control. It is applicable in systems where the sender operates faster than the receiver. This mechanism cannot deal with asynchronous or high-priority events.

- *Push* (event driven): The upstream module issues a message whenever new values are available. The mechanism can be implemented: as procedure calls containing new data as arguments; as non-returning point to point messages or broadcast; as high-priority interrupts; or as continuation-style program jumps. Usually the sending module does not know whether the receiver is ready or not. To prevent data loss the receiver can have a queue. If there

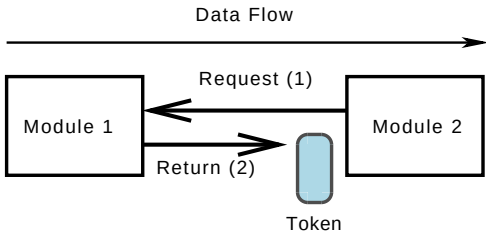


Figure 3.6: The pull model for inter-module communication.

are asynchronous or high-priority events, the queue must let them pass, else a simple queue can do.

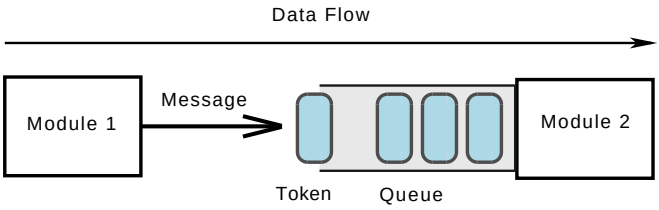


Figure 3.7: The push model for inter-module communication.

- *Indirect* (shared resources): Requires a shared repository accessible to both modules. When the sender is ready to pass a payload to the receiver, it writes in the shared repository. When ready to process, the receiver takes a payload from the repository. The sender and the receiver can process at different rates. If not all the payloads are required by the receiver, the upstream module can overwrite data.

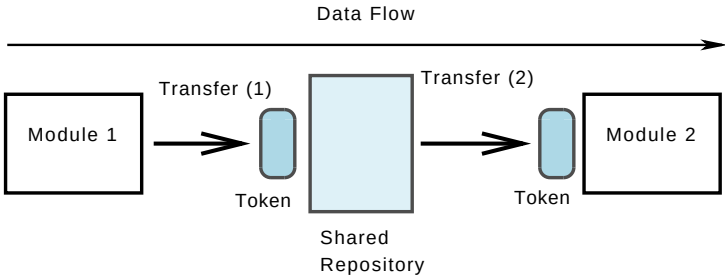


Figure 3.8: The indirect model for inter-module communication.

It must be noted though that having more than one input port complicates flow control and requires additional policies.

3.4.4 Pattern: Out-of-band and in-band partitions

An interactive application has a dual functionality: first it interfaces with the user handling event-driven programming associated with the user interface and the response times have to be in the order of hundreds of milliseconds; second it handles the data processing according to the domain requirements

User actions are non-deterministic so user interface code has to cover many possibilities. Data processing has strict requirements and the sequence of operations (algorithm) is known before hand. Human users require response in the order of hundreds of milliseconds but applications emphasize performance that is irrelevant for the user interface. Generally, a large fraction of the running time is spent waiting for user input. The user interface code and data processing code are part of the same application and they collaborate with each other.

The solution is to organize the application into two different partitions:

- Out-of-band partition: typically responsible for user interaction.
- In-band partition: it contains the code that performs data processing. This partition does not take into account any aspects of user interaction

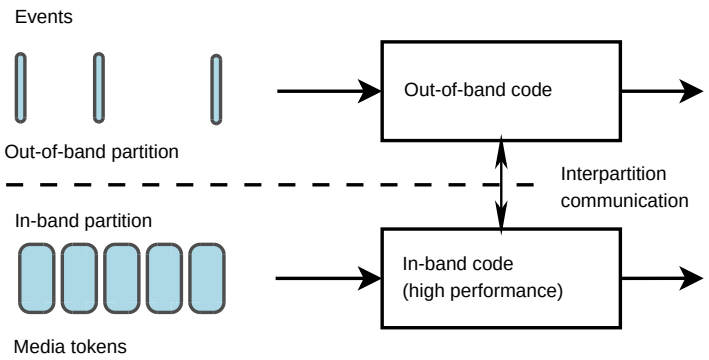


Figure 3.9: Out-of-band and in-band partitions within an application.

3.5

Summary

This chapter has reviewed previous art related to the problems this thesis address. We have started analysing extensions of the Synchronous Dataflow model that support the concept of time. In the *Discrete Time* model , all tokens flow at regular time intervals. Though promising, it does not address our problem. This is because we are interested in adapting dataflows in a time-triggered callback-based architecture. For accomplishing that, only the input nodes must be time-triggered. Once a callback execution is triggered, all actors must execute as quickly as possible in order to finish before the real-time deadline. Discrete Time, on the contrary, execute all actors in regular intervals. *SDF-for-VLSI* is a model similar to Discrete Time that addresses hardware implementation with synchronous circuits. Other time-related SDF extensions have been used to improve the parallelization of dataflow systems, but their solution is too restrictive: multi-rate is not allowed.

We have reviewed timeliness actor-oriented models that do not belong to the Dataflow family. *Simulink* and *Giotto* systems implement *time-triggered models* exploiting an underlying multitasking and preemptive operating system. These models do not address our problem because they lack queues on each arc, and so, dataflow semantics and execution order independence. The same happen with the *Discrete Events* model.

The *Discrete-Event Runtime Framework* combines time-triggered models with dataflow untimed computation. This resembles very much our requirements. However it does not have output nodes that close the time-triggered callback activation. Hence, it is not suited for time-triggered callback-based architectures.

This chapter has also reviewed an *Object-Oriented Meta-Model for Multimedia Processing Systems (4MPS)*. Meta-model analysis is a technology related to our stated goals because it facilitates the translation of models into object-orientation designs. It gives names and semantics to elements common in many reviewed actor-oriented models and frameworks, which can be used in the design patterns.

The 4MPS meta-model is based on a classification of signal processing objects into two categories: *Processing* objects that operate on data and control, and *Data* objects that passively hold media content.

While 4MPS offers a valid high-level meta-model for most dataflow-based frameworks and environments, it is sometimes more useful to present a lower-level architecture in the language of design patterns, where recurring and non-obvious design solutions can be shared (such the ones presented in chapter 5). Thus, such pattern language bridges the gap between an abstract meta-model such as 4MPS and the concrete implementation given a set of constraints.

The multimedia domain has not given much attention regarding domain-specific patterns. They are actually hard to find. Some catalogs exist on multimedia sub-domains such as Music Information Retrieval and real-time audio processing —unfortunately, the later doesn't cover dataflow-based systems.

Outside (or not restricted to) the multimedia domain, we find some catalogs of dataflow patterns. Fortunately, most of them have been collected in a single and well crafted catalog. We have reviewed the patterns more relevant to our purposes.

Most patterns from this catalog are quite high-level (thus, not implementation oriented) and valid for our domain. Others are not. Some requirements imposed by the multimedia processing systems tends to be quite restrictive —for instance, real-time constrains. Thus, specific patterns for our domain are required.

Next chapter contributes the *Time-Triggered Synchronous Dataflow model* that overcomes the problems of dataflow models and real-time.

CHAPTER 4

Time-Triggered Synchronous Dataflow

In the previous chapter we established the grounds of Dataflow models of computation (in section 2.2.2). And static scheduling of Synchronous Dataflow (SDF) was reviewed in section 2.2.4.

Building upon this, the present chapter contributes a new actor-oriented model of computation, within the Dataflow models family: the *Time-Triggered Synchronous Dataflow (TTSDF)*. This model is inspired on the Synchronous Dataflow (SDF) but, differently from SDF, it is suited for real-time processing. Specifically, it adds the advantage of enabling the model to be driven by time-triggered interrupts while preserving correct computation. TTSDF models can be run efficiently because no extra operating-system threads are required, they operate with the optimum latency, and jitter is totally avoided.

We start in section 4.1 specifying the issues that Synchronous Dataflow have in real-time systems —specifically, when integrating a SDF model in a time-triggered callback-based architecture—, and give a detailed scenario experimenting such issues.

The TTSDF model is formally described in section 4.2. Next, a scheduling algorithm is proposed and proved valid in section 4.3. To further demonstrate the behaviour of the model, a scheduling example is presented in section 4.4.2 and more examples are presented in appendix B.

Finally, we recap and draw conclusions in section 4.7.

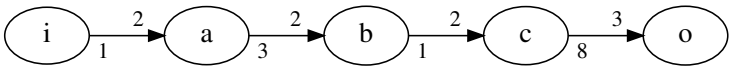
4.1

The Problem of Timeliness in Dataflows

As we have stressed in section 2.1.7, the "callback" style of programming is prefered over the "blocking I/O" style for real-time multimedia processing. However, an SDF schedule, like the one in figure 4.1, does not fit well in a timed callback-based architecture. On one hand, in such architectures, callbacks driven by the hardware are triggered in regular timings. On each trigger, inputs have to be read and outputs have to be produced, within the established real-time deadlines (see section 2.1.2). On the other hand, the SDF model is an *untimed* abstraction because timing constrains are not captured by the model —only indirectly by the flowing token ordering.

Ideally we would like to combine regularly timed executions with SDF data dependencies. Specifically, some dataflow actors should be *timed* —the ones linked with callback inputs and outputs—, and they should be interleaved by sequences of the other *untimed* actors, while respecting the graph data dependencies. However, a periodic SDF scheduling may contain many instances of each —timed or untimed— actor, therefore constructing such combination is not obvious.

A different, and simpler, approach consists on relying on buffering done in another thread. We will devote the next paragraphs in this section to show that



$$\Gamma = \begin{bmatrix} -2 & 0 & 0 & 1 & 0 \\ 3 & -2 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 \\ 0 & 0 & 8 & 0 & -3 \end{bmatrix}$$

- Period executions: $\vec{q}$={8,4,6,3,8}. Nodes order: *i, a, b, c, o*
- SDF periodic schedule: $\mathbf{i}_0$, $\mathbf{i}_1$, a_0, b_0, $\mathbf{i}_2$, $\mathbf{i}_3$, a_1, b_1, c_0, $\mathbf{i}_4$, $\mathbf{o}_0$, b_2, $\mathbf{i}_5$, $\mathbf{o}_1$, a_2, b_3, c_1, $\mathbf{i}_6$, $\mathbf{o}_2$, $\mathbf{o}_3$, $\mathbf{i}_7$, a_3, b_4, $\mathbf{o}_4$, b_5, c_2, $\mathbf{o}_5$, $\mathbf{o}_6$, $\mathbf{o}_7$

Figure 4.1: A simple SDF graph and scheduling. It is problematic to run such graph within real-time constrains

this approach is flawed and not usable in the general case.

It is actually easy to find examples of this buffering technique in audio (programming) libraries that offer access to the audio device —for example, the Linux's Alsa, and the cross-platform PortAudio [Bencina and Burk, 2001]. The technique is often called *callback to blocking interface adaptation* and is implemented having a user's processing thread that synchronises with the callback thread, exclusively dedicated to buffering. While this works for audio device access using a fixed buffer size, it is not appropriate for running multi-rate dataflows. To illustrate the exact problems we will use the example of SDF scheduling in figure 4.1. We are interested on the latency and jitter issues, thus we assume that the callback real-time deadlines are always met.

Chronogram Analysis of the Naive Solution

The chronogram in figure 4.1 shows the result of executing the SDF schedule —as provided by the static scheduler algorithm—, by means of the described callback to blocking interface adaptation. The chronogram requires a detailed analysis. Communication with the outside world happens through buffers provided by the callback process. These buffers also act as thread boundaries using mutexes and semaphores between the callback and the SDF threads —using lock-free techniques from the callback thread.-

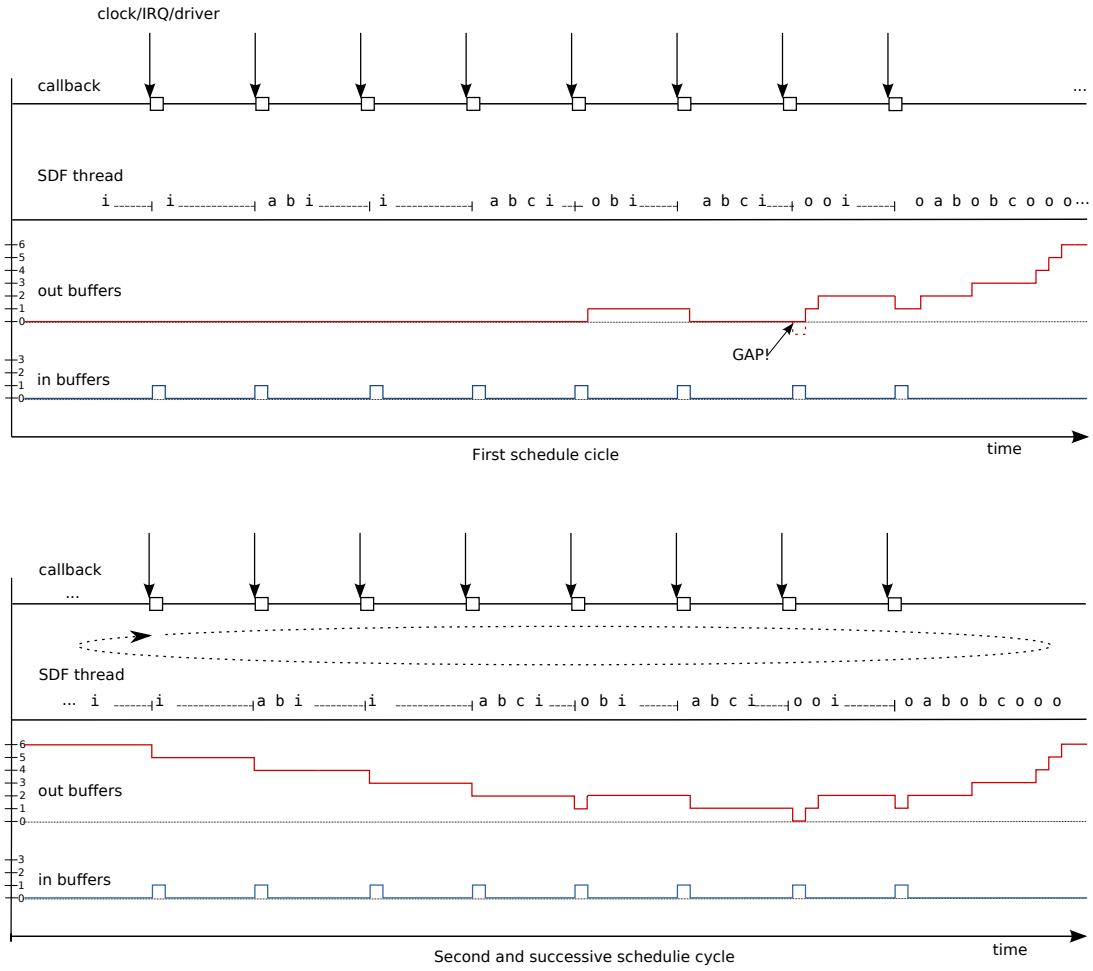


Figure 4.2: A sequence diagram (chronogram) showing the problematic adaptation of an SDF schedule (from figure 4.1) on top of a callback architecture. The line on the top shows the callback process that deals with input and output buffering. The second line shows the SDF execution thread, with blocking executions of actor **i** (input). Next two lines show the size of output and input buffers. Points marked with *GAP* indicate a lack of output buffer to be served by the callback. This will produce a gap in the output stream.

The first observation on this example is that all input (actor **i**) executions are blocking —because it have to wait for new data to be available— while output (actor **o**) executions are not blocking —because output buffers are always ready. But this is not the general norm: one could reduce the total size of output buffers so that output executions would need to block until some space is freed. In the chronogram it is easy to see that each blocking input causes a unitary increase of the output buffers, while each blocking output would cause a unitary increase of input buffers.

The second observation is that this technique adds excessive *latency*. The first output goes out at the 6th callback activation. And this is not necessary: executions order could be arranged differently to obtain half that latency. To make matters worse, latency is not stable but increases during the first schedule cycle: in the 7th callback activation, output is not sent out, causing a *gap* in the output stream, and it is deferred until the 8th callback. This gap increases latency. Such variation on the latency in which a continuous stream is presented to the user is called *jitter*.

The third observation is that the second cycle starts with enough buffering, therefore no more gaps are introduced. We also observe that the second cycle ends with the same buffer sizes as it starts, therefore next iterations will follow the same sequence.

Summing up, running an SDF schedule in a callback-based architecture is problematic because in general:

- it introduces more latency than necessary.
- it introduces jitter. At least during the first cycle execution, but also in successive cycles depending on how buffering is managed.
- it introduces run-time overhead because of the context switching of threads. Moreover, in some of our target architectures (e.g., some audio plug-in architectures), spawning threads is not allowed. In other cases it is allowed, but they run at lower priority than the callback process.

The Time-Triggered Approach

Therefore we need a new approach that combines regularly triggered actors that consume the external inputs with other untimed actors. We would also like to en-

able the scheduling to run *inside* the callback process, avoiding jitter and guaranteeing optimum latency. Such a wish list is realized in the chronogram in figure 4.1, for the same SDF example.

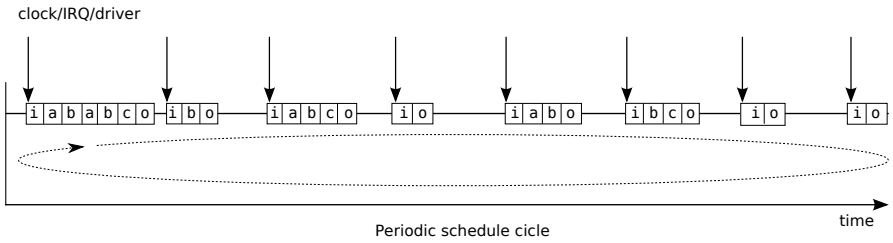


Figure 4.3: The desired way of running the same SDF periodic schedule (defined in figure 4.1). Now processing blocks are run inside the callback and no buffering is required nor blocking reads/writes. The reason why this is possible is that inputs and outputs are well distributed. However, before running the periodic schedule, the following prologue is needed to initialize the internal dataflow queues: (i,o), (i,o), (i,o)

Although the example in figure 4.1 shows a sequential computation —all actors are executed in the callback process—, it is still possible to optimize the run-time by parallelizing on multiple processors in a similar way that is done for SDF graphs (see section 4.5.

Now that the problem and a possible solution have been illustrated we are ready to precisely define a new model of computation that suites that purpose.

4.2

The TTSDF Computation Model

Our goal is to find a systematic methodology or algorithm for obtaining sequential schedules of actors in a way that can be run in the real-time callback

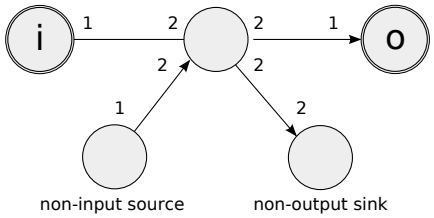


Figure 4.4: A TTSDF graph with a non-input source and a non-output sink. Note that such source and sink can run at different rates, while inputs and outputs must run at the same rate.

function, avoiding jitter and optimizing the latency. Additionally we want to prove that all graphs that have an SDF schedule also have an schedule that fulfils these conditions.

This work follows a similar development than the one found in [Lee and Messerschmitt, 1987a], on statical scheduling of synchronous dataflows and builds upon some of its definitions, lemmas, and theorems.

Definition 1. A *Time-Triggered Synchronous Data Flow (TTSDF)* is a connected and directed graph with each arc labeled with two integers, one corresponding to the amount of samples being produced to that arc and the other corresponding to the amount of samples being consumed from that arc, a subset of source nodes I or *inputs*, a subset of sink nodes O or *outputs*, and $\vec{b}(0)$ a vector containing the initial buffer sizes of all arcs.

Note that all input nodes are sources and all output nodes are sinks but the opposite is not true. Figure 4.4 shows a graph with a source node tagged as "input" a sink node tagged as "output" and an additional source and sink.

The only difference with an SDF graph is that a TTSDF comes with some sources tagged as inputs and some sinks tagged as outputs. Like in SDF, the number of tokens consumed and produced for each block execution is known *a priori*. Note that any SDF model can be interpreted as a TTSDF model with no sink or source tagged as input or output.

Dataflow models, and SDF and TTSDF in particular, are *coordination languages* as defined in [Halbwachs, 1998] and [Lee and Parks, 1995]. Coordination languages permit nodes in a graph to contain arbitrary subprograms, but define a precise semantic for the interaction between nodes. We call *host languages* those languages used to define the subprograms, which are usually conventional

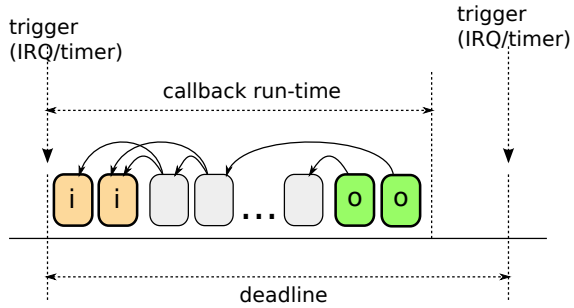


Figure 4.5: A callback activation with two inputs and two outputs. Arrows represents execution dependencies between nodes imposed by the dataflow graph.

languages such as C or C++.

We give the semantics of the TTSDF coordination language by defining the interaction between time-triggered callbacks, timed (input/output) actors and untimed actors. Furthermore, we use a mathematical computation model to reason about the admissible schedules of the dataflow.

4.2.1 Callback-based Coordination Language

Three types of actors are distinguished: *inputs*, *outputs* and *untimed* actors. Inputs are time-triggered, that is, their execution is driven by the time-triggered callback. Each input is associated to a callback buffer, each of them may consist in different number of tokens. All inputs run together sequentially in an arbitrary order. An input execution consists on reading its callback input buffer and sending out tokens to its dataflow queues. Immediately after the inputs, zero or more untimed nodes are run. Note that untimed actors can be either filters, sources or sinks.

Outputs close the execution sequence triggered by the callback. As the inputs, they run together in an arbitrary order. Termination of outputs execution finishes the callback function. Conversely to inputs, each output writes tokens from its queues into its callback output buffer. Outputs are not time-triggered but are time-restricted because they must run before the deadline —which often coincides with the next callback.

Each execution sequence triggered by a callback is called *callback activation*. The dataflow semantics are preserved by an important restriction: the concatenation of one or more callbacks activations must form a periodic scheduling. Fig-

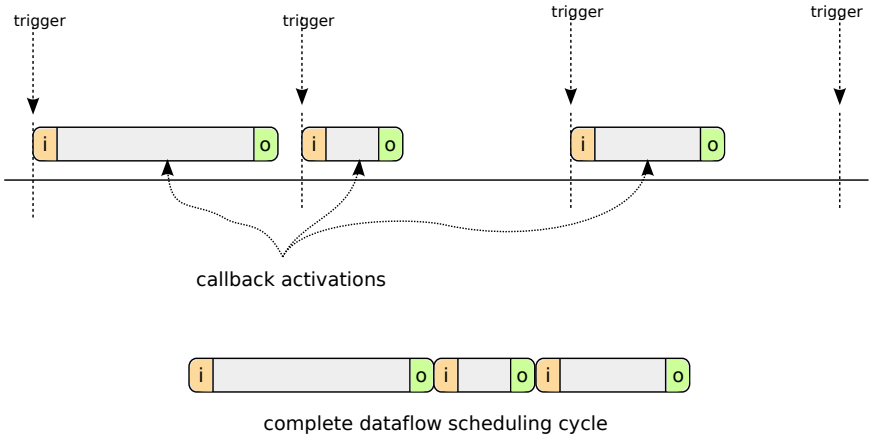


Figure 4.6: Multiple callback activations forming a dataflow cycle scheduling. For clarity, here **i** and **o** represents a sequence of all inputs and outputs (or $\wp(I)$ and $\wp(O)$)

ure 4.5 shows a graphical example of a callback activation with dependencies between node executions. Figure 4.6 shows how one or more callback activations form a complete dataflow scheduling cycle (or period).

A first consequence of this model is that jitter cannot exist because no input or output buffer can be missed. When a callback activation finishes, the outputs are always filled with new data[1].

Finally, note that we always refer to the "callback function", but a "function" is not technically necessary. It could also be a similar —and equivalent— scheme like a thread that is awaken by a time-trigger. However, defining callback functions is the typical solution used in open architectures to allow developers to plug in their processing algorithm.

4.2.2 Formal Computation Model

Definition 2. $\wp(A)$ indicates an arbitrary sequence of all elements in the set A.

Definition 3. A **callback activation** is any a sequence in the following language: $C = \{\wp(I)(V - (I \cup O)^*)\wp(O)\}$. And l *callback activations* is any sequence in C^l, where V is the set of nodes in the graph, and I and O are the nodes labeled as inputs and outputs respectively.

[1] which can either be a transformation of the input buffers provided in the same callback or a transformation of inputs in previous callbacks.

Definition 4. A **callback order** is any prefix of the language C^*

The following sequences A and B are examples of a callback order —and all their sub-sequences as well— while C is not in callback order. Assume that the inputs and ouputs are $I = \{i_a, i_b, i_c\}$, $O = \{o_d, o_e\}$ and the rest of nodes $V - (I \cup O) = \{m, n\}$. Parenthesis are used to indicate complete callback activations

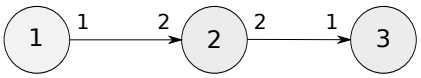
$$\begin{aligned} A &= [(i_a, i_b, i_c, m, n, o_d, o_e), (i_a, i_b, i_c, n, n, o_d, o_e)] \\ B &= [(i_a, i_b, i_c, o_d, o_e), i_a, i_b] \\ C &= [i_a, i_b, m, i_a] \end{aligned} \tag{4.1}$$

As the SDF, this computational model consists on a graph with a FIFO queue on each arc that gets tokens from its producer processing node and passes them on to its consumer processing node. Such queues will be called *buffers* and their size after n executions (or firings) can be computed as follows:

$$\vec{b}(n+1) = \vec{b}(n)\Gamma\vec{v}(n) \tag{4.2}$$

where $\vec{v}(n)$ represents the processing node being executed in nth place. Each $\vec{v}(n)$ is a vector with a 1 at the position corresponding to the node to be executed and zeros all the rest.

Taking this simple graph as example,



it needs two executions of nodes 1 and 3 and one execution of node 2 to complete a period. This is the execution rate per period and is given by the vector $\vec{q}$ which is the smallest integer vector in the *nullspace* of the graph topology matrix Γ:

$$\Gamma = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 1 & -2 \end{bmatrix} \Gamma\vec{q} = \vec{0}; \vec{q} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \tag{4.3}$$

An execution is valid if it fulfils two conditions: the buffer size $\vec{b}(n)$ remains a non-negative vector, and the sequence of firings remains in *callback order*. This second condition is what makes TTSDF a different model than SDF.

This condition means that any allowed sequence of executions begins with all inputs, then any number of non input/outputs, then all outputs, and then it

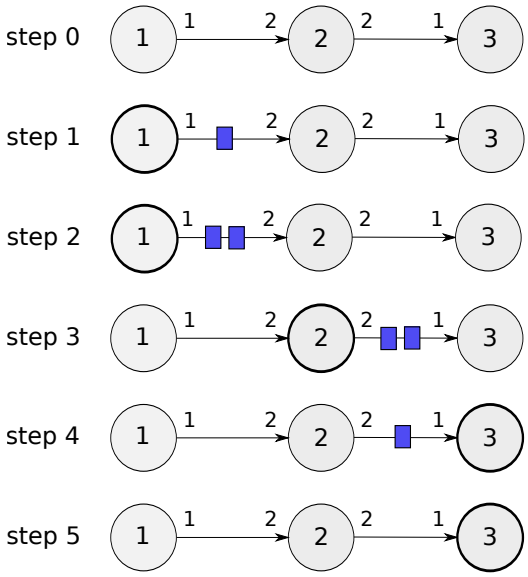


Figure 4.7: A sequence of executions in a simple graph. For each step n, nodes in bold represent the last fired node, which corresponds to vector $\vec{v}(n)$; and the tokens in the two arcs correspond to vector $\vec{b}(n)$. For example $\vec{v}(1) = \{1, 0, 0\}$ and $\vec{b}(1) = \{1, 0\}$.

starts with the inputs again. Note that non-input sources and non-output sinks can exist and, regarding the callback order, are considered exactly as other non input/output nodes.

In our example, assuming that node 1 is an input and node 3 an output, figure 4.7 shows a valid TTSDF scheduling.

4.3

Static Scheduling of TTSDF Graphs

Definition 5. (From [Lee and Messerschmitt, 1987a]) A **periodic admissible sequential schedule (PASS)** is a sequence including all nodes in the graph such that if the nodes are executed in that order the amount of tokens in the buffers (or buffer sizes) will remain non-negative and bounded.

Definition 6. A **callback periodic admissible sequential schedule (CPASS)** is a periodic and infinite admissible sequential schedule in *callback order*. It is specified by a list ϕ that represents one execution period. The *period* of a CPASS is the number of nodes in one execution period.

Definition 7. An **l-latency CPASS** is an infinite admissible sequential schedule in *callback order* consisting in two parts (σ, ϕ):

- the first part is a finite prologue σ made of l *empty callback activations*, that is $(\wp(I)\wp(O))^l$.
- the second part is an infinite schedule specified by one execution period ϕ.

The scheduled TTSDF graph is a modified version of the given process graph in which buffers (delays) are added to the incoming arcs of the output nodes. The amount of delay tokens added to an output arc is rl, where r is the consuming rate of the arc and l the given latency.

This means that, in terms of communication with the outside world, the first l callback activations (the *prologue*) will only buffer in new tokens and buffer out the added delays. Starting from the $(l+1)$th execution the output nodes will produce tokens that have actually been *processed* by the graph. In typical situations these correspond to transformations of tokens from the input nodes.

Adding delays enables the schedule to do buffering while respecting the callback order restriction. As we will see, this buffering is necessary in order to reach a (buffering) state in which a periodic execution schedule in callback order exists.

Theorem 1. Necessary conditions *Given a TTSDF with topology matrix* Γ*, the two following conditions are necessary for the existence of a CPASS:*

- $rank(\Gamma) = s - 1$*, where* s *is the number of nodes*
- *for any non-zero* $\vec{q}$ *such that* $\Gamma\vec{q} = \vec{0}$ *it is true that* $\forall i \in I \cup O, \vec{q_i} = r$ *for some integer* r

In other words, if these two conditions are not fulfilled a CPASS will not exist. Compared to the SDF model the TTSDF just adds the second condition. This second condition means that all nodes marked as input and output must have the same rate in terms of number of executions per period.

We will now prove the theorem validity.

Proof. The proof for the first condition can be found in [Lee and Messerschmitt, 1987a] and it is valid for both SDF and TTSDF). But we sketch the proof here for completion sake:

We need to prove that the existence of a CPASS implies $rank(\Gamma) = s - 1$ By definition of the computation model, the size of the buffers is given by: $\vec{b}(n+1) = \vec{b}(n) + \Gamma\vec{v}(n)$ Let p be period of the schedule and $q = \sum_{n=0}^{p-1} \vec{v}(n)$. Therefore we can write $\vec{b}(p) = \vec{b}(0) + \Gamma\vec{q}$. And since the CPASS is periodic, we can write $\vec{b}(np) = \vec{b}(0) + n\Gamma\vec{q}$. Since the CPASS is admissible, the buffers must remain bounded by Definition 1. Buffers remain bounded if and only if $\Gamma\vec{q} = \vec{0}$. Therefore if a CPASS exists then $rank(\Gamma) < s$, and $rank(\Gamma)$ can only be s or $s - 1$, thus $rank(\Gamma) = s - 1$

The second condition will be proved by contraposition: we have to see that if two input/output nodes are to be executed a different number of times per cycle then a CPASS does not exist.

Let be i and j indices such that $\vec{q_i} > \vec{q_j}$ and let $n = \vec{q_i} - \vec{q_j}$. After running m complete periods, node i will run exactly mn more times than node j. A CPASS, on the other hand, imposes that after each cycle terminates, all input/output nodes have been executed the same number of times. So a CPASS does not exist for such $\vec{q}$.

□

Our algorithm for finding a CPASS will begin by checking these two necessary conditions. We now need a sufficient condition that asserts that a CPASS exists for a given TTSDF graph. To that purpose we will characterize a class of algorithms and prove that if a CPASS exists the algorithm will find it, and return failure if not.

Definition 8. Class C algorithms ("C" for callback) : An ith node is said to be *runnable* at a given position if it has not been run $\vec{q}_i$ times and running it will not cause any buffer size to go negative.

Given a latency l, a positive integer vector $\vec{q}$ such that $\Gamma\vec{q} = \vec{0}$ and an initial state for the buffers $\vec{b}(0)$, a *class C algorithm* is any algorithm that

1. First, initialises $\vec{b}(0)$ adding delays to the incoming arcs of output nodes (as many delays as l executions of the outputs will consume), defines the prologue σ as any sequence in $(\wp(I)\wp(O))^l$, and updates $\vec{b}$ accordingly.

2. And second, schedules the period ϕ with the given $\vec{q}$: schedules any node if it is *runnable* and does not break the callback order, and updates $\vec{b}$, and stops only when no more nodes are runnable. If the periodic schedule terminates before it has scheduled each node i q_i times it is said to be *deadlocked*, else it terminates successfully.

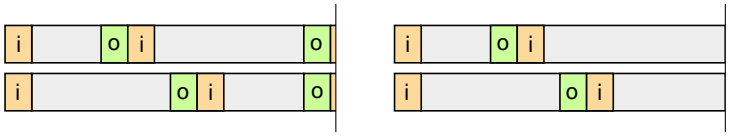
Lemma 1. *To determine whether a node* x *in an SDF graph can be scheduled at time* i*, it is sufficient to know how many times* x *and its predecessors have been scheduled, and to know* b(0)*, the initial state of the buffers. That is, we need not know in what order the predecessors where scheduled nor what other nodes have been scheduled in between.*

Proof. To schedule a node α, each of its input buffer must have sufficient data. The size of the input buffer j at time i is given by the jth entry in the vector $\vec{b}(i)$. And by definition of the computation model we know that

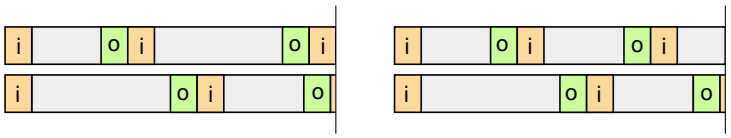
$$\vec{b}(i) = \vec{b}(0) + \Gamma\vec{q}(i)$$
$$\vec{q}(i) = \sum_{n=0}^{i-1} \vec{v}(n)$$

The vector $\vec{q}(i)$ only contains information about how many times each node has been invoked before iteration i, and not the order.

□



(a) Pairs of "in-phase" callback-order sequences.



(b) Pairs of "out-of-phase callback-order sequences."

Figure 4.8: In-phase and out-of-phase callback activations. (Here **i** and **o** represent $\wp(I)$ and $\wp(O)$). This diagram makes intuitive the proof of lemma 2 "Sequence permutations are in callback-activation phase". Note that sequences not in callback activations cannot be permutations since they have different number of non-input/output nodes.

Lemma 2. Sequence permutations are in "callback-activation phase"*: Two sequences ϕ and χ in callback order such that one is a permutation of the other, either both end with a node in $V-(I\cup O)$ (that is, not necessarily the same) or both end with the same node in $I\cup O$ (and so, both ends with the same prefix of $\wp(I)$ or $\wp(O)$)*

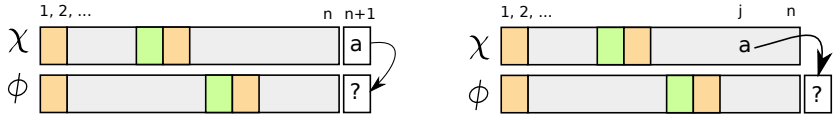
Figure 4.8 shows callback-order sequences both in-phase and out-of-phase, and illustrates the lemma's proof.

Proof. By contraposition: Assume that a sequence, say ϕ, ends with a $\wp(I)$ or $\wp(O)$ sub-sequence and χ ends with a node different than the last node in ϕ. Then, the ending $\wp(I)$ or $\wp(O)$ sub-sequence of ϕ is different than the ending sequence of χ. But incomplete sub-sequences of $\wp(I)$ and $\wp(O)$ are only allowed at the end of the sequence, by callback order definition. Therefore, ϕ and χ differ in the total number of nodes in I or O, and they cannot be permutations.

□

The following theorem builds upon the previous lemas and is the heart of our scheduling algorithm proof.

Theorem 2. Sufficient condition*: Given a TTSDF with topology matrix Γ such that $rank(\Gamma) = s-1$ and given a positive integer vector $\vec{q}$ such that $\Gamma\vec{q}=0$, and all input/output node i have equal $\vec{q}_i$ (the "necessary conditions"); If a* latency-l CPASS *of period $p=\vec{1}^T\vec{q}$ exists, any class* C *algorithm will find such a CPASS.*



(a) $\chi(n)$ and $\phi(n)$ are permutations, then the $n+1$th element in ϕ can be scheduled.

(b) $\chi(n)$ and $\phi(n)$ are not permutations, then an element a exists that is not input and can be scheduled.

Figure 4.9: Finding the $(n+1)$th element to schedule, assuming that a scheduling exist.

In other words: successful completion of any class C algorithm is a *sufficient condition* for the existence of the CPASS.

Proof. It is sufficient to prove that if an l-latency TTSDF scheduling (σ, ϕ) of latency l and period p exists, a class C algorithm will find such scheduling. That is, it will not deadlock before the termination condition is satisfied.

Trivially, any class C algorithm will find a prologue equivalent to σ, since the different permutations $\wp(I)$ and $\wp(O)$ do not affect the runnability of further nodes.

Let's now demonstrate that such an algorithm will find the periodic part of the scheduling. We need to show that if the algorithm schedules $\chi(n)$ for the first n executions, where $0 \leq n < p$, it will not deadlock for its $(n+1)$th execution before $n = p$.

Lets proceed doing a case analysis of the nth element in $\chi(n)$ (that is the last element scheduled).

1. If the nth element in $\chi(n)$ is an input but its $\wp(I)$ sub-sequence has not been completed then the next input in $\wp(I)$ will be scheduled.

2. If the nth element in $\chi(n)$ is an output but its $\wp(O)$ sub-sequence has not been completed then the next output in $\wp(O)$ will be scheduled.

3. If the nth element in $\chi(n)$ is an output that closes a complete $\wp(O)$ sequence, the first element in $\wp(I)$ will be scheduled. This will happen because the callback order of $\chi(n)$ guarantees that, given that $n < p$ (that is, we have not reached the end), at least another callback activation is to be scheduled.

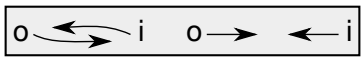


Figure 4.10: Pushing sources towards the beginning and sinks towards the end does not affect the scheduling runnability. Here i and o represents any source and sink, not just inputs and outputs.

4. The remaining cases —the non trivial ones— are that the nth element scheduled is either the last of a $\wp(I)$ sub-sequence or is a node in $V - (I \cup O)$ (a non input/output).

   Since an l-latency CPASS exists, assume that $\phi(n)$ are the n first entries of the periodic part of such CPASS.

   (a) If $\chi(n)$ is a permutation of $\phi(n)$ then the $(n+1)$th element in ϕ is runnable by lemma 1 and 2. —in a nutshell, lemma 1 says that the runnability of a node depends on how many times its predecessors have run, but not their order, and lemma 2 says that permutations end at the same part of a callback activation, and hence the $(n+1)$th element in ϕ will not break the callback order. See figure 4.9a.

   (b) If $\chi(n)$ is not a permutation of $\phi(n)$ then at least one node appears more times in $\phi(n)$ than in $\chi(n)$. See figure 4.9b.

   Let α be the first such node. We can prove that $\alpha \notin I$. This can be seen by contraposition: Let's assume that $\alpha \in I$. Let j be the position that α takes in ϕ. $\phi(j-1)$ and $\chi(j-1)$ are clearly permutations. Since ϕ is in callback order, α is preceded by $\wp(O)$ and zero or more inputs. But lemma 2 states that two permutations in callback order must end at the same phase of the callback activation. That means that the whole $\wp(O)\wp(I)$ sub-sequence will remain in permutation order, and α cannot be the first different node.

   (c) In the other cases, $\alpha \in O$ or $\alpha \in V - (I \cup O)$, α can be scheduled as the $(n+1)$th element in ϕ because it is runnable by lemma 1 and it will not break the callback order.

□

Therefore, we know fore sure that the if a graph is schedulable, any algorithm in the *class C algorithms* will find such a schedule. Next we proof that any TTSDF

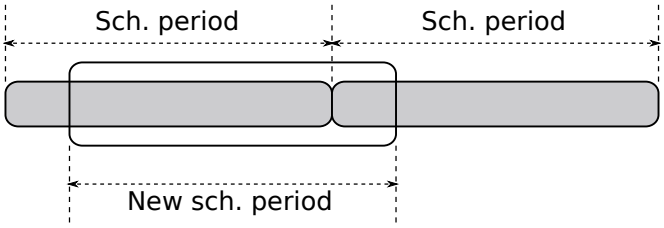


Figure 4.11: Adding two schedule periods and splitting them to create a new schedule period.

graph that has an SDF-style scheduling will also have a TTSDF-style scheduling. Therefore, both models have equivalent computability.

Theorem 3. TTSDF and SDF have equivalent computability*: If a TTSDF has a PASS, then it also has an l-latency CPASS for some $l < c$, where c (for "callback activations") is the number of inputs/outputs in a period (formally, $c = \vec{q}(i)$ where i is any index of an input or output)*

Proof. We can prove it by constructing a p-latency CPASS out of a PASS. Let Γ be the topology matrix of the graph, and assume that the PASS is characterised by a period of executions ϕ, and has been calculated with a given $\vec{q}$ such that $\Gamma\vec{q} = \vec{0}$

First, note that on any synchronous dataflow scheduling, source nodes can be pushed towards the beginning and sink nodes can be pushed toward the end. Such modifications of the scheduling sequence result in increased buffer sizes, but they do not change the runnability of any successor node. See figure 4.10.

Also note that we can transform a periodic schedule ϕ into a prologue and periodic part, by spanning two periods and slicing it with the same period size. To be precise, for any given n the new prologue is the n first elements in ϕ, and the new period is the last elements in ϕ starting from the $n+1$th concatenated, again, with the first n elements in ϕ. See figure 4.11.

With the previous observations, we will now show how to transform a PASS into a p-latency CPASS.

1. Push all input nodes in ϕ to the beginning, forming sub-sequences of $\wp(I)$. And push all output nodes to the end, forming sub-sequences of $\wp(O)$.

$$\phi' = (\wp(I)\wp(I)...\wp(O)\wp(O))$$

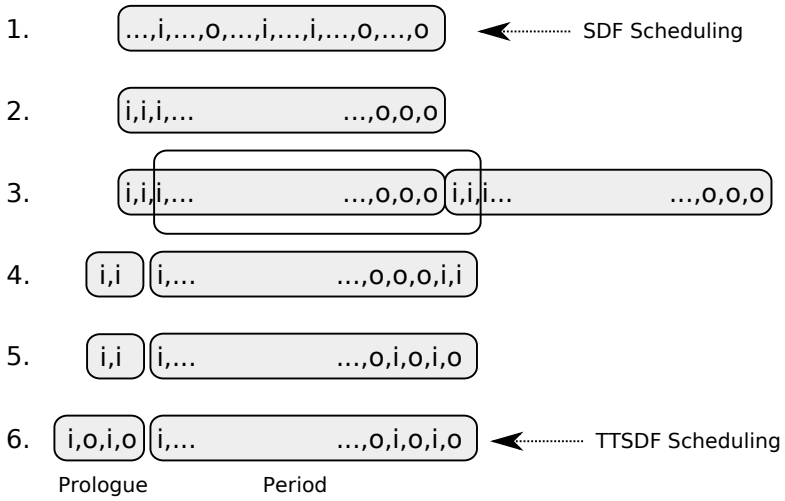


Figure 4.12: Steps to convert an SDF-style scheduling to a TTSDF-style scheduling. The result, however, is not optimal neither in latency nor in run-time load between callback activations. But it proofs that if an SDF-scheduling exists for a graph a TTSDF-scheduling also exists. And, in that case, the TTSDF scheduling (class C) algorithm will find an optimal scheduling in terms of latency (by means of testing the algorithm with different latencies from 0 to p) .

2. Let n be the number of $\wp(I)$s in ϕ'. We define the prologue σ as $\wp(I)^{n-1}$, and the new period ϕ'' as ϕ' taking out the prologue and concatenating it to the end, applying the aforementioned technique of slicing two spanned periods.

3. Modify ϕ'' by pushing the $n-1$ $\wp(I)$ sub-sequences towards the beginning, next to each $\wp(O)$.

4. Add $n-1$ $\wp(O)$s at the prologue which will be executed with the added delays of the CPASS.

□

These steps that convert an SDF-style scheduling to a TTSDF-style scheduling are illustrated in figure 4.12.

Corollary 1. *We can choose the smallest $\vec{q}$ in the null-space of Γ to find the schedule.*

Proof. We can choose the smallest $\vec{q}$ for finding a PASS (a SDF scheduling) as proved in [Lee and Messerschmitt, 1987a]. Theorem 3 (equivalent computability) guarantees that with the same $\vec{q}$ a CPASS exist.

In other words, the SDF model have the nice property for which the smallest vector in the nullspace of Γ is as good as any other in order to find a scheduling. By using this smallest vector in the algorithm we have the guaranty of obtaining the smallest scheduling cycle. Corollary 1 says that, given that this is true for SDF, it is also true for TTSDF.

4.4

The TTSDF Scheduling Algorithm

Given the theorems and proofs in the previous section, we now propose a specific scheduling algorithm that falls into the defined *class C algorithms*, and therefore, will find an l-latency CPASS, if one exists. The algorithm takes the following inputs: Topology matrix Γ, number of nodes s, added latency l, initial delays $\vec{b}_0$

The algorithm, using pseudo-code in Python-like syntax:

```
def ttsdf_schedule :
  # necessary conditions:
  if rank(Γ) != s-1:
    return "Rate mismatch"
  q⃗ = smallest_integer_in_null-space(Γ)
  I = arbitrary list of input nodes
  O = arbitrary list of output nodes
  L = arbit. list of non input/output nodes
  for n in O+I : if q⃗(n) != q⃗(O_0) :
```

```
    return "In/outs should have same rate"
  v_inputs = sum(v_i for each i in I}
  v_outputs = sum(v_o for each o in O}
  prologue_schd = (IO)^l
  # set added latency buffering
  b = b_0 + Γ * v_inputs * l
  x = 0 # the current number of executions
  activation_closed=True
  while x != q :
    found_any_runnable = False
    if activation_closed:
      cycle_schd += I
      b += Γ * v_inputs
      activation_closed = False
    for n in L:
      # run node n if runnable
      if min(b + Γ * v(n))>=0 and x(n)<q(n) :
        cycle_schd += [n]
        x(n) += 1
        b += Γ * v(n)
        found_runnable = True
    # run all output nodes if runnable
    if min(b + Γ * v_outputs)>=0 and x(O_0)<q(O_0) :
      cycle_schd += O
      for o in O : x(o) += 1
      x += Γ * v_outputs
      found_runnable = True
      activation_closed = True
    if not found_runnable :
      #lacks initial buffering
      return "DEADLOCK"
  return prologue_schd, cycle_schd
```

Finding the minimum l for an l-CPASS using the previous function is easy. We begin testing for $l = 0$ and increase l until found, or its upper bound c is reached. c, is the number of callback activations, or the number of times each input/output appears in a period ($c = \vec{q}(i)$, where i is any input or output position).

```
for l in [0,c] :
  result = ttsdf_schedule(l)
  if result is not deadlock:
    return result
return "DEADLOCK"
```

4.4.1 Cost Analysis of the Scheduling Algorithm

The *ttsdf_schedule* function finds an schedule —if it exists— given a latency l. It first computes the smallest integer vector $\vec{q}$ in the nullspace of the topology matrix Γ. This is quadratic with respect to the number of nodes n, so $O(n^2)$. The second phase of the *ttsdf_schedule* function constructs a cyclic scheduling of length p. Each scheduled node is, basically, found by a simple iteration on the nodes list. Therefore, its maximum cost is $O(pn)$. Given that $p > n$ in general, the final cost of the function is $O(n^2 + pn) = O(pn)$.

The optimal scheduling is found by repeated calls to *ttsdf_schedule* giving l values from 0 to c. The upper bound c, is number of callback activations in a period. Therefore, the total cost of the optimal scheduling algorithm is $O(cpn)$. Since c is bounded by p, the final expression can be reduced to $O(p^2 n)$.

Finally, note that p and n are totally independent values (apart form the restriction $p \geq n$), because p depends on the *token rates* values specified on the graph.

4.4.2 TTSDF Scheduling Example

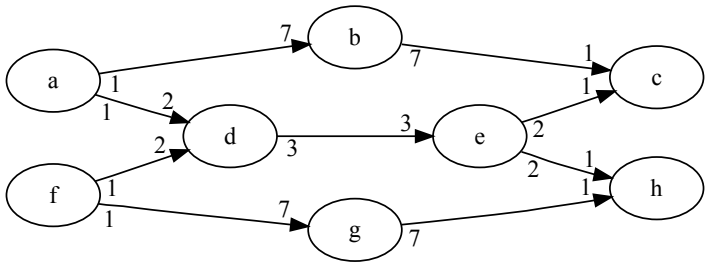
This section shows a scheduling example that results from executing the previous algorithm on a TTSDF graph. Six more examples can be found in appendix B — namely: "Simple TTSDF Pipeline", "Split and Reunify", "Dense Graph", "Graph with Optimum Between Bounds", "Audio and Video Multi-Rate Pipelines", "Simplified Modem". Another example can be found as part of the *3D-Audio Dataflow* case study, in section 6.2.4.

Graph with Optimum Between Bounds

Callback activations in the resulting scheduling are separated using parentheses "( )". The prologue and periodic parts are separated with a "+". We also give

a non-time-triggered scheduling obtained with an SDF scheduling algorithm, in order to compare TTSDF and SDF schedulings, Finally, we also give a diagram with the evolution of buffering during the periodic cycle. This diagram shows the total amount of tokens (that is, summing all the FIFO queues in the graph).

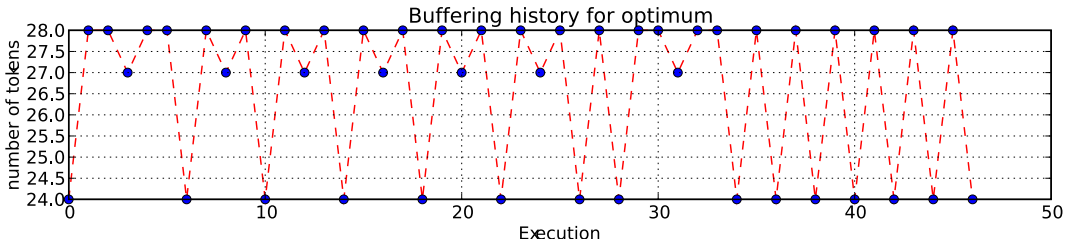
This example graph (and its name) is taken from [Ade et al., 1997].



$$\Gamma = \begin{bmatrix} 1 & -7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 7 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & -3 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & -1 \\ 0 & 0 & 0 & -2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & -1 \end{bmatrix}$$

- Executions per period $\vec{q} = \{14, 2, 14, 7, 7, 14, 2, 14\}$ corresponding to nodes: *a, b, c, d, e, f, g, h*

- Time-Triggered scheduling. Prologue + period: $(\mathbf{a}_0, \mathbf{f}_0, \mathbf{c}_0, \mathbf{h}_0)$, $(\mathbf{a}_1, \mathbf{f}_1, \mathbf{c}_1, \mathbf{h}_1)$, $(\mathbf{a}_2, \mathbf{f}_2, \mathbf{c}_2, \mathbf{h}_2)$, $(\mathbf{a}_3, \mathbf{f}_3, \mathbf{c}_3, \mathbf{h}_3)$, $(\mathbf{a}_4, \mathbf{f}_4, \mathbf{c}_4)$, $\mathbf{h}_4)$, $(\mathbf{a}_5, \mathbf{f}_5, \mathbf{c}_5)$, $\mathbf{h}_5)$ + $(\mathbf{a}_0, \mathbf{f}_0, b_0, d_0, e_0, g_0, \mathbf{c}_0)$, $\mathbf{h}_0)$, $(\mathbf{a}_1, \mathbf{f}_1, d_1, e_1, \mathbf{c}_1, \mathbf{h}_1)$, $(\mathbf{a}_2, \mathbf{f}_2, d_2, e_2, \mathbf{c}_2, \mathbf{h}_2)$, $(\mathbf{a}_3, \mathbf{f}_3, d_3, e_3, \mathbf{c}_3, \mathbf{h}_3)$, $(\mathbf{a}_4, \mathbf{f}_4, d_4, e_4, \mathbf{c}_4, \mathbf{h}_4)$, $(\mathbf{a}_5, \mathbf{f}_5, d_5, e_5, \mathbf{c}_5, \mathbf{h}_5)$, $(\mathbf{a}_6, \mathbf{f}_6, \mathbf{c}_6, \mathbf{h}_6)$, $(\mathbf{a}_7, \mathbf{f}_7, b_1, d_6, e_6, g_1, \mathbf{c}_7, \mathbf{h}_7)$, $(\mathbf{a}_8, \mathbf{f}_8, \mathbf{c}_8, \mathbf{h}_8)$, $(\mathbf{a}_9, \mathbf{f}_9, \mathbf{c}_9, \mathbf{h}_9)$, $(\mathbf{a}_{10}, \mathbf{f}_{10}, \mathbf{c}_{10}, \mathbf{h}_{10})$, $(\mathbf{a}_{11}, \mathbf{f}_{11}, \mathbf{c}_{11}, \mathbf{h}_{11})$, $(\mathbf{a}_{12}, \mathbf{f}_{12}, \mathbf{c}_{12}, \mathbf{h}_{12})$, $(\mathbf{a}_{13}, \mathbf{f}_{13}, \mathbf{c}_{13}, \mathbf{h}_{13})$



- Optimal latency added by the TTSDF schedule prologue: 6 callback activations.

- SDF scheduling (non Time-Triggered) $\mathbf{a}_0$, $\mathbf{f}_0$, $\mathbf{a}_1$, $\mathbf{f}_1$, $\mathbf{a}_2$, d_0, e_0, $\mathbf{f}_2$, $\mathbf{a}_3$, $\mathbf{f}_3$, $\mathbf{a}_4$, d_1, e_1, $\mathbf{f}_4$, $\mathbf{a}_5$, $\mathbf{f}_5$, $\mathbf{a}_6$, b_0, $\mathbf{c}_0$, d_2, e_2, $\mathbf{f}_6$, g_0, $\mathbf{h}_0$ (completed first output), $\mathbf{a}_7$, $\mathbf{c}_1$, $\mathbf{f}_7$, $\mathbf{h}_1$, $\mathbf{a}_8$, $\mathbf{c}_2$, d_3, e_3, $\mathbf{f}_8$, $\mathbf{h}_2$, $\mathbf{a}_9$, $\mathbf{c}_3$, $\mathbf{f}_9$, $\mathbf{h}_3$, $\mathbf{a}_{10}$, $\mathbf{c}_4$, d_4, e_4, $\mathbf{f}_{10}$, $\mathbf{h}_4$, $\mathbf{a}_{11}$, $\mathbf{c}_5$, $\mathbf{f}_{11}$, $\mathbf{h}_5$, $\mathbf{a}_{12}$, $\mathbf{c}_6$, d_5, e_5, $\mathbf{f}_{12}$, $\mathbf{h}_6$, $\mathbf{a}_{13}$, b_1, $\mathbf{c}_7$, $\mathbf{f}_{13}$, g_1, $\mathbf{h}_7$, $\mathbf{c}_8$, d_6, e_6, $\mathbf{h}_8$, $\mathbf{c}_9$, $\mathbf{h}_9$, $\mathbf{c}_{10}$, $\mathbf{h}_{10}$, $\mathbf{c}_{11}$, $\mathbf{h}_{11}$, $\mathbf{c}_{12}$, $\mathbf{h}_{12}$, $\mathbf{c}_{13}$, $\mathbf{h}_{13}$

- Initial latency added by the SDF schedule (that is, number of input executions without a correspondent output execution): 7 callback activations.

This example shows a graph that in spite of being rather small (8 nodes), exhibits multiple port rates and therefore has a long scheduling period (62 executions per period). The TTSDF-scheduling algorithm predicts that, in order to run the graph in real-time without gaps or jitter, the first 6 callback activations will be spent only to buffer inputs into the internal FIFO's, and from the 7th the graph will produce output continuously.

Compared to the presented SDF scheduling, the TTSDF scheduling have a slightly better latency (6 callbacks against 7). But two considerations are in order here:

- First, in order to run the SDF scheduling in a real-time system some extra infrastructure is needed: a separate thread that collects inputs and outputs from the hardware device and manages buffering between the device and the dataflow graph. This results in a run-time penalty against its real-time performance, related to extra cost of the inter-process communication and context-switching needed.

- Second, and more important: although in this example the SDF scheduling was reasonably good, no guarantee exists that this will be the case with another algorithm (also in the class of SDF algorithms) or with another graph. That is, the worst case scenario in terms of maximum latency and gaps at the outputs (or jitter) is not determined.

4.5

The Parallel TTSDF Scheduling

Constructing parallel schedules for the TTSDF model is basically equivalent to the SDF case. Consequently, in this thesis we summarise the existing techniques.

The multiprocessor dataflow scheduling problem can be reduced to a familiar problem in the well established *operations research* field, for which good heuristic methods are available.

The first step is to construct an *acyclic precedence graph* for j periods of a CPASS ϕ. Given an acyclic precedence graph, the problem of constructing a parallel schedule is identical with parallelizing assembly line problems in operations research. This problem can be solved for the optimal schedule —that is, a schedule with optimal CPU usage—, though the solution is combinatorial in complexity. If the TTSDF graph to be scheduled is small, the solution complexity is not a problem. For large ones we can use well studied heuristic methods in the *critical path* methods family [Adam et al., 1974]. One simple algorithm in this family that closely approximates an optimal solution for most graphs is known as *Hu-level* scheduling algorithm [Kohler, 1975, Yu-Kwong, 1996].

The acyclic precedence graph captures the dependency between nodes executions —where "dependency" means that the execution of a particular node is necessary for the invocation of another node. The algorithm for constructing an acyclic graph is simple and can be found in [Lee and Messerschmitt, 1987a].

The acyclic precedence graph algorithm needs to be implemented consistently with the callback-order restrictions of the TTSDF computation model. With such a precedence graph, the final parallel scheduling computed will also be in callback-order, and thus easy to run in callback activations.

4.6

Applying the Time-Triggered Scheduling to Other Dataflows

Other dataflow models could apply the TTSDF scheduling algorithm, thus gaining real-time capabilities. The obvious target are dataflow models, beyond SDF's, that allow for static scheduling, at least in practical cases.

An example is the Boolean-controlled Dataflow (BSDF) model (see section 2.2.5), which allows the rates of actors to change in response to external control inputs. Such relaxation of the SDF constrains makes static scheduling undecidable. However, algorithms exist for computing static schedules in most practical cases.

A second approach to port rates reconfiguration is to use each configuration state as a state of an extended finite state machine or *modal model*, as in the **-charts* and *Heterochronos* model [Girault et al., 1999]. A third approach to reconfiguration consists on using parameterized token rates, as in the *Parameterized dataflow* model [Bhattacharya and Bhattacharyya, 2001].

In many cases, static scheduling can still be performed by representing token rates symbolically and generating a symbolic or *quasi-static* schedule. In hierarchical heterogeneous dataflow systems, such reconfigurations are allowed to occur at *quiescent points* in the herarchical execution of the model [Neuendorffer and Lee, 2004], [Neuendorffer, 2005].

We have shown that any correct SDF with callback semantics —that is, with some sources and sinks marked as inputs and outputs, all with the same firing rate, and that do not deadlock— also has a TTSDF schedule —a cyclic schedule that can be split into several callback activations. Therefore, we can apply the TTSDF techniques to extend all such less-restricted models to support schedules that fit in a timed environment with time-triggered callbacks. An open issue here is that any token rate reconfiguration can change the amount of added latency.

Since operating with fixed latency should be desirable, some kind of configurations analysis should be necessary to find the maximum latency.

4.7

Summary

Actor-oriented models of computation offer better abstractions than prevailing software engineering techniques when the goal is building real-time multimedia processing systems. The family of Process Networks and Dataflow models are the most suited for continuous stream processing. Such models allow the developer to express the designs close to the problem domain —instead of focusing in implementation details such as threads synchronization—, and enable better modularization and hierarchical composition. This is possible because the model does not over-specify how the actors must run, but only imposes data dependencies in a declarative language fashion.

Process Networks and Dataflows models, however, does not handle time in a useful way to provide real-time multimedia computation. Specifically, they do not adapt well to time-triggered callback-based architectures. This chapter has presented the *Time-Triggered SDF (TTSDF)*, a new model that mends the real-time limitations of Dataflow models because:

- Combines a set of actors associated to a (single) timed-trigger with a set of actors which are untimed.

- Retains token causality and dataflow semantics (arguably the added delay could be considered an exception), without compromising its calculability, which is equivalent to SDF.

- Avoids jitter and does so by using an optimum amount of latency. A superior bound for that latency is also given by the model.

- Naturally fits in callback-based architecture: the periodic dataflow scheduling is split into smaller sequences which can efficiently execute within the callback, avoiding external buffering and further threads.

Apart from introducing real-time capabilities, the TTSDF model offers further benefits such as:

- It enables static analysis for optimum distribution of run-time load among callbacks.

- It can be parallelized using well known techniques in the *operations research* field [Kohler, 1975] also used in other dataflow models.

- The callback-triggered style of scheduling can be adapted to other dataflow models with quasi-static token rates, such as *Boolean-controlled SDF*, or with dynamic (but explicit) token rates, like *Dynamic Dataflow*.

Let us now summarize the present chapter: we start underlining the limitations of existing Synchronous Dataflow scheduling algorithms when the application needs to operate in real-time. Specifically, in such algorithms it is impossible to avoid gaps and jitter in the output, nor guarantee an optimal latency.

We then define a concrete syntax and semantic for the TTSDF graphs. A computational model is formalized to allow us stating theorems and to reason about the model schedulability. These formalizations introduces the key concepts of sequences in *callback-order*, and scheduling cycles split in *callback activations*.

We precisely define what a TTSDF admissible scheduling is, and propose and prove three theorems —and lemmas on which the theorems build on— about the model schedulability. The first theorem specifies the *necessary conditions* (or preconditions) that a graph must guarantee to be able to run the scheduling algorithm. Specifically, it needs *rate consistency* and the same execution rate for all inputs and outputs. The second theorem gives the *sufficient conditions* for the existence of a time-triggered cyclic scheduling. It proves that a generic class of algorithms defined as *class C* ("C" for callback) will find a scheduling if such scheduling exist or will return failure if not. The third and last theorem proves that the TTSDF and SDF have equivalent computability, meaning that any valid

TTSDF graph that have an SDF schedule, also have a TTSDF schedule with minimum latency.

A detailed TTSDF scheduling algorithm with optimal latency is then given. The computational cost (in the worst case) is $O(pn^2)$, where p is the size of the periodic scheduling, and n the number of nodes. Finally, a parallel scheduling is summarized. The parallelized scheduling problem is reduced to a well known problem in *operations research* where heuristics exist that find the optimal solution in terms of CPU usage.

4.7.1 Applicability and Future Work

We have recently implemented the TTSDF model in the open-source CLAM framework[2] (see section 6.1). Prior to using the TTSDF model, CLAM used an SDF scheduler that was only able to schedule trivial graphs in a callback-based architecture. Thus, it could not cope with most multi-rate models. Several real-time multi-rate applications have been tested in CLAM in the domains of audio features analysis, spectral audio transformation and virtual-reality 3D audio generation. See 6.2 for a description of the latter case.

Some of the multimedia sub-domains that we believe might take advantage of multi-rate dataflows are: video coding and decoding, multi-frame-rate video processing, joint video and audio processing, audio processing in the spectral domain, real-time video and audio feature extraction using arbitrary sized token windows, and real-time computer graphics.

Many lines are open for future work. Open architectures that leverage parallelism and multiprocessors but are still compatible with callbacks for input and output should be studied. Such architectures would not see the users callback as a black-box function, but would have access to the dataflow graph declaration. Further, techniques for balancing the run-time load, not only between sequential callbacks, but multiple processors, should be developed.

The next chapter addresses the topic of how to design complete software systems using Dataflow models of computation —such as the TTSDF model. In next chapter, the focus is put on a systematic approach to software development based on interrelated design patterns.

[2] http://clam-project.org

CHAPTER 5

A Pattern Language for Dataflow-based Multimedia Processing Systems

We reviewed (in section 3.4) previous efforts on building pattern languages for the dataflow paradigm. This chapter offers a pattern language for dataflow-based multimedia processing systems. All the presented patterns fit within the generic Dataflow Flow Architecture pattern (reviewed in section 3.4.1).

The Dataflow Architecture pattern solves the problem of designing a system that performs some number of sorted operations on similar data elements (that we call *tokens*) in a flexible way, so they can dynamically change the overall functionality without compromising performance. The pattern solution is an architecture that can be seen as a *network* of *modules* with strict interfaces at the module boundary, allowing a large number of possible combinations.

Modules read incoming tokens through their in-ports and writes them through their out-ports. Module connections are done by connecting out-ports to in-ports, forming a network.

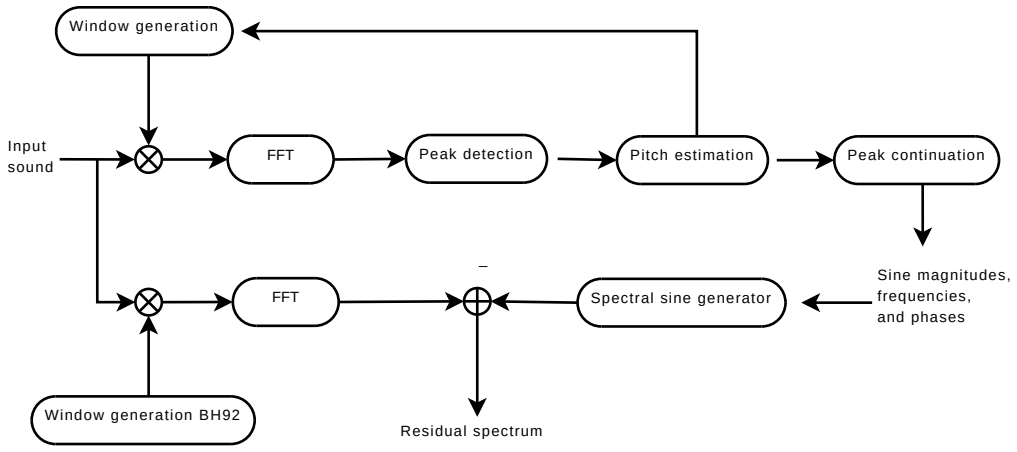


Figure 5.1: A use case for audio dataflow: the Spectral Modeling Synthesis.

In multimedia processing systems, tokens flow through modules in two different fashions: at regular (or almost regular) rate, which is known as a *stream* flow; and without any regularity, which is known as *event* flow. For example, the flow of data coming from an audio card is a stream flow, while the flow of note-on and note-off messages from a MIDI keyboard is an event flow.

Each module in a network is periodically *executed*, which means a call to the module's *execution method* (also known as *module's algorithm*).

It is important to note that the Dataflow Architecture pattern does not impose any restrictions on issues like message passing protocol, module execution scheduling, or data token implementation. All these aspects imply different problems that can be addressed in other fine-grained patterns, like the ones in the present pattern language. This pattern granularity [Vlissides, 1998] proved very useful because we have been able to incorporate orthogonal patterns that work synergistically among them and with the existing ones from Manolescu.

The proposed patterns are inspired by our experience in the audio domain. And some patterns are clearly motivated by the requirements of spectral-domain processing. A use case that exemplifies its complexity is the analysis-synthesis using sinusoids plus residual (see figure 5.1), where different Fast Fourier Transforms are done consuming different number of tokens (audio samples) in parallel.

5.1

Chosen Pattern Structure

The following pattern structure has been chosen for all our patterns. Adherence to a structure facilitates browsing the catalog and comparing patterns.

Context and Problem Statement Sets the solution space. Defines what is an "admissible" solution and what is not. Problem statement is just the core statement of the problem. Should be punchy and easy to remember. But it is not different from context in essence.

Forces Do not define the solution space but give criteria on what is a good solution and what is a bad one. In other words, the quality-of-services that we want to optimize.

Solution The architecture/design/implementation that solves the problem, without giving many justifications. The given solution should make clear that it belongs to the solution space.

Consequences They justify why the solution is a good one in terms of the stated forces. That is, why all the forces are optimized (or resolved) or how the forces are balanced in case they are conflicting.

Related Patterns Reference higher-level patterns describing the context in which this pattern can be applied, and lower-level patterns that could be used to further refine the solution, as well as other used or similar patterns.

Examples Give a list of real-life systems where the pattern can be found implemented.

Taking into account the previously introduced background, the patterns contributed in this thesis are presented in the next 3 sections, organized in three categories:

- *General Dataflow Patterns:* Address problems about how to organize high-level aspects of the dataflow architecture, by having different types of modules connections.
- *Flow Implementation Patterns:* Address how to physically transfer tokens from one module to another, according to the types of flow defined by the *general dataflow patterns*. Tokens life-cycle, ownership and memory management are recurrent issues in these patterns.
- *Network Usability Patterns:* Address how humans can interact with dataflow networks without compromising the network processing efficiency.
- *Visual Prototyping Patterns:* Addresses how domain experts can generate applications on top of a dataflow network, with interactive GUI, without needing programming skills.

5.2

General Dataflow Patterns

5.2.1 Pattern: Semantic Ports

Context

Applications with a dataflow architecture consist on a directed graph of modules, like the one shown in figure 5.2. It is a very common case that a module receives tokens with different semantics. For example, a module that mixes n audio channels will receive tokens of audio data corresponding to each channel. Identifying which token corresponds to each channel —the token semantics— is fundamental to produce output tokens containing the audio mix. The Payloads pattern described by Manolescu provides a solution to this problem consisting on adding a

descriptor component into each token which provides the semantic information about the token, as well as type-specific parameters. The implication of applying Payloads is that incoming tokens needs to be dispatched according to its descriptor component, before doing any processing.

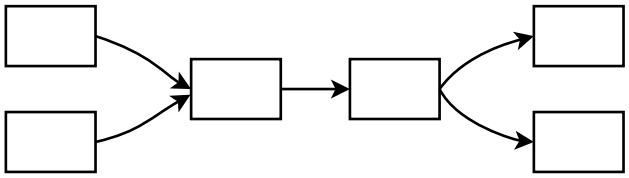


Figure 5.2: A directed graph of components forming a network

Tokens produced by a module may also have different semantics. One might want to send to a connected module only tokens with a given semantics and not all the produced tokens.

Problem

How can a module manage tokens according to their semantics in order to deal with the incoming ones in different ways and send the produced ones to different destinations?

Forces

- Module implementation should be as simple as possible, because modules are developed by different authors while general infrastructure is just implemented once by experienced programmers.
- Dispatching tokens adds complexity to module programming
- Module execution should be efficient in time, often real-time constrains are imposed.
- Dispatching tokens adds a run-time overhead.
- Token semantics fields on tokens add overhead
- Token semantics should be given by the module and they should not be restricted.
- Incoming might also have different priorities, and modules should consume the tokens with greatest priority first.

Solution

Use different ports for every different token semantics in each module, instead of using the same port for different kind of messages. Modules should have as many in-ports and out-ports as different input and output semantics are needed. Instead of connecting modules directly, connect modules by pairing out-ports with in-ports, as shown in figure 5.3. Module's execution method knows the semantics associated to each port, thus, it can obtain tokens of specific semantics just by picking the proper in-port. Because connections are done among ports instead of modules, a processed tokens will target the proper destination just by sending tokens through the proper out-port.

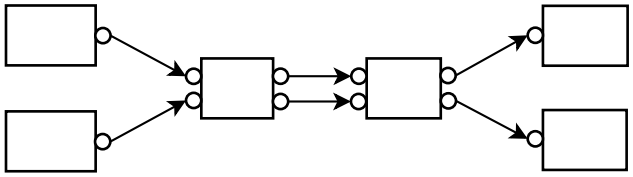


Figure 5.3: A network of components with multiple ports

Consequences

Tokens do not need to incorporate a description component. Module implementation is simplified because programming a token dispatcher regarding its semantics is not needed. Also, run-time penalty associated with dispatching is avoided. The pattern solution implies that token semantics is not defined inside the tokens with a description component, but semantics is something intrinsic to the ports.

Retaking the audio mixer example; instead of having a "channel" field on each token arriving to the mixer, using the Semantic Ports pattern, we would have a mixer with n different in-ports, each one receiving tokens of a single channel.

Tokens with different priorities should be routed to different in-ports. The module knows the priority of each in-port and so is able —in its execution method— to consume tokens in the right order.

Related Patterns

Most patterns in this collection build on Semantic Ports: Driver Ports, Stream and Event Ports, and Propagating Event Ports are clear examples of separation of ports

regarding its semantics.

Semantic Ports also relates to Payloads in the sense that the problems they solve are similar but, since they have different forces, they end up with different solutions.

Semantic Ports can handle different token types by using the Typed Connection pattern.

Examples

CLAM uses Semantic Ports to separate different flows. Visual environments like Pure-Data (PD) [Puckette, 1997] or MAX/MSP [Puckette, 1991] also do. Their ports separate both audio ("tilde") streams lower rate streams on their semantic. We find another good examples in Open Sound World (OSW) [Chaudhary et al., 1999] and the JACK sound server [Davis et al., 2004].

Anti-examples —systems that do not use *Semantic Ports* because they use other approaches— are also interesting to see for this pattern: Marsyas [Tzanetakis and Cook, 2002] and SndObj [Lazzarini, 2001] do not use separated ports for their network connections but they do it at module level. SndObj modules, for instance, keep a pointer to their connected producers and read their input tokens doing a direct call.

5.2.2 Pattern: Driver Ports

Context

Module execution on dataflow system is driven by the availability of flowing tokens. But not all token flows drive the execution.

Imagine a module that receives an audio signal and performs a low-pass filter with a given cutoff frequency. The audio signal is fed into the module with a constant rate but the cutoff values are seldom fed into the module. These cutoff values typically come from a sequencer module or a knob in the user interface. Each execution of the module must wait for the availability of new audio signal data. But there is not such dependency on the seldom received cutoff values, it just uses the last value. To summarize, whereas audio stream tokens drive the modules execution, the frequency event tokens does not.

Problem

How can we make module execution depend on the availability of tokens on certain in-ports and not on others?

Forces

- Concrete module implementation should be simple.
- Visual programming tools should be able to distinguish the flow that drives the module execution from the one that does not.

Solution

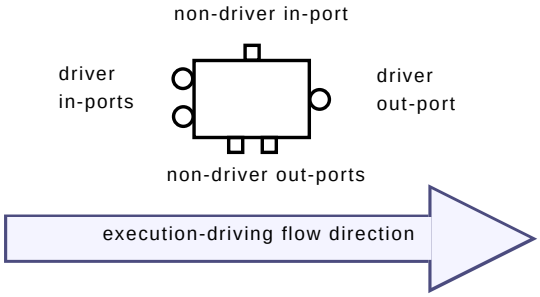


Figure 5.4: A representation of a module with different types of in-ports and out-ports

Allow the concrete module developer to define which are the driver in-ports and which are not. Give the modules a common interface from which external entities can know which are the drivers and which are not. The module execution will be enabled by the availability of enough tokens on the driver in-ports. Note that enabling is not the same as triggering. The network scheduling policy determines if a module will be executed as soon as it is able —in a *pull* strategy— or if it will be postponed until other module executions end.

This solution is rather general and it can be implemented with different strategies. A concrete design is shown in figure 5.5. This class collaboration separates the general infrastructure in base classes making the concrete classes simpler to implement —this is actually an example of *white box reuse* in frameworks. Some

module services are implemented in the base class, usually delegating to its ports, and freeing the concrete module writer from this responsibility.

It is important to note that although concrete modules create their port objects, the module base class aggregates them to provide a generic interface to all the ports. Examples of such operations are *ableToExecute* which can be useful to a firing manager (or scheduler); and *driverPorts*/*nonDriverPorts* that give the lists of driver and non-driver ports to, say, a GUI client.

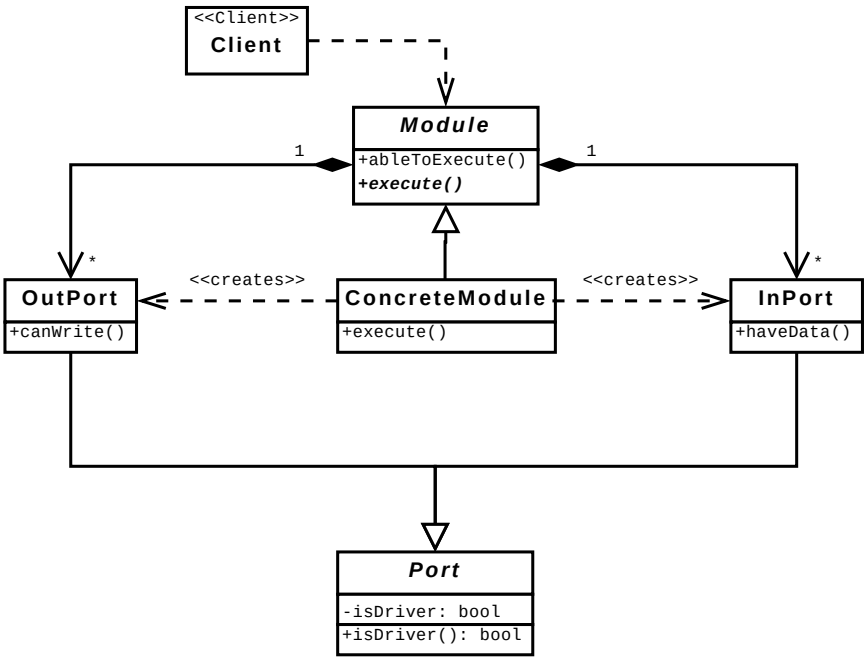


Figure 5.5: Separated Module and ConcreteModule classes, to reuse behaviour among modules

Other patterns like Stream and Event Ports and Typed Connections also benefit from using this class structure. However, each pattern enriches the *Port* and *Module* base class interfaces to fit its needs.

Consequences

Separating driver and non-driver ports makes it possible to check whether a module is ready to be executed or not without relying on the concrete module implementation which gets simpler and safer to programming errors.

Visual builder tools can distinguish driver and non-driver flows by identifying driver and non-driver ports and displaying them differently.

As mentioned in [Foote, 1988] module networks are often built with visual programming tools. Such tools should give the user a clear separation between stream ports and event ports, else, event connections might hide the main dataflow —the stream flow that drives the modules execution.

For example, CLAM's visual builder called Network Editor (see figure 5.6) uses horizontal connections (left to right) for driver flow, and vertical (top-down) connections for the non-driver flow.

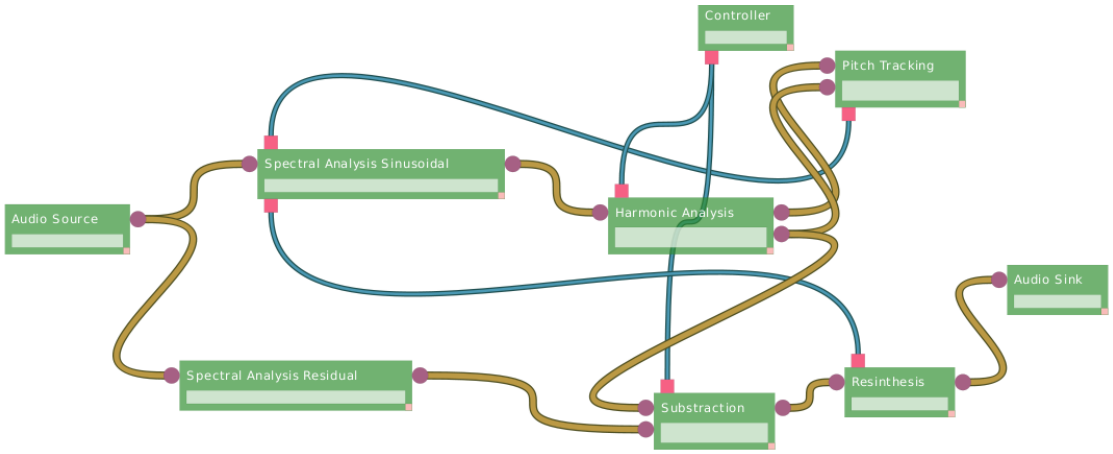


Figure 5.6: Screenshot of CLAM visual builder (NetworkEditor) performing spectral analysis and synthesis

Other visual builders takes different approaches. Open Sound World (OSW), for instance, paints the driver ports in green while the non-driver ports are gray. This can be appreciated —though if the copy is not colored it can be hard— in figure 5.7.

Related Patterns

Driver Ports is strongly related to Stream and Event Ports. Driver ports tend to be stream ports. However, they are better off being separate patterns because they solve orthogonal problems. Moreover, examples exist where driver ports and stream ports are totally independent.

Token availability conditions are complex when connected stream ports produce and consume tokens at different rates. Multi-rate Stream Ports pattern solves this.

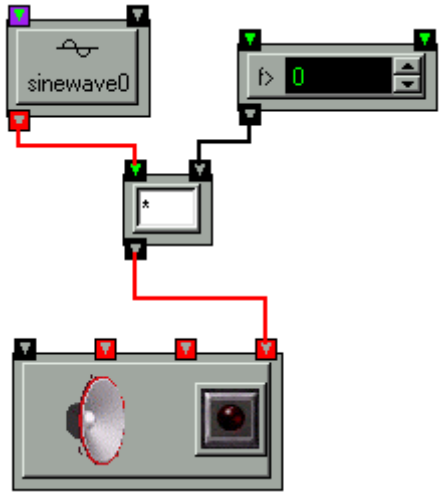


Figure 5.7: Screenshot of Open Sound World visual builder

Examples

Pure Data (PD) [Puckette, 1997] and MAX/MSP [Puckette, 1991] are graphical programming environments for real-time musical applications that are widely used by composers. Its ports are called *inlets* and *outlets* and they are visually arranged horizontally. With few exceptions (notably the "timer"), objects treat their left-most inlet as "hot" in the sense that messages to left inlets can result in output messages. Other inlets are "cold", they only store the received message and do not trigger any execution. Thus, the "hot" or leftmost inlets are the driver ports. However, since modules have only one driver port and modules are executed at the time a token arrives at the driver port, the following problematic situation may occur when two modules are connected by more than one connection: the module might be triggered before receiving all its data because the "hot" inlet was not the last to receive the data. In order to avoid this output messages are —by convention— written from right to left and modules connections should be (visually) done without any crossing lines. Finally, PD and MAX/MSP is important example where driver ports do not coincide with stream ports.

Open Sound World (OSW) [Chaudhary et al., 1999] has a similar approach to PD and MAX/MSP but it does not limit the number of driver ports. In the JACK audio server [Davis et al., 2004] all ports are drivers. CLAM also uses *Driver Ports* and restricts its drivers to be constant-rate stream ports.

5.2.3 Pattern: Stream and Event Ports

Context

A module may receive tokens of both kinds —stream and event— coming from different sources. Moreover, streams may arrive at different rates. For example, a module may receive two audio samples streams one at 44100 Hz and the other at 22050 Hz. Figure 5.8 shows another example: a module is receiving two streams at different (though constant) rates and an irregularly distributed flow of events. Its output stream has the same rate as the second input stream.

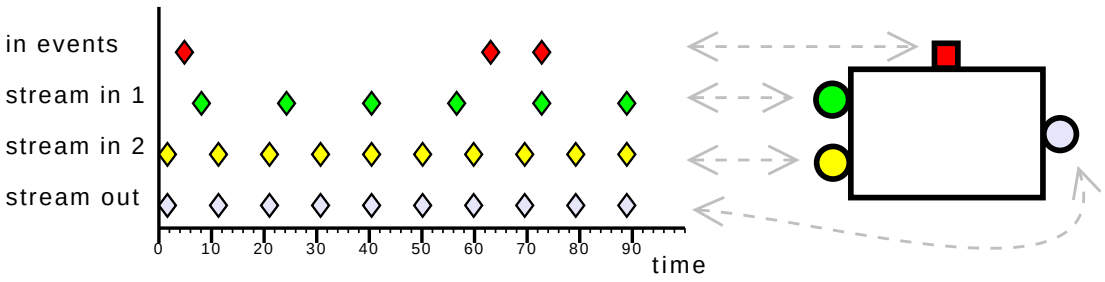


Figure 5.8: Chronogram of the arrival (and departure) time of stream and event tokens

Such a module consumes its incoming tokens and then calls its execution method which will take the consumed tokens as its input. When receiving tokens at different rates, the module needs to synchronize all the incoming tokens prior to its processing. This synchronization can also be seen as a time-alignment of incoming tokens, and it implies knowing the time associated to each token. Here, is important to differentiate the time associated to the tokens, with the "real" time where the module is executed. The two kinds of time might be totally different. Figure 5.9 illustrates the alignment of tokes of different nature.

While incoming stream tokens always needs to be accurately aligned, this is not always true for incoming event tokens. Some applications requires a precise alignment of event tokens, while others admit a loose time alignment.

An obvious approach is to use the Payloads pattern, adding a precise time information —*time-stamp*— to each token. In real-time systems, this time-stamp relates to the time when the token is introduced into the system. In non real-time systems it relates to a virtual time. Transformations on a token should preserve

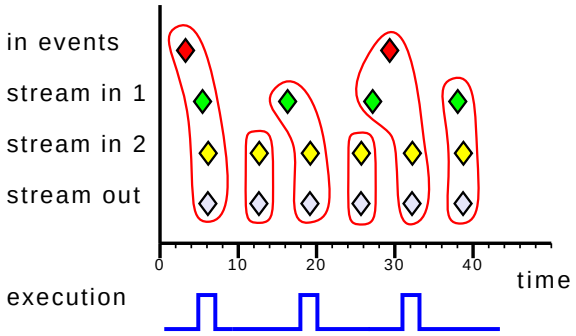


Figure 5.9: Alignment of incoming tokens in each execution. Note that time corresponds to token's time-information and does not relate to the module execution time (though they are equally spaced).

the original time-stamp. However, this Payloads approach can be overkill when stream tokens flow at a high rate, as it happens with audio samples.

Problem

Syncronizing incoming tokens requires time information. How can we get the time information of incoming tokens?

Forces

- Time-stamp is an overhead when the data token is relatively small.
- Propagating timestamps from the consumed tokens to the produced tokens is a run-time overhead and makes concrete module implementation more complex.
- Concrete module implementation should be simple.
- All stream sources must share the same hardware clock, so that their (token) rates cannot vary among different streams.

Solution

Calculate time information of incoming stream tokens instead of using time-stamps. If the application needs accurate timing for events use time-stamps —only for event tokens—, else do not.

Separate the stream and event flow in different kinds of ports: stream and event ports. Place the stream timing responsibility into the stream in-port class. Stream

in-ports are initially configured with a "port-rate" and "first-token-time" values, and they also keep the sequence —with a counter, for example— of consumed tokens. When a module execution method asks the stream in-port for new tokens to consume, the in-port provides the time information along with the tokens itself.

Port parameters ("port-rate" and "first-token-time") configuration is a key issue to solve. Two main approaches exist: ports handshaking and centralized management.

Ports handshaking consists in propagating parameters down-stream. In-ports receives parameters from their connected out-ports, and modules propagate them from in-ports to out-ports. In most cases, modules only need to copy them from the in-ports to out-ports. However, in some cases, the module processing may introduce delay and may change the port-rate; thus, this must be reflected in the out-port settings. In consequence, modules do not impose port parameters, they receives it and propagate them. Of course, source modules [1] are the exception to that rule. They must set the out-port parameters, because they are the *source* of the stream.

The second approach —centralized management— consist in incorporating an entity that orchestrates the configuration —and maybe the modules execution— of the whole network. This configuration manager is responsible for configuring all the stream ports in the network.

Alignment of event tokens with stream tokens is done in slightly different ways depending on whether the application needs accurate event timing or not —that is, whether they incorporate time-stamps or not. Note that on each execution, the module may consume not only one stream token but a bunch of them. In some cases, like with audio samples, even a large number of them like, say, 1000. If incoming events are time-stamped, the module knows the time information for all the incoming tokens, thus the module can align each event token with the stream tokens precisely. If events are not time-stamped, the module should align all consumed events with the first consumed stream token of each in-port.

Consequences

Making the stream tokens time implicit avoids space overhead. Event tokens may have time-stamps, but it is not required. Time-stamps do not have as much

[1] Source modules are that ones that do have stream out-ports but do not have any stream in-ports

overhead in event tokens as in stream tokens, since they flow non continuously, in much lower frequency than streams.

Events Jitter: Having a big number of stream tokens to be consumed on each execution and having non time-stamped event tokens at the same time is a common cause for jitter. That is, unsteadiness or irregular variation on the time the system responds to incoming events. The amount of jitter is bounded by the time interval between executions, which is proportional to the number of stream tokens consumed on each execution. Thus, making modules consume fewer stream tokens each time, reduces jitter. Of course, when events comes in with time-stamps, jitter can be eliminated completely. The consequences of having jitter varies enormously depending on the concrete application. In most cases jitter can be neglected, in other cases, however reducing it is paramount.

Ports connectivity: The solution forbids time-stamps in the stream tokens and this restricts how stream ports can be connected. Because time must be inferred from the incoming stream sequence, in-ports must receive well formed sequences —without gaps, etc.— of an individual stream. Therefore, in general, N-to-1 connections of stream ports are to be forbidden by the system. However, an exception to this rule exists when the in-port is able to implicitly perform a combining operation prior to keeping track of the incoming sequence. For example, imagine an in-port of audio samples fed by multiple different streams. Parallel samples are added, forming an audio mix. Because the mix is a single token, the in-port assigns time the same way as if the source were unique. Apart from addition, packing —that is, create a new composite token— is another common combining operation.

In general, multiple stream combinations is better handled explicitly in specific modules. It gives the system designer flexibility to choose his or her combining operation. Moreover, a system can have token types without any valid combining operation, thus making implicit combinations impossible.

We have seen that, in general, N-to-1 connections of stream ports are to be forbidden by the system. On the other hand, N-to-1 connections of event ports are perfectly fine, since there is no need to infer the token time information from the order of arrival. Time information is either read from the time-stamp or simply ignored.

Splitting one stream to multiple streams is a different story: 1-to-N connections of both stream and event ports are allowed. The consideration that has to be done here is how to duplicate outgoing tokens flowing to multiple destinations. Two strategies exist: one is making a copy of each outgoing token to every in-port, and the other is passing a managed reference to each in-port. In the later case, the in-ports will have to enforce read-only semantics.

Related Patterns

Stream and Event Ports usually goes together with Driver Ports and, in most of the cases, stream ports are also driver ports.

Systems that use Stream and Event Ports may also use Multi-rate Stream Ports for designing the stream ports —allowing stream ports to consume and produce at different cadences—, and may use Propagating Event Ports for its event ports —allowing event ports to propagate events immediately.

Stream ports designed with the Multi-rate Stream Ports pattern defines the number of released tokens for each stream port on each execution. This numbers influence how the ports "port-rate" settings are propagated —in this case, being re-calculated— from in-ports to out-ports.

Stream and Event Ports can handle different token types by using the Typed Connection pattern.

Examples

SuperCollider3 [McCartney, 2002] and CSL [Pope and Ramakrishnan, 2003] use the pattern but events cannot arrive at any time: they have a dual rate system, control rate and audio rate, with the control rate being a divisor of the audio block rate.

Marsyas [Tzanetakis and Cook, 2002] uses this pattern though its event ports do not follow the dataflow architecture because connections are not done explicitly. It implements the token packing technique (from multiple sources) as mentioned in the Ports Connectivity section above.

CLAM and OSW [Chaudhary et al., 1999] use this pattern, separating ports for streams and for events.

5.2.4 Pattern: Typed Connections

Context

Most simple audio applications have a single type of token: the sample or the sample buffer. But more elaborated processing applications must manage some other kinds of tokens such as spectra, spectral peaks, MFCC's, MIDI... You may not even want to limit the supported types. The same applies to events channels, we could limit them to floating point types but we may use structured events controls like the ones OSC [Wright, 1998] allows.

Heterogeneous data could be handled in a generic way (common abstract class, void pointers...) but this adds a dynamic type handling overhead to modules. Module programmers would have to deal with this complexity and this is not desirable. It is better to directly provide them the proper token type. Besides that, coupling the communication channel between modules with the actual token type is good because this eases the channel internal buffers management.

But using typed connections may imply that the entity that handles the connections should deal with all the possible types. This could imply, at least, that the connection entity would have a maintainability problem. And it could even be unfeasible to manage when the set of those token types is not known at compilation time, but at run-time, for example, when we use plugins.

Problem

Connectable entities communicate typed tokens but there is an unlimited number of types of tokens. Thus, how can a connection maker do typed connections without knowing the types?

Forces

- Process needs to be very efficient and avoid dynamic type checking and handling.
- Connections are done in run-time by the user, so they can mismatch the token type.
- Dynamic type handling is a complex and error prone programming task, thus, placing it on the connection infrastructure is preferable than placing it on concrete modules implementation.

- Token buffering among modules can be implemented in a wiser way by knowing the concrete token type rather than just knowing an abstract base class.
- The set of token types evolves and grows.
- A connection maker coupled to the evolving set of types is a maintenance workhorse.
- A type could be added in run time.

Solution

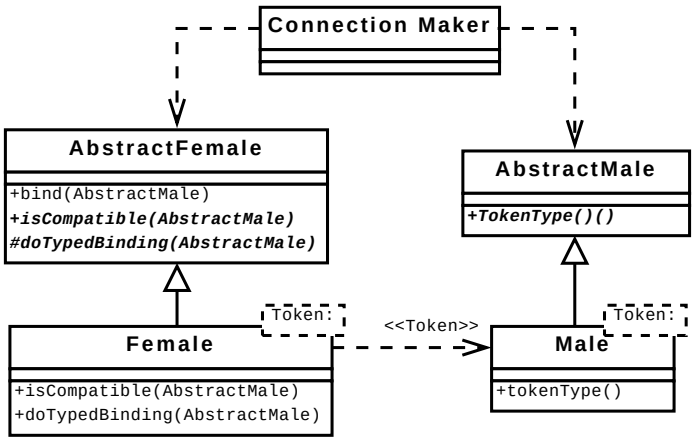


Figure 5.10: Class diagram of a cannonical solution of Typed Connections

Split complementary ports interfaces into an abstract level, which is independent of the token-type, and a derived level that is coupled to the token-type. The class diagram of this solution is shown in figure 5.10.

Let the connection maker set the connections through the generic interface, while the actual bind ins done in the subclasses to be type-safe. Also, the connected entities use the token-type coupled interface to communicate with each other efficiently.

Use run-time type checks when modules get connected (*binding time*) to get sure that connected ports types are compatible, and, once they are correctly connected (*processing time*), rely just on compile-time type checks.

To do that, the generic connection method on the abstract interface (`bind`) should delegate the dynamic type checking to the concrete (token-type coupled) classes using the abstract methods `isCompatible`, `tokenType` and `doTypedBinding`. The implementation in C++ of listing 5.1 shows two classes

that gets connected by a pointer in the concrete female pointing to the concrete male. Depending on the situation, a pointer is not enough and a (token-type coupled) "connection" entity is needed.

Consequences

By applying the solution, the connection maker is not coupled to token types. Just concrete modules are coupled to the token types they use.

Type safety is ensured by checking the dynamic type on binding time and relying on compile-time type checks during processing time. So this is both efficient and safe.

Because both sides on the connection know the token type, buffering structures can deal with tokens efficiently during allocation, initialization, copy, etc.

Concrete modules just access to the static typed tokens. So, no dynamic type handling is needed.

Besides the static type, connection checking gives the ability to do extra checks on the connecting entities such as semantic type information. For example, implementations of the bind method could check that the size and scale of audio spectra match.

Related Patterns

This pattern enriches Multi-rate Stream Ports and Event Ports, and can be also useful for the binding of the visualization and the Port Monitor.

The proposed implementation of Typed Connections uses the Template Method [Gamma et al., 1995] to call the concrete binding method from the generic interface.

Examples

OSW [Chaudhary et al., 1999] uses Typed Connections to allow incorporating custom data types.

The CLAM framework uses this pattern notably on several pluggable pairs such as in and out ports and in and out controls, which are, in addition, examples of the Multi-rate Stream Ports and Event Ports patterns.

But the Typed connection pattern in CLAM is not limited to port like pairs. For example, CLAM implements sound descriptors extractor modules which have

```
#include <typeinfo>

class AbstractFemale
{
public:
    void bind( AbstractMale& male ) {
        if ( isCompatible(male) )
              doTypedBinding(male);
        else
            throw InvalidTypes();
    }
    virtual void isCompatible(
                  const AbstractMale& male
                  ) = 0;

protected:
    virtual void doTypedBinding(
                  AbstractMale& male
                  ) = 0;
}

template<class Token> class Female :
                      public AbstractFemale
{
public:
    bool isCompatible( const AbstractMale& male) {
        return typeid(Token) == male.tokenType();
    }
    void doTypedBinding( AbstractMale& male ) {
        _male = &dynamic_cast< Male<Token>& >(male);
    }
private:
    Male<Token>* _male;
};

class AbstractMale
{
public:
    const std::type_info& tokenType() = 0;
};

template<class Token> class Male :
                      public AbstractMale
{
public:
    const std::type_info& tokenType() {
        return typeid(Token);
    }
}
```

Listing 5.1: Sample C++ code for TypedConnections

ports directly connected to a descriptor container which stores them. The extractor and the container are type coupled but the connections are done as described in a configuration file, so handling generic typed connections is needed.

The Music Annotator [Amatriain et al., 2005] is a recent application which provides another example of non-port-like use of Typed Connections. Most of its views are type coupled and they are mostly plugins. Data to be visualized is read from an storage like the one before. A design based on the Typed Connection pattern is used in order to know which data on the schema is available to be viewed with each vista so that users can attach any view to any type compatible attribute on the storage.

5.3

Flow Implementation Patterns

5.3.1 Pattern: Propagating Event Ports

Context

Music and audio systems for real-time usage offer users interfaces, such as GUI slider or MIDI interfaces, to alter the processing while playing. Users should get fast feedback on their actions, so elapsed time between the event and its effect on the processing should be as short as possible.

Some modules transform such events and propagate them in a proper way to other modules. One simple design consists on propagating the incoming events during the module execution. However modules that receive the event may be executed before the module that is going to propagate the event, thus the event propagation can take too long.

Problem

How can we send event tokens and run associated actions on the receiver, including propagation, so that they get to the destination before other modules are executed?

Forces

- Module execution should be able to send events.
- Event token reception may imply changing the module state.
- Event token reception may imply sending new event tokens to other modules.
- Event token propagation should not be costly in respect to module execution in order not to break execution cadence and real-time restrictions.
- Coarse event tokens are hard to propagate by copy.
- Feedback loops on event ports should be allowed.
- 1 to N event ports connections should be allowed.
- N to 1 event ports connections should be allowed.
- All modules are executed from a single process.

Solution

Provide the concrete module implementers a way to bind a event in-port with a callback method to be called on event reception. This callback might change the module state, or propagate other events through the module event out-ports. Sending an event through an event out-port implies the immediate cascade execution of callback methods associated with every connected event in-port.

If event propagation by copy is too expensive, propagate event tokens using references instead of copies and make them read-only for the receiving modules. Limit the life of event tokens sent by reference to the cascade propagation and, forbid the receiving modules to keep references further than the callback execution.

Consequences

Event in-port callback implementers should be careful not to implement costly operations on them. Events may be sent in bursts; thus, expensive callbacks could break the real-time restrictions.

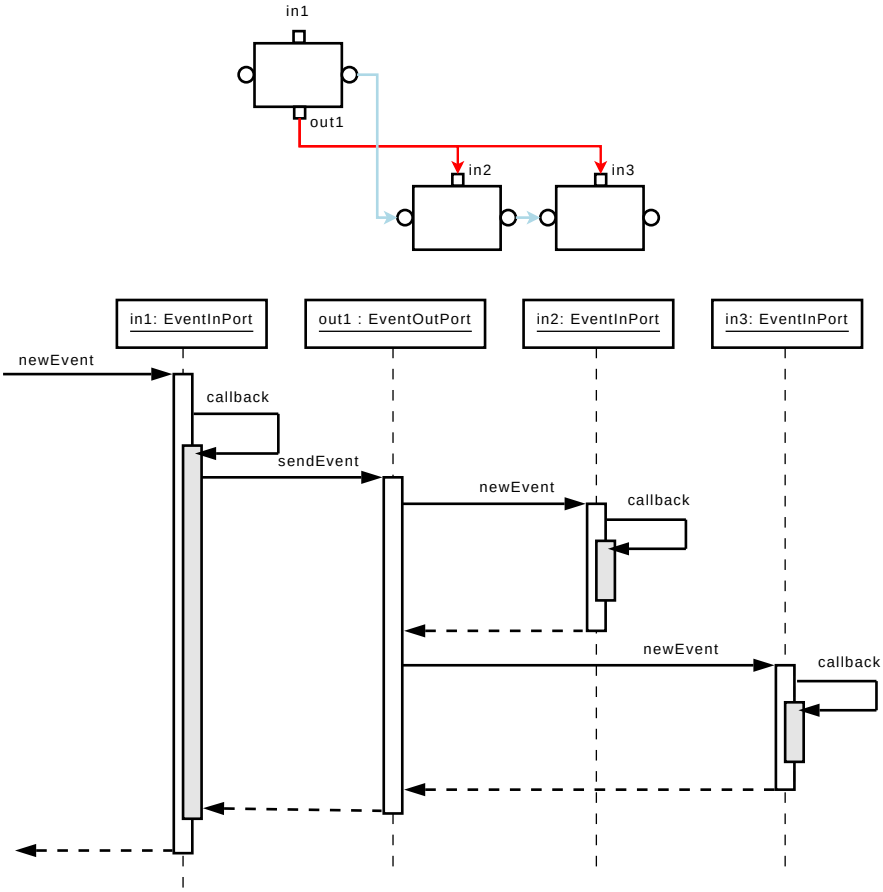


Figure 5.11: A scenario with propagating event ports and its sequence diagram.

Propagation of coarse events is something that could add penalty to in-port callbacks, but, by using references, this is avoided. Sending references could be dangerous when considering 1 to N connections, as one of the receiving modules may modify the event token. This is solved by making them read-only for the receiving modules.

Another danger associated to sending references is that modules might keep references to such tokens. Because of this, keeping references is forbidden, but we could loose this restriction by using reference counting on event tokens. The use of garbage collectors is not a good solution due to real-time restrictions.

The solution allows setting up loops on the event ports connection graph. Those loops might be harmless but they might be pernicious because the cascade callback calling enters in a non-ending loop. Harmful loops happen whenever the

call sequence reaches a port that was already involved on the cascade.

Static analysis of the network topology to warn the user about harmful loops is useless: not every event reception implies propagation on the event out-ports of the module; it depends on the callbacks methods. Because the sending of events is a synchronous call, one simple solution is to block sending tokens through a port which is already sending one. This is implemented just by adding a "sending-on-progress" flag in each port.

Related Patterns

Event tokens could be restricted to a given set of types, but we could also use the Typed Connections pattern to a more flexible solution.

Propagating Event Ports provides a flexible way for communicating the two partitions of the Out-of-band and in-band-partitions pattern (see a summary of this pattern in section3.4). The user interface partition communicates with the processing partition via connected event ports. Since both partitions are in different threads, a safe thread boundary must be established. Using, for example, the Message Queuing pattern [Douglass, 2003].

This pattern could be seen as a concrete adaptation of Observer [Gamma et al., 1995] to a dataflow domain where modules can act both as *observers* and *subjects*, in a way that they can be chained.

Examples

CLAM implements controls as propagating event ports. Multiple control inputs and outputs are supported. By default, events are copied as part of the module state but you can add a callback method to process each control in a special way. In its current version (0.91) event tokens are limited to floating point numbers.

PD [Puckette, 1997], MAX/MSP [Puckette, 1991] they all use propagating event ports for their non-audio-related modules "hot inlets".

5.3.2 Pattern: Multi-rate Stream Ports

Context

Many applications in the multimedia domain need to process chunks of consecutive audio samples in a single step. A common example, in the audio domain, is an

FFT transformation which consumes N audio sample tokens and produces a single spectrum token. Therefore, the rate of spectrum tokens is $\frac{1}{N}$ the sample rate. The FFT transformation may also need to process overlapping sample windows. That is, the FFT module reads N samples through an in-port and, after the execution, the window slides a step of M samples, where M and N are different. In this case, the rate of spectrum tokens is $\frac{M}{N}$ the rate of samples.

This example shows two different —though related— problems:

- Streams can flow at different rates. Like, for example, sample and spectrum streams do.
- Modules may need to process different numbers of tokens on each execution, regardless of the rate of its incoming streams. For example, an FFT module may require 512 samples while another FFT module may require 1024.

That means that the number of tokens a module consumes and produces should be flexible, allowing modules to operate with different consuming and producing rates.

How to approach this problem is not obvious. Some real-life systems[2] perform multi-rate processing inside their modules while restricting inter-module communication to a single rate. Since the number of tokens that a module's algorithm needs is not the same as the number of tokens consumed on each execution, input and output buffering inside the module is needed. A side effect of this approach (hiding multi-rate inside modules) is that the module execution not always implies its algorithm execution; when not enough tokens are ready for the algorithm, the module execution just adds incoming tokens to the internal buffers. Of course, this approach yields complex code in every concrete module.

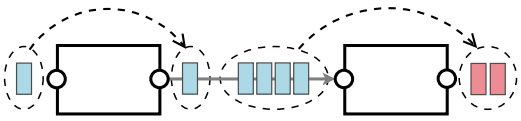


Figure 5.12: Two modules consuming and producing different numbers of tokens

[2]One example is the JACK [Davis et al., 2004] audio server, with the Jamin mastering tool.

Problem

How to allow modules to access a different number of consecutive tokens on every stream port?

Forces

- The number of accessed tokens and the number of released tokens is independent for each port.
- For a given port, the number of accessed tokens and the number of released tokens are unrelated.
- Modules have to process a sorted sequence of stream data tokens (usually a time sorted sequence)
- All the stream tokens have the same priority.
- An out-port could be connected to multiple in-ports so that the tokens produced might be consumed by different modules.
- A module may need access to a number of consecutive tokens for each incoming stream to be able to execute.
- A module execution may produce a number of consecutive tokens for each output stream.
- Arbitrary consuming and producing rates in a network renders static scheduling of executions impossible.
- Feedback loops should be allowed
- Copy of coarse tokens may be an important overhead to avoid.
- Concrete module implementation should be simple.
- All modules are executed from a single process.

Solution

Design stream ports so that they support consuming and producing at different rates. This way they can adapt to the rate the module's algorithm needs, while keeping the buffering details outside the module. The in-port and out-ports should give access to N tokens from a queue[3] and should release M tokens on every module

[3]By queue we mean the abstract data type with its generic operations without making assumptions on it implementation.

execution. Let the module developer define N and M for each port.

Give the ports an interface for accessing tokens —but only a window of N tokens at the head of the queue— and for releasing them. Releasing tokens means that M tokens will be dequeued and put away from the module reach. "Access" and "release" are operations implemented as port methods and they are to be called consecutively by the module execution method. Seen as a single operation, "access-and-release" is equivalent to "consume", when called on an in port, and "produce", when called on an out-port.

Make the ports own the tokens flowing between two modules, and make them responsible for all necessary buffering between out-ports and in-ports.

Buffers can be either associated with in-ports or with out-ports. In 1-to-N connections —that is, a single out-port connected to N in-ports— this decision makes the difference between heaving N different buffers (at the in-ports) or having a single buffer (at the out-port).

In the following paragraphs we discuss the implications of having buffers at the in-ports and at the out-ports.

Buffers at the in-ports: This is the simplest solution to implement. Give each in-port an associated buffer (figure 5.13). Tokens being produced from an out-port are then passed to the connected in-port buffers. Tokens can be either passed by reference or by copy. Passing tokens by copy is easy to implement since each in-port is the owner of its tokens. Passing references, on the other hand, is more efficient because copies are avoided. This efficiency gain can be very important when tokens are to be passed to many in-ports or when tokens are coarse objects.

Of course, the efficiency gain associated with passing references instead of copying comes at a price: it is more difficult to implement. Given that multiple modules receive a reference to the same token, aliasing problems have to be avoided. In-ports have to be designed in a way that guarantees read-only semantics —or copy-on-write semantics— on the incoming tokens. The other aspect to be addressed here is the tokens life-cycle. Since token memory cannot be freed —or recycled— while references to it exist, we need a reference-counting mechanism.[4]

However, passing references to the in-ports is not always feasible. Some applications may require their modules to operate on tokens placed on contiguous memory —examples of this are very common in the audio domain— as a conse-

[4]Like C++ smart pointers

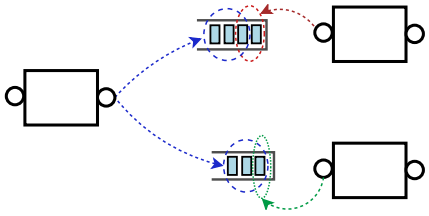


Figure 5.13: Each in-port having its own buffer.

quence, such modules need the actual tokens data (not references) placed together in circular buffers at the in-ports.

This shortcoming can be overcome placing the buffers at the out-ports, which allows having both reference passing and contiguity. Again, we will see that efficiency comes at a price.

Buffers at the out-ports: Having a single buffer for a 1-to-N connection — thus, associated to the out-port— allows benefiting from passing tokens by reference while achieving data contiguity (figure 5.14).

For that, the buffer must be implemented with a circular buffer. Not only the out-port is accessing the buffer but also the N connected in-ports.

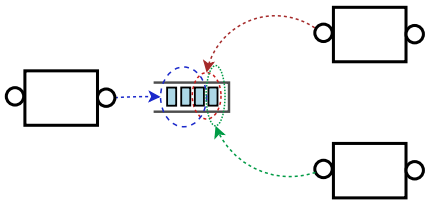


Figure 5.14: A buffer at the out-port is shared with two in-ports.

Allowing a buffer to be written by a producer and read by N different consumers, while allowing each one to produce or consume at a different cadence needs to done carefully. The following two basic restrictions must be enforced by design:

- The out-port cannot over-write tokens that still have to be read/consumed by some in-port.
- The in-ports cannot read/consume tokens that still have to be written/produced by the out-port.

Though complex to implement, this approach avoids the need for unnecessary copies of tokens and allows contiguous memory access to all involved modules.

Summing up: We have seen different strategies for implementing ports buffering that present a trade-off between simplicity and efficiency. Placing buffers at the in-ports and passing tokens by copy is the most simple approach. If copies are to be avoided, token references can be passed, but they have to be managed. Sometimes this is not enough. Apart from avoiding copies we need memory contiguity. Then, a circular buffer must be placed at the out-port and some restrictions must be enforced.

Consequences

Since all buffers adaptation is done at the ports level by the general infrastructure, concrete module implementors do not have to deal with buffers adaptation. That results in simpler, less error-prone code in every module.

Modules with different production and consumption rates can be connected together, as drawn in figure 5.12. As a result, this increases the number of the possible networks that can be built out of a set of modules.

The number of tokens stored in each port connection depends on two factors: In one hand, the requirements given by each port regarding the number of accessed and released tokens and, on the other hand, the scheduling policy in use.

This solution implies that, in general, the network will need a dynamic scheduler of module executions. To facilitate the task of such scheduler, modules may provide an interface to inform whether they are ready to execute or not. Such module method could be easily implemented in the module base class by delegating the question into every driver port, and returning the *and* combination if its responses.

Related Patterns

Multi-rate Stream Ports is applied in the context of systems that uses Stream and Event Ports, it addresses how stream ports can be designed so that they offer a flexible behavior.

Multiple Window Circular Buffer pattern addresses the low level implementation of the more complex variant of Multi-rate Stream Ports, that is *buffers at the out-ports*.

The design and implementation of *buffer at the out-ports* is a clear example of the Multiple Window Circular Buffer pattern.

Examples

In most of the systems reviewed, ports of the same type have all the same window size, and thus do not need to use this pattern. This is, for example, the case of the CSL [Pope and Ramakrishnan, 2003] and OSW [Chaudhary et al., 1999] frameworks and the visual programming tool MAX [Puckette, 2002].

On the other hand the Marsyas [Tzanetakis and Cook, 2002], SuperCollider3 [McCartney, 2002] and CLAM frameworks allow different window sizes, but they follow different approaches.

SuperCollider3 [McCartney, 2002] features variable block calculation and single sample calculation. For example, modules corresponding to different voices of a synthesizer may consume and produce different block sizes. The SuperCollider3 framework permits embedded graphs that have a block size witch is an integer multiple or division of the parent. This allows parts of a graph which may require large or single sample buffer sizes to be segregated allowing the rest of the graph to be performed more efficiently.

Marsyas allows buffer size adaptation using special modules. CLAM —probably for its bias towards the spectral domain— is the most flexible, allowing any port connection regardless of its window size. CLAM sets up a buffer at each out-port.

5.3.3 Pattern: Multiple Window Circular Buffer

Context

As a result of incorporating the Multi-rate Stream Ports pattern, the ports-connection queues need a complex behaviour, in order to access and release different number of tokens. Besides, such systems often have real-time requirements and some optimization factors must be taken into account: avoiding unnecessary copies, totally avoiding allocations and being able to work with contiguous tokens. Take for example (again) modules that performs the FFT transformation —delegating to some external library— upon a chunk of audio sample tokens; input samples must be provided to the library as an array. In the audio domain, not

only FFTs need to operate with arrays, temporal domain processing is typically done that way too.

A simple implementation of Multi-rate Stream Ports consists in having a buffer associated to each in-port. But, unfortunately, this means copying tokens. The copy-saving implementation of Multi-rate Stream Ports requires a single buffer to be shared by an out-port and many in-ports. A design for this is not obvious at all. So, this is what this pattern addresses.

Finally, note that though a normal circular buffer is not suited for accommodating the given requirements, what we are seeking may be seen as a "generalized" circular buffer. Moreover, this can be useful in scenarios other than dataflow architectures.

Problem

What design supports a single source of tokens with one writer and multiple readers, giving each one access to a subsequence of tokens?

Forces

- Each port must give access to a subsequence of N tokens (the window).
- The subsequence of tokens should be in contiguous memory, since many algorithms or domain tool-kits and libraries works on contiguous memory.
- Windows sizes and steps should all be independent.
- Reading windows can only map tokens that have been already produced through the writing window.
- Allocation during processing time should be avoided, since (normal) dynamic memory allocation breaks the real-time requirements.
- All buffer clients executes in the same process.

Solution

Have a contiguous circular buffer with windows that map (contiguous) portions of the buffer. There will be as many reading windows as needed but only one writing window. Associate the reading windows with the writing window because, as we will see they will need to calculate their relative distances. Also, provide them means for sliding along the circular buffer.

The modules (the buffer clients) executions must be done in the same thread. Its scheduling can be done either statically —fixed from the beginning— if all ports consuming and producing rates are known; or dynamically, which is much simpler to implement.

Windows clients need to follow the following protocol in order to avoid data inconsistencies:

- The access to windows mapped elements and the subsequent slide of the window must be done atomically in respect to other window operations. So, these operations might be regarded as a single read-and-slice (or write-and-slice) operation. Only when a window has finished the sliding, other clients can access their own window.
- A reading window can only start a read-and-slice (also known as *consume*) operation when it is not overlapping the writing window (overlapping other reading windows is perfectly fine). This reader-overlapping-writer problem indicates that the client is reading too fast. This problem should be detected and, as a response, the reading module should not be executed till more data has been written into the buffer.
- The writing window can only start a write-and-slice (or *produce*) operation when it is not overlapping the furthest reading region. Such overlapping is possible since regions are circulating over the underlying circular buffer. This writer-overlapping-reading problem indicates either that a client is reading too slow or that the buffer size is not large enough. When this is detected, the writing module should not be executed and should wait for the readers to advance.

The solution design uses the Layered pattern [Douglass, 2003] for arranging different semantic concepts at different layers. Concretely, we distinguish three levels of abstraction (figure 5.15). Starting from the layer that gives direct service to the clients:

Windows Layer This is the upper or more abstract layer, which gives the clients a view of the windows advancing on an infinite buffer. It offers, at least, the following interface :

- Accessing the N contiguous elements mapped by the window.

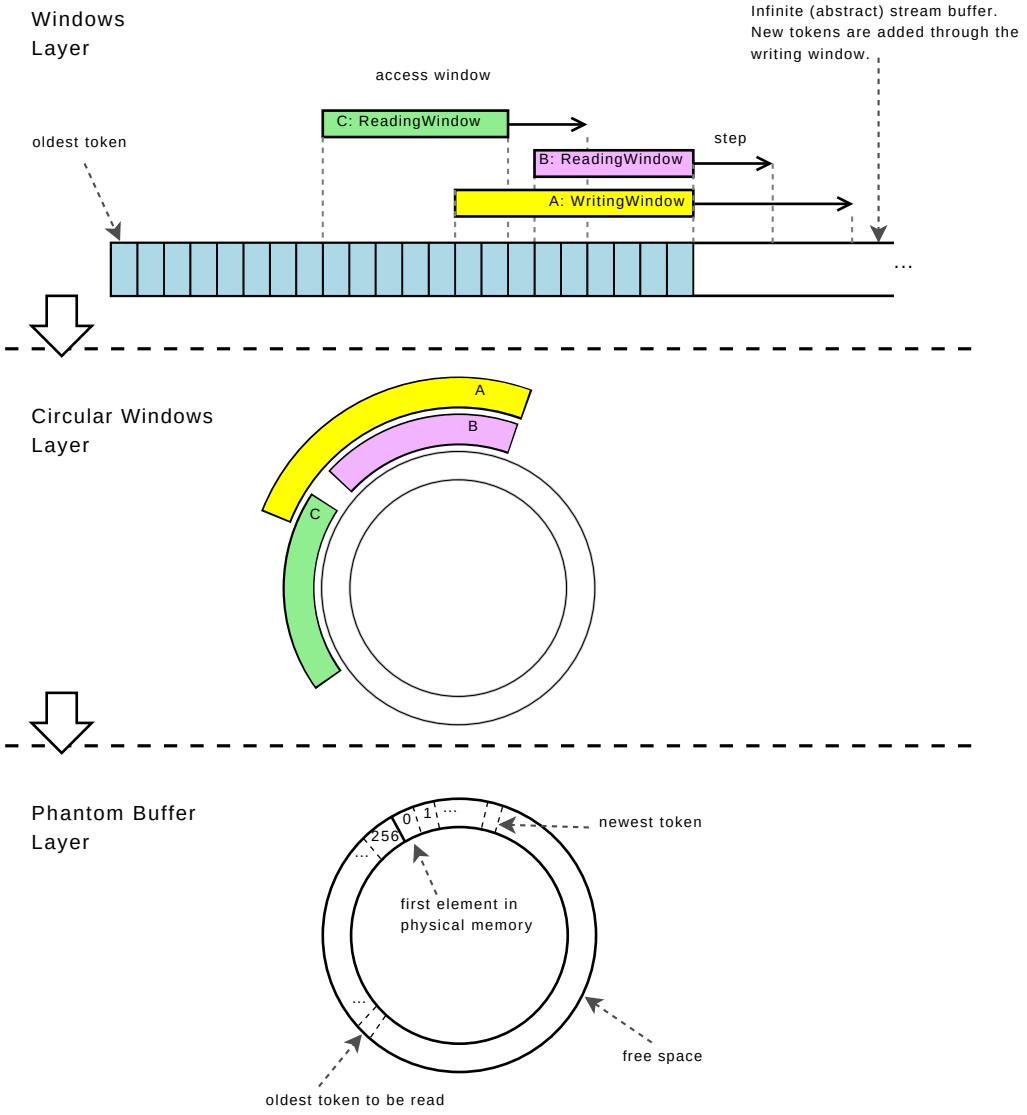


Figure 5.15: Layered design of port windows.

- Advancing a window its slicing step (not necessarily N) as if the windows where on an infinite buffer.
- Checking if a window is ready to be accessed-and-slided.

The state of this layer keeps the relative distances between each reading and the writing window. This layer is in charge of detecting when a reading window overlaps with the writing window, and delegates other checks to its underlying layer.

Circular Windows Layer The state of this layer keeps the physical pointers (to a circular buffer) for each window and also provides physical pointers to the upper layer. This layer is in charge of detecting —and preventing— circular overlapping with a reading window. That is, the case when the writer is about to write on a still not read element.

Phantom Buffer Layer This is the lower layer, which knows nothing about writing and reading and writing windows and is solely dedicated to provide chunks of contiguous elements for each window. Therefore, this layer's goal is to to provide contiguous elements subsequences of size N or smaller. Where N is the size of the biggest window.

The main problem this layer has to solve is the discontinuity problem associated to circular buffers —the next element in a logical sequence of the last physical element is the first physical element. To idea behind the solution is to replicate the first N elements at the end of the buffer. This can be implemented using a data structure that we call "Phantom Buffer" and is presented in this catalog as the Phantom Buffer pattern.

Consequences

The non-overlapping restrictions might suggest that there always exists a distance between writing and reading windows and, thus, causing the introduction of certain latency. But this is not the case, because the non-overlapping restrictions only apply, at the time of an access-and-slide operation. After a window has been slided, it is perfectly legal to be in an overlapping state. This allows the reading windows to consume the same tokens that the writing window has just produced.

The reader-too-slow and writer-too-slow problems can be handled in the context of a dynamic scheduler. Before doing any access-and-slide operation, a *canProduce()* or *canConsume()* check is done, so that the operation can be safely aborted.

The consequence of the layered approach is a flexible design that allows changing the underlying data structure easily, without affecting the windows layer and its client. It also eases the implementation task since the overall complexity is split in well balanced layers which can be implemented and tested separately. On the other hand, those many levels of indirections might carry a performance penalty. However, it should be note that the implementation does not require polymor-

phism at all. Thus, when implemented in C++, with a modern compiler, most of the indirections should be converted to in-line code by the compiler, reducing the function-calls overhead.

In general, setting the window parameters can be done at configuration time; that is, before the processing or module executions starts.

Related Patterns

This pattern solves the *buffer at the out-port* approach of the Multi-rate Stream Ports pattern, which was the optimal one. Multiple Window Circular Buffer uses the Layered pattern [Douglass, 2003].

Examples

This pattern is maybe a *proto-pattern* [www-PatternsEssential,] as the authors only know their own implementation in the CLAM framework [www-CLAM,]. Nevertheless, CLAM is a general purpose framework and several applications with different requirements have proven the value of the pattern.

5.3.4 Pattern: Phantom Buffer

Context

The goal of Multiple Window Circular Buffer is to design a generalized circular buffer where, instead of having a writing and a reading pointer, we deal with a writing window and multiple reading windows. The difference between a window and a plain pointer is that a window gives access to multiple elements that, moreover, need to be arranged in contiguous memory. Multiple Window Circular Buffer relies, for its elements storage, upon some data structure with the following functionalities: First, to be able to store a sequence of any size ranging from 0 to MAX elements. And second, to be able to store each subsequence up to N elements in contiguous memory.

A normal circular buffer efficiently implements a queue within a fixed block of memory. But in a normal circular buffer the contiguity guarantee does not hold: given an arbitrary element in the buffer, chances are that its next element in the (logical) sequence will be physically stored on the other extreme of the buffer.

Note that here window management is not relevant at all because it is a responsibility of upper layers. Thus, the only concern of this pattern is how the low-level memory storage is organized.

Problem

Which data structure holds the benefits of circular buffer while guaranteeing that each subsequence of N elements sits in contiguous memory?

Forces

- Element copies are an overhead to avoid.
- Buffer reallocations are to be avoided.
- It should be possible for clients to read and write a subsequence of elements using a pointer to the first element. The rationale is that modules might want to use existing libraries that typically use pointers as their input and output data interface.
- All buffer clients execute in the same process.

Solution

The *buffer with phantom zone* —*phantom buffer* for short— is a simple data structure built on top of an array of $MAX + N$ elements. Its main particularity is that the last N elements are a replication of the first N elements. This guarantees that starting at any physical position from 0 to $MAX - 1$, there exists a contiguous subsequence of size up to N elements. In effect, this is clear considering the worst case scenario: take the element at position $MAX - 1$; let it be the first one in a subsequence; since it is a circular buffer of MAX elements, the next element is in the position 0, but positions from 0 to $N - 1$ are also replicated at the end (starting at position MAX); thus the contiguity condition is guaranteed.

Interface of a PhantomBuffer class should include two methods: one for accessing a given window of elements, and the other, for synchronizing a given window of elements. See figure 5.2 for an example in C++.

A client that wants to read a window should call the *access* method, and read elements starting from the returned pointer. If the client wants to write, the sequence is a little different; first if should call *access*, write elements, and finally, call *synchronize*. This method synchronizes, when needed, a portion of

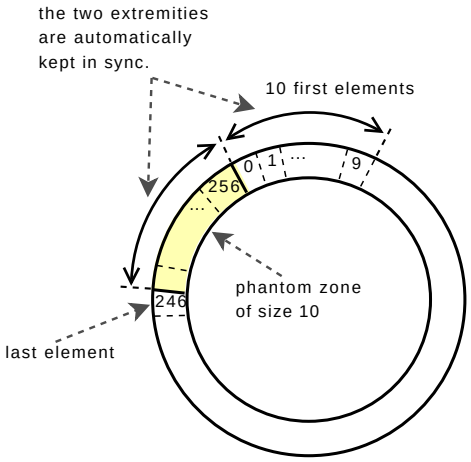


Figure 5.16: A phantom buffer of (logical) size 246, with 256 allocated elements and phantom zone of size 10.

```
template<class T> class PhantomBuffer
{
public:
    T* access(unsigned pos, unsigned size);
    void synchronize(unsigned pos, unsigned size);
    ...
};
```

Listing 5.2: PhanomBuffer class definition in C++

the phantom zone with its counterpart in the buffer beginning. To be accurate, a copy of elements will only be necessary when the window passed as argument to *synchronize* has intersection with the phantom or the initial zone.

Summing up, a phantom buffer offers a contiguous array where the last N elements are a replication of the first N. Each write on the first or last N element is automatically synchronized in its dual zone. Thus, the client of a phantom buffer will always have access to chunks of up to N contiguous elements,

Consequences

As a result of this design, clients must be well-behaved. This includes two aspects: the first is that clients that receive a pointer for a given window should not access elements beyond that window; the second is that, after a write, a client must call the *synchronize* method. Failing to do any of this might result in a serious run-time failure.

Certainly, this results in a lack of robustness. But this is the price to pay

for the requirement of providing plain pointers to the window, and avoiding unnecessary copies and reallocations. However, the phantom buffer interface should not be directly exposed to the concrete module implementation. The port classes presents a higher level interface to the module while hiding details such as window parameters and synchronizations.

The circular buffer allocation should be done at configuration time and the phantom size depends on the maximum window size (in that it must be greater).

Related Patterns

This pattern can be regarded as a part of a more extensive pattern that provides a generalized circular buffer with many readers. In this context, the windows management issues are addressed in the more general Multiple Window Circular Buffer pattern. Phantom Buffer provides a refinement of the lower-level layer drawn in the general pattern. Therefore, these two patterns collaborate together to give a complete solution for a generalized circular buffer.

Examples

This pattern can be found implemented in the CLAM framework. Specifically in the *PhantomBuffer* class.

5.4

Network Usability Patterns

5.4.1 Pattern: Recursive networks

Context

The potential of the interconnected modules model is virtually infinite. You can connect more and more modules to get larger and more complex systems. But module networks are normally defined by humans and humans have limitations on the complexity they can handle. So, big networks with a lot of connections are difficult to handle by the user, and this fact limits the potential of the model.

One of the reasons why audio systems become larger is duplication. Duplication happens, for example, whenever two audio channels have to be processed the same way, This duplication is very hard to maintain, because it implies having to apply repeated changes, and this is a very tedious and error prone process.

Duplication may happen also outside the system boundaries. The same set of interconnected modules may be present on several systems. Fixes on one of those systems do not apply to the other one so we have to apply it repeatedly and this is even more tedious and error prone.

Problem

How to reduce the complexity the user has to handle in order to define large and complex networks of interconnected modules?

Forces

- User defining big networks maybe too complex
- Human complexity handling is limited on the number of elements and relations

- Divide and conquer techniques help humans to handle complexity by focussing on smaller problems instead of the whole problem
- Duplications of sets of modules and connections is hard to maintain
- Reuse of previously designed networks helps on productivity
- Encapsulation hides details that can be useful on tracing the behavior of the system

Solution

By applying the 'divide & conquer' idea, we allow the user to define an abstraction of a set of interconnected modules as a single module that can be used in any other network. Some of the stream and event ports of the internal modules may be externalized as the stream and event ports of the container module.

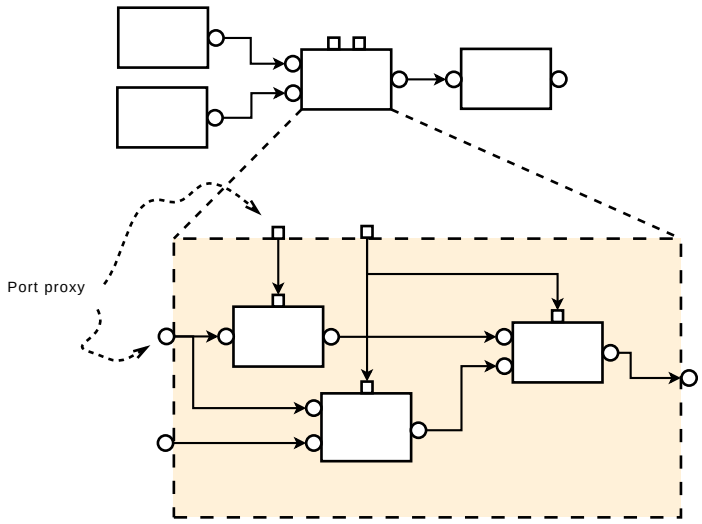


Figure 5.17: A network acting as a module.

Several internal *stream in-ports* may be merged as a single external one. So that, incoming stream tokens are read by all the internal stream in-ports. The same happens with in and out *event ports*. But it doesn't happen with the *stream out-ports*. The same reasons that forbid to *stream out-ports* feed a single *stream in-port* apply here.

If the system forbids merging ports on externalization, the externalized ports may be the internal ones. But when port merging is permitted, the user needs

an abstraction on connecting a single port. This abstraction is given by a Proxy [Gamma et al., 1995] port.

Depending on the implementation, the *proxy port* may act as a proxy on connection time or additionally on process time.

A *connect time port proxy* is a proxy port that delegates binding calls to the proxied ports. This way, during processing time the communication is done directly at non-proxy port level.

A *processing time port proxy* is a proxy port that acts as a the complementary in/out port for the internal ports. For example, an in proxy port is seen as out port for the internal ports connected to it. This is similar to have an identity module that just pipes tokens. The *processing time port proxy* adds overhead but it is useful when we need a clear boundary between inwards and outwards.

Also several approaches can be used for the flow control to handle *recursive networks*. One approach is to make the inner modules visible to the outer flow control, so that once all the modules are accessible by the flow control, all happens the same way it would happen if the recursive network was not there.

A second approach is to hide the inner modules to the flow control. This can be done by providing an inner flow control to the subnetwork. The subnetwork execution as module triggers the inner flow control. This approach is useful when a special flow control is needed, and also when we want to keep control on the proxied modules while processing.

Related Patterns

This pattern makes direct use of the Composite and Proxy [Gamma et al., 1995].

The flow control approach that hides inner modules to the outer flow control by providing a inner one, is a Hierarchical Control [Douglass, 2003].

Adjacent performance critical modules can be replaced by an optimized version as an static composition, trading flexibility by performance, using Adaptive Pipeline [Posnak and M., 1996].

Examples

Most audio domain frameworks implement Recursive Networks. For example MAX/MSP [Puckette, 1991], CSL [Pope and Ramakrishnan, 2003], OSW [Chaudhary et al., 1999], Aura [Dannenberg and Brandt, 1996b], Marsyas [Tzanetakis and Cook, 2002] and CLAM.

CLAM provides examples of most of the variants explained before. CLAM *Processing Composites* are compiled networks that provide their own flow control and they are seen for the flow control as a single module. *Processing Composites*'s ports are connection proxies so, external modules are actually connected on processing time to the inner ports. On the other side, CLAM also provides dynamic assembled networks, In this case, dummy modules which pipes directly event and stream tokens, are used as process time port proxies.

5.4.2 Pattern: Port Monitor

Context

Some audio applications need to show a graphical representation of tokens that are being produced by some module out-port. While the visualization needs just to be fluid, the processing has real-time requirements. This normally requires splitting visualization and processing into different threads, where the processing thread has real-time requirements and is a high priority scheduled thread. But because the non real-time monitoring should access to the processing thread tokens some concurrency handling is needed and this often implies locking.

Problem

We need to graphically monitor tokens being processed. How to do it without locking the real-time processing while keeping the visualization fluid?

Forces

- The processing has real-time requirements (ie. audio)
- Visualizations must be fluid; that means that it should visualize on time and often but it may skip tokens
- Just the processing is not filling all the computation time

Solution

The solution is to encapsulate concurrency in a special kind of process module, the *Port monitor*, that is connected to the monitored out-port. *Port monitors* offers the visualization thread an special interface to access tokens in a thread safe way.

In order to manage the concurrency avoiding the processing to stall, the *Port monitor* uses two alternated buffers to copy tokens. In a given time, one of them is the writing one and the other is the reading one. The *Port monitor* state includes a flag that indicates which buffer is the writing one. The *Port monitor* execution starts by switching the writing buffer and copying the current token there. Any access from the visualization thread locks the buffer switching flag. Port execution uses a *try lock* to switch the buffer, so, the process thread is not being blocked, it is just writing on the same buffer while the visualization holds the lock.

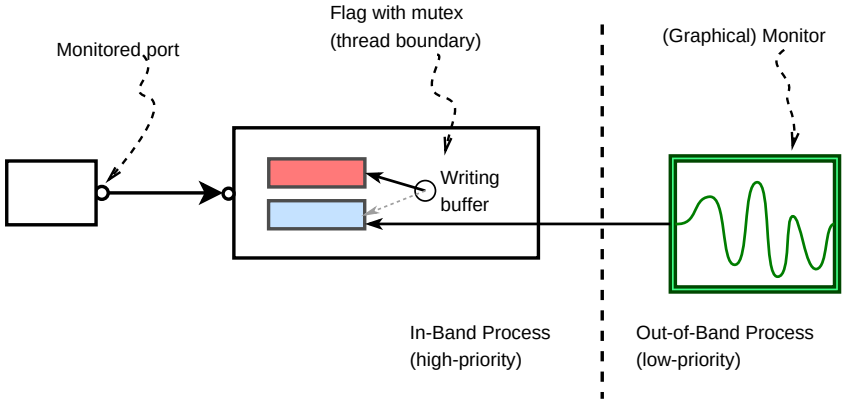


Figure 5.18: A port monitor with its switching two buffers

Consequences

Applying this pattern we minimize the blocking effect of concurrent access on two fronts. On one side, the processing thread never blocks. On the other, the blocking time of the visualization thread is very reduced, due that it only lasts a single flag switching.

Anyway, the visualization thread may suffer starvation risk. Not because the visualization thread will be blocked but because it may be always reading the same buffer. That may happen when every time the processing thread tries to switch the buffers, the visualization is blocking. This effect is not critical and can be avoided by minimizing the time the visualization thread is accessing tokens, for example, by copying them and release.

When this effect is too notorious, a solution is to use three buffers. This way, even when the visualization is blocking the buffer, the processing thread may alternate on the other buffers. The constraints that should apply are:

- Two buffer marks are always kept the *reading* buffer and the *last written* buffer.
- Three mutually exclusive operations may happen:
  - The processing thread should choose to write on any buffer that has none of those marks.
  - When the processing thread ends writing it updates the *last written* buffer.
  - When the visualization thread access, it moves the *reading* mark to the current *last written* mark.

This solution is not as good for real-time requirements as the one based on just two buffers. The former may block the processing thread, while the latter never blocks it.

Another issue with this pattern is how to monitor not a single token but a window of tokens. For example, if we want to visualize a sonogram (a color map representing spectra along the time) where each token is a single spectrum. The simplest solution, without any modification on the previous monitor is to do the buffering on the visualizer and pick samples at monitoring time. This implies that some tokens will be skipped on the visualization, but, for some uses, this is a valid solution.

The number of skipped tokens is not fixed, thus, this solution may show time stretching like artifacts that may not be acceptable for some application. Double/triple buffering on the port monitor the full window of tokens solves that. It is reliable but it affects the performance of the processing thread.

Related Patterns

Port Monitor is a refinement of Out-of-band and In-band Partition pattern (see a summary of this pattern in section3.4). Data flowing out of a port belongs to the In-band partition, while the monitoring entity (for example a graphical widget) is located in the out-of-band partition.

It is very similar to the Ordered Locking real-time pattern [Douglass, 2003]. Ordered Locking ensures that deadlock cannot occur, preventing circular waiting. The main difference is in their purpose: *Port Monitor* allows communicate two band partitions with different requirements.

Examples

The CLAM Network Editor [Amatriain and Arumí, 2005] is a visual builder for CLAM that uses Port Monitor to visualize stream data in patch boxes. The same approach is used for the companion utility, the Prototyper, which dynamically binds defined networks with a QT designer interface.

The Music Annotator also uses the concurrency handling aspect of Port Monitor although it is not based on modules and ports but in sliding window storage.

An example of Port Monitor outside CLAM can be found in Streamcatcher, an application for visualizing audio streams arranged by similarity, developed at the Austrian Research Institute for Artificial Intelligence [Gasser and Widmer, 2008]. Streamcatcher uses Port Monitor to visualize waveforms and similarity data on the GUI thread while keeping the processing thread lock-free and real-time safe. See section 6.3 for a description of Streamcatcher.

5.5

Visual Prototyping Patterns

5.5.1 Pattern: Visual Prototyper

Context

Many multimedia dataflow-based framework implements the *black-box* and *visual builder* patterns (see [Roberts and Johnson, 1996]). The behavior of a *black-box* framework is entirely determined by how processing objects are interconnected. In such framework, the *visual builder* enables non programmers to visually design dataflow compositions, and run them within the same visual builder. However, in the multimedia processing domain, sometimes it is necessary to build a standalone

application or plugin with an interactive GUI on top of dataflow processing core. Such standalone application (or plugin) cannot be the visual builder itself because it shows many details (such as the dataflow network) that are not useful (and potentially dangerous) to the final user.

Problem

The code for building a GUI with visual components that connects and interact with an underlying dataflow system is complex but similar from application to application, with only specific objects and parameters being different. What architecture and methodology enables to avoid the creation of such code?

Forces

- Domain experts are rarely programmers
- Building tools is expensive
- The composition of processing objects with visual components is convoluted and difficult to understand and generate with code.
- The target applications have a limited application logic (configure, start, stop), and send asynchronous events to the underlying dataflow while it is running.
- The GUI of the target application should not expose more details than the ones considered useful by the domain expert.
- The prototype should be embeddable in a wider application with a minimal effort
- The final application should allow communication of any kind of data and control objects between GUI and processing core (not just audio buffers)
- Plugin extensibility should be allowed for processing units, for graphical elements which provide data visualization and control sending, and for system connectivity backends (for example, in the audio domain: JACK, ALSA, PORTAUDIO, LADSPA, VST and AudioUnit)

Solution

Use a GUI framework with a visual builder (such as *Qt* with *Qt Designer*). Develop visual components plugins that extend such visual builder. Use the GUI visual

builder to compose the target prototype GUI. Also use the GUI visual builder to add meta-data that specify the links between the visual components and the dataflow processing objects.

Develop a run-time engine (maybe, but not necessarily, a program) that dynamically instantiate definitions coming from both tools —the dataflow visual builder, and the GUI visual builder— and relates them by inspecting the visual components meta-data introduced to this purpose. This run-time engine should also manage the application logic, such as configuring, starting and stopping the underlying dataflow.

We now give details of such architecture. Though the architecture is generic, for clarity sake we often refer to the implementation done in the CLAM framework using the Qt GUI framework

Target Applications The set of applications the architecture is able to visually build includes real-time audio processing applications, which have a relatively simple application logic. That is synthesizers, real-time music analyzers (figure 5.19) and audio effects and plugins (figure 5.20).

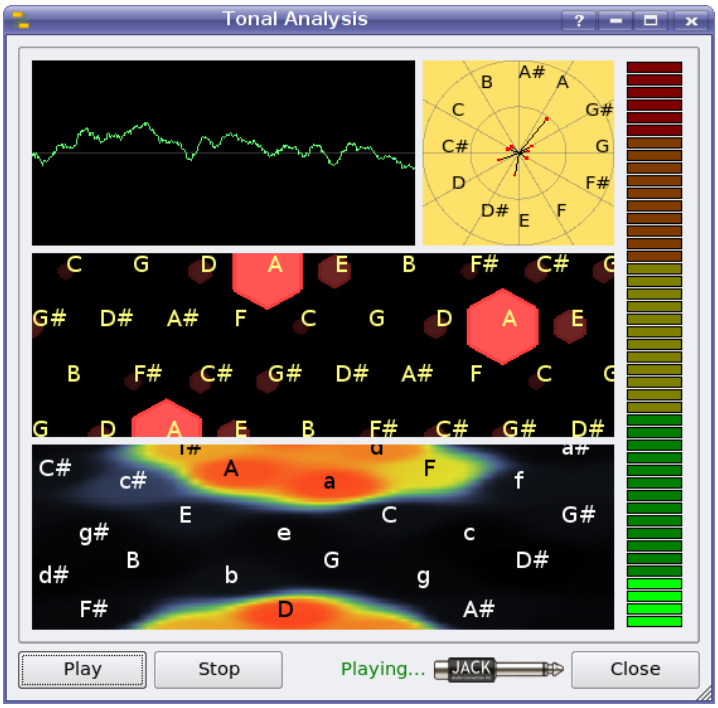


Figure 5.19: An example of an audio analysis application: Tonal analysis with chord extraction

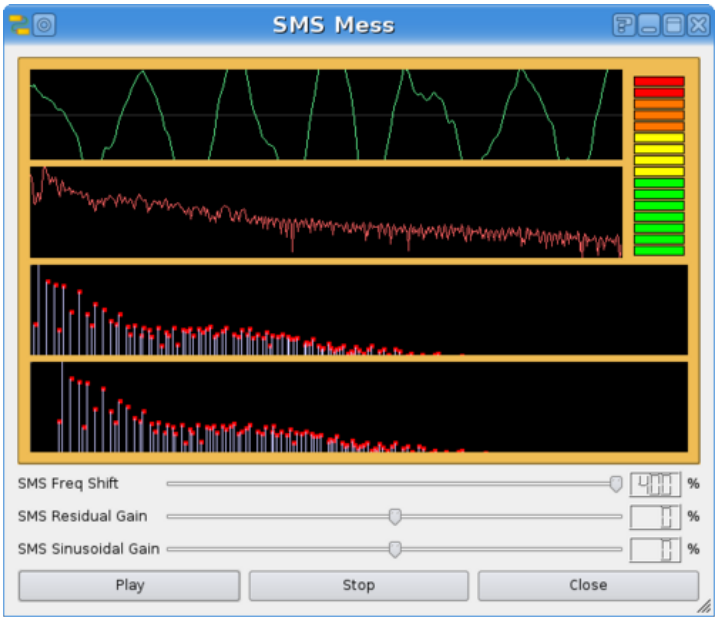


Figure 5.20: An example of a rapid-prototyped audio effect application: Pitch transposition. This application, which can be prototyped in CLAM in a matter of minutes, performs an spectral analysis, transforms the audio in the spectral domain, and synthesizes back the result. Note how, apart from representing different signal components, three sliders control the process interacting directly with the underlying processing engine.

The only limitation imposed on the target applications is that their logic should be limited to just starting and stopping the processing algorithm, configuring it, connecting it to the system streams (audio from devices, audio servers, plugin hosts, MIDI, files, OSC...), visualizing the inner data and controlling some algorithm parameters while running. Note that these limitations are very much related to the explicit life-cycle of a 4MPS Processing object outlined in section 3.2.

Given those limitations, the defined architecture does not claim to visually build every kind of audio application. For example, audio authoring tools, which have a more complex application logic, would be out of the scope, although the architecture would help to build important parts of such applications.

Main Architecture

The proposed architecture (figure 5.21) has three main components: A visual tool to define the audio processing core, a visual tool to define the user interface and a third element, the run-time engine, that dynamically builds definitions coming

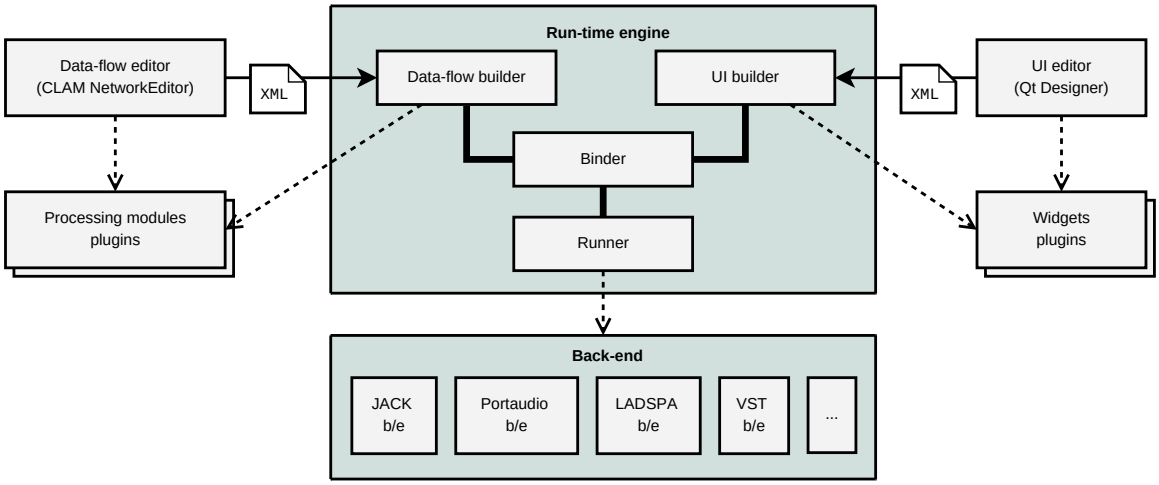


Figure 5.21: Visual prototyping architecture. The CLAM components that enable the user to visually build applications.

from both tools, relates them and manages the application logic. We implemented this architecture using some existing tools. We are using CLAM NetworkEditor as the audio processing visual builder, and Trolltech's Qt Designer as the user interface definition tool. Both Qt Designer and CLAM NetworkEditor provide similar capabilities in each domain, user interface and audio processing, which are later exploited by the run-time engine.

Qt Designer can be used to define user interfaces by combining several widgets. The set of widget is not limited; developers may define new ones that can be added to the visual tool as plugins. Figure 5.22 shows a Qt Designer session designing the interface for an audio application, which uses some CLAM data objects related widgets provided by CLAM as a Qt widgets plugin. Note that other CLAM data related widgets are available on the left panel list. For example to view spectral peaks, tonal descriptors or spectra.

Interface definitions are stored as XML files with the ".ui" extension. Ui files can be rendered as source code or directly loaded by the application at run-time. Applications may also discover the structure of a run-time instantiated user interface by using introspection capabilities.

Analogously, CLAM Network Editor allows to visually combine several processing modules into a processing network definition. The set of processing modules in the CLAM framework is also extensible with plugin libraries. Processing network definitions can be stored as XML files that can be loaded later by applications

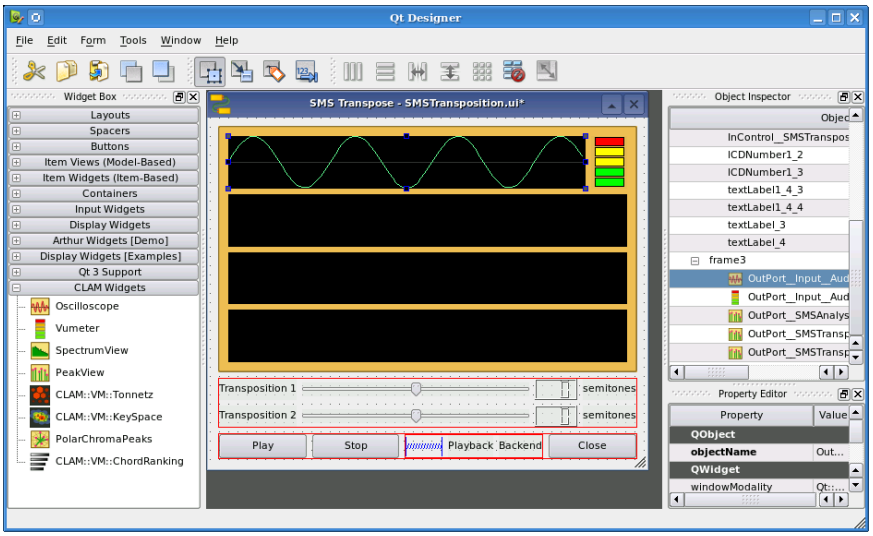


Figure 5.22: Qt Designer tool editing the interface of an audio application.

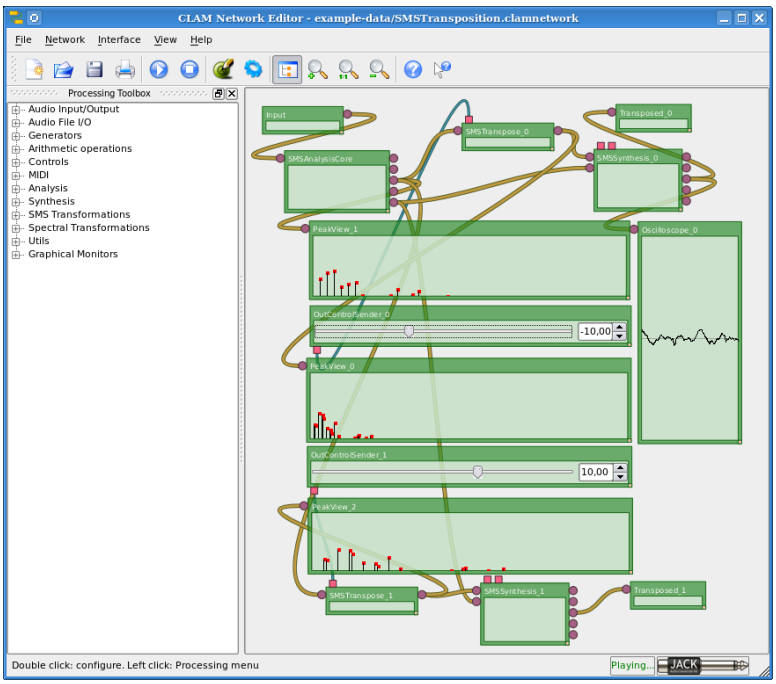


Figure 5.23: The processing core of an application built with the CLAM Network Editor

in run-time. And finally, the CLAM framework also provides introspection so a loader application may discover the structure of a run-time loaded network.

Run-time engine If only a dataflow visual builder and a visual interface designer was provided, some programming would still be required to glue it all together and launch the application. The purpose of the run-time engine, which is called Prototyper in our implementation, is to automatically provide this glue. Next, we enumerate the problems that the run-time engine faces and how it solves them.

Dynamic building

Both component structures, the audio processing network and the user interface, have to be built up dynamically in run-time from an XML definition. The complexity to be addressed is how to do such task when the elements of such structure are not known before hand because they are defined by add-on plugins [5]

Both CLAM and Qt frameworks provide object factories that can build objects given a type identifier. Because we want interface and processing components to be expandable, factories should be able to incorporate new objects defined by plugin libraries. To enable the creation of a certain type of object, the class provider must register a creator on the factory at plugin initialization.

In order to build up the components into an structure, both frameworks provide means for reflection so the builder can discover the properties and structure of unknown objects. For instance, in the case of processing elements, the builder can browse the ports, the controls, and the configuration parameters using a generic interface, and it can guess the type compatibility of a given pair of ports or controls.

Relating processing and user interface The run-time engine must relate components of both structures. For example, the spectrum view on the Transposition application (second panel on figure 5.20) needs to periodically access spectrum data flowing by a given port of the processing network. The run-time engine first has to identify which components, are connected. Then decide whether the connection is feasible. For example, spectrum data cannot be viewed by an spectral peaks view. And then, perform the connection, all that without the run-time

[5] Note that this is a recurring issue in audio applications where the use of plug-ins is common practice.

engine knowing anything about spectra and spectral peaks.

The proposed architecture uses properties such the component name to relate components on each side. Then components are located by using introspection capabilities on each side framework.

Once located, the run-time engine must assure that the components are compatible and connect them. The run-time engine is not aware of the types of data that connected objects will handle, we deal that by applying the Typed Connections design pattern introduced in section 5.2.4. In a nutshell, this design pattern allows to establish a type dependent connection construct between two components without the connector maker knowing the types and still be type safe. This is done by dynamically check the handled type on connection time, and once the type is checked both sides are connected using statically type checked mechanisms which will do optimal communication on run-time.

Thread safe communication in real-time One of the main issues that typically need extra effort while programming is multi-threading. In real-time audio applications based on a data flow graph, the processing core is executed in a high priority thread while the rest of the application is executed in a normal priority one following the Out-of-band and In-band partition pattern (see a summary of the pattern in section 3.4). Being in different threads, safe communication is needed, but traditional mechanisms for concurrent access are blocking and the processing thread cannot be blocked. Thus, new solutions, as the one proposed by the Port Monitor pattern in section 5.4.2, are needed.

A Port Monitor is a special kind of processing component which does double buffering of an input data and offers a thread safe data source interface for the visualization widgets. A flag tells which one is the read and the write buffer. The processing thread does a try lock to switch the writing buffer. The visualization thread will block the flag when accessing the data but as the processing thread just does a 'try lock', so it will just overwrite the same buffer but it won't block, fulfilling the real-time requirements of the processing thread.

System back-end Most of the application logic is coupled to sinks and sources for audio data and control events. Audio sources and sinks depend on the context of the application: JACK, ALSA, ASIO, DirectSound, VST, LADSPA... So the way of dealing with threading, callbacks, and assigning input and outputs is differ-

ent in each case. The architectural solution for that has been to provide back-end plugins to deal with this issues.

Back-end plugins address the often complex back-end setup, relate and feed sources and sinks in a network with real system sources and sinks, control processing thread and provide any required callback. Such plugins, hide all that complexity with a simple interface with operations such as setting up the back-end, binding a network, start and stop the processing, and release the back-end.

The back-end also transcends to the user interface as sometimes the application may let the user to choose the concrete audio sink or source, and even choose the audio back-end. Back-end plugin system also provides interface to cover such functionality.

5.6

Patterns as a Language

The 11 patterns presented in this catalog have different scope. Some are very high-level, like Semantic Ports and Driver Ports, while other are low-level, focused on implementation issues, like Phantom Buffer). Although the catalog is not domain-complete, it could be considered a *pattern language* because each pattern references higher-level patterns describing the context in which it can be applied, and lower-level patterns that could be used after the current one to further refine the solution. These relations form a hierarchical structure drawn in figure 5.24. The arcs between patterns mean "enables" relations: introducing a pattern in the system enables other patterns to be used.

This pattern catalog shows how to approach the development of a complete dataflow system for multimedia computing, in an evolutionary fashion without needing a *big up-front design*. The patterns at the top of the hierarchy suggest

to start with high level decisions driven by questions like: "do all ports drive the module execution or not?" and "does the system have to deal only with stream flow or also with event flow?" Then move on to address issues related to different token types such as: "do ports need to be strongly typed while connectible by the user?", or "do the stream ports need to consume and produce different block sizes?", and so on. On each decision, which will introduce more features and complexity, a recurrent problem is faced and addressed by one pattern in the language.

At some point, humans might need to interact with the system. Possible interaction includes building (complex) networks and monitoring the flowing data. This is what the *Network Usability Patterns* and *Visual Prototyper Pattern* do. They can be introduced in the first stages of the system evolution or later on.

5.6.1 Patterns Applicability

The presented patterns are mainly inspired by our design experience in the audio domain. But have an immediate applicability in the more general multimedia processing systems domain.

Typed Connections, Multiple Window Circular Buffer and Phantom Buffer have applicability beyond dataflow systems. And, regarding the Port Monitor pattern, though its description is coupled with the dataflow architecture, it can be extrapolated to other environments where a normal priority thread is monitoring changing data on a real-time one.

Most of the patterns in this catalog can be found in many audio systems. However, examples of a few others (namely Multi-rate Stream Ports, Multiple Window Circular Buffer and Phantom Buffer) are hard to find outside of CLAM so they should be provisionally considered innovative patterns (or proto-patterns).

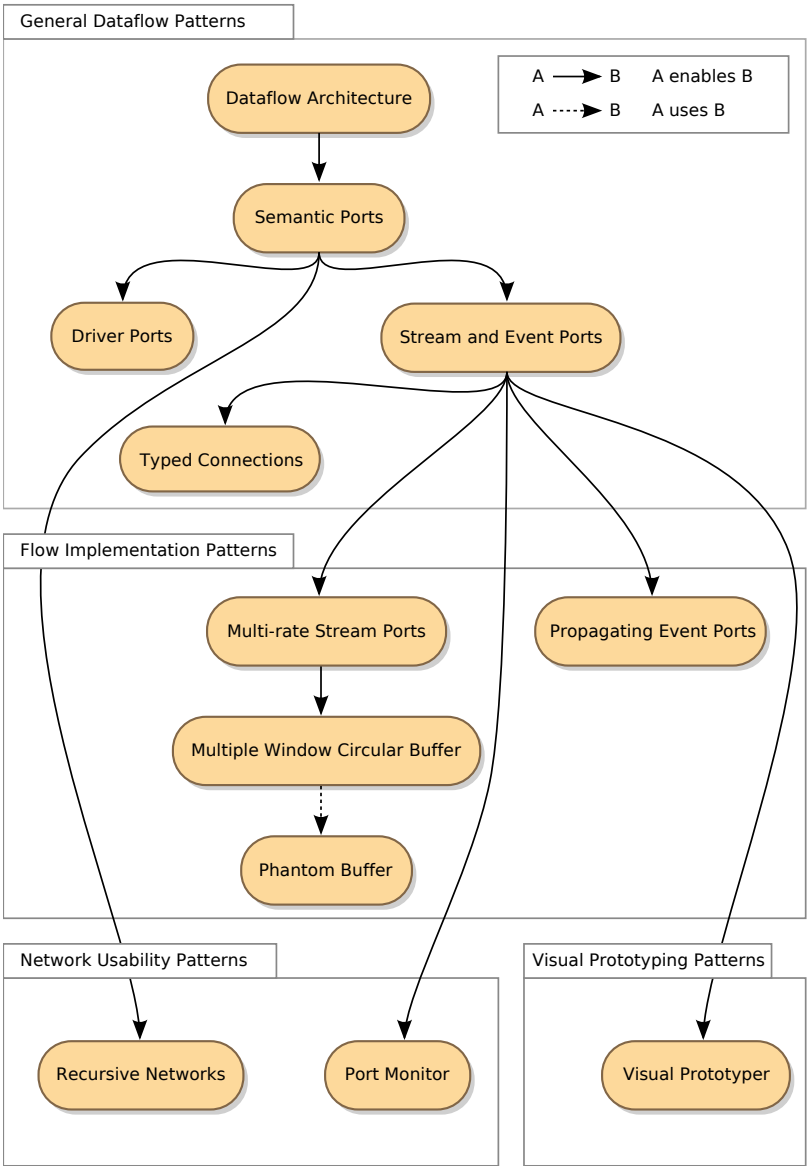


Figure 5.24: The multimedia dataflow pattern language. High-level patterns are on the top and the arrows represent the order in which design problems are being addressed by developers.

5.7

Summary

In this chapter we have proposed a pattern language for the multimedia domain. The pattern language is made up of 11 interrelated pattern addressing the following aspects of dataflow-based multimedia processing systems:

General Dataflow Patterns: Address problems about how to organize high-level aspects of the dataflow architecture, by having different types of modules connections. Belonging to this category:

- Semantic Ports addresses distinct management of tokens by semantic.
- Driver Ports addresses how to make modules executions independent of the availability of certain kind of tokens.
- Stream and Event Ports addresses how to synchronize different streams and events arriving to a module.
- Typed Connections addresses how to deal with typed tokens while allowing the network connection maker to ignore the concrete types.

Flow Implementation Patterns: Address how to physically transfer tokens from one module to another, according to the types of flow defined by the *general dataflow patterns*. Tokens life-cycle, ownership and memory management are recurrent issues in these patterns.

- Propagating Event Ports addresses the problem of having a high-priority event-driven flow able to propagate through the network.
- Multi-rate Stream Ports addresses how stream ports can consume and produce at different rates;
- Multiple Window Circular Buffer addresses how a writer and multiple readers can share the same tokens buffer.

- Phantom Buffer addresses how to design a data structure both with the benefits of a circular buffer and the guarantee of window contiguity.

Network Usability Patterns: Address how humans can interact with dataflow networks.

- Recursive Networks makes feasible for humans to deal with the definition of big complex networks;
- Port Monitor addresses how to monitor a flow from a different thread, without compromising the network processing efficiency.

Visual Prototyping Patterns: Address how domain experts can generate applications on top of a dataflow network, with interactive GUI, without needing programming skills.

- Visual Prototyper Addresses how to dynamically build a graphical user interface that interacts with the underlying dataflow model. This pattern enables rapid applications prototyping using (but not restricted to) visual tools.

The presented design patterns provide useful design reuse in the domain of multimedia processing systems. They cover the fundamental features of dataflow systems, and their solutions are general enough to be used in many different contexts.

All of them provide a *teaching component*, mostly found in the "forces" and "consequences" sections, which provides the fundamental insight that enables the pattern solution to be reused effectively.

Finally, this chapter shows how the patterns are interrelated forming a pattern language, enabling developers to follow a path of design decisions. The pattern language offers options with informed trade-offs on each development stage. The choice for the next pattern to implement, as well as the specific forces involved in the pattern, is driven by the requirements of the system under development.

5.7.1 Summary of Usage Examples

For each pattern we show, in the "examples" section, that it can be found in different well known applications and contexts. When possible we have presented three or more examples of varied systems. Some other patterns are "innovative", in that they address design problems we found during the course of the CLAM

framework development, but not found elsewhere. However, the CLAM framework has been instantiated in many applications with different purposes and on different fields.

This section collect all the examples found for each pattern, in order to give a general picture. The source for the examples are the following 11 systems: Pure-Data (PD) [Puckette, 1997], Max/MSP [Puckette, 1991], Open Sound World (OSW) [Chaudhary et al., 1999], JACK [Davis et al., 2004], SuperCollider3 [McCartney, 2002], CSL [Pope and Ramakrishnan, 2003], Marsyas [Tzanetakis and Cook, 2002], Aura [Dannenberg and Brandt, 1996a], CLAM (framework) [Amatriain and Arumí, 2005], CLAM Music Annotator [Amatriain et al., 2005], and Streamcatcher [Gasser and Widmer, 2008].

- General Dataflow Patterns:
  - Semantic Ports addresses distinct management of tokens by semantic.
    Found in: PD, Max/MSP, OSW, JACK, CLAM
  - Driver Ports addresses how to make modules executions independent of the availability of certain kind of tokens.
    Found in: PD, Max/MSP, OSW, JACK, CLAM
  - Stream and Event Ports addresses how to synchronize different streams and events arriving to a module.
    Found in: SuperCollider3, CSL, Marsyas, OSW, CLAM
  - Typed Connections addresses how to deal with typed tokens while allowing the network connection maker to ignore the concrete types.
    Found in: OSW, Music Annotator, CLAM
- Flow Implementation Patterns:
  - Propagating Event Ports addresses the problem of having a high-priority event-driven flow able to propagate through the network.
    Found in: PD, Max/MSP, CLAM
  - Multi-rate Stream Ports addresses how stream ports can consume and produce at different rates;
    Found in: Marsyas, SuperCollider3, CLAM

  - Multiple Window Circular Buffer addresses how a writer and multiple readers can share the same tokens buffer.
    Found in: CLAM
  - Phantom Buffer addresses how to design a data structure both with the benefits of a circular buffer and the guarantee of window contiguity.
    Found in: CLAM

- Network Usability Patterns:

  - Recursive Networks makes feasible for humans to deal with the definition of big complex networks.
    Found in: PD, Max/MSP, CSL, OSW, Aura, Marsyas, CLAM
  - Port Monitor addresses how to monitor a flow from a different thread, without compromising the network processing efficiency.
    Found in: CLAM, Music Annotator and Streamcatcher.

- Visual Prototyping Patterns:

  - Visual Prototyper addresses Addresses how to dynamically build a graphical user interface by relating a dataflow network with graphical monitors and actuators.
    Found in: CLAM

5.7.2 Patterns as Elements of Design Communication

Design patterns are useful to communicate, document and compare the designs of multimedia systems. As an example, consider the following sentence "The framework X has a Dataflow Architecture. Its module's ports are *Semantic Ports*. Uses Stream and Event Ports. Stream ports are Driver Ports and implement Typed Connections, with concrete types including Audio, Spectrum, Note or Melody, event ports are restricted to floats. Event ports are implemented with Propagating Event Ports while audio stream ports use Multi-rate Stream Ports implemented with Multiple Window Circular Buffer and a Phantom Buffer.

This paragraph concisely conveys a big amount of design information. It shows how these patterns enables an efficient communication and documentation of the design of a multimedia processing system. Of course, this requires that the receptor

is familiar with the cited patterns. This requirement can be alleviate by including links to a pattern catalog.

Christopher Alexander —the building architect who first introduced the patterns formalism— defined a pattern as both a "thing" —the design— and "instructions on how to produce the thing" [Alexander, 1977]. When documenting a system design we are using the first meaning. When designing and implementing a system we, of course, interested in the instructions part of a pattern.

These patterns are also efficient tools for comparing different systems. Again, an example: "Max/MSP uses Driver Ports but unlike CLAM, its event ports can also be Driver Ports.". Another example: "While CSL uses Stream and Event Ports, JACK (in its version 0.100) does not because its network only streams audio samples. Neither CSL nor JACK use Typed Connections"

Next chapter validates the two main contributions of the thesis —namely: the Time-Triggered Synchronous Dataflow model and the Multimedia Dataflow Pattern Language— by presenting case studies that uses them both.

CHAPTER 6

Case Studies

This chapter presents case studies that demonstrates the use of the Time-Triggered Synchronous Dataflow model and Multimedia Dataflow Pattern Language. We begin in section 6.1 describing CLAM, a framework we developed, that implements and uses both (the model, and patterns) abstractions. Some of the applications build with the framework are also reviewed.

Next, in section 6.2, we present a real-time 3D-audio system integrated in a digital cinema workflow. In this case study, we emphasize how the time-triggered model and dataflow patterns are used. This 3D-audio system uses an underlying dataflow exhibiting multi-rate, and therefore, have an interesting time-triggered scheduling. Moreover, the whole system involves not only audio, but video and 3D graphics.

6.1

CLAM: A Framework for Rapid Development of Cross-platform Audio Applications

The history of software frameworks is very much related to the evolution of the multimedia field itself. Many of the most successful and well-known examples of software frameworks deal with graphics, image or multimedia[1]. Although probably less known, the audio and music fields also have a long tradition of similar development tools. And it is in this context where we find CLAM, a framework that recently received the 2006 ACM Best Open Source Multimedia Software award.

CLAM stands for C++ Library for Audio and Music and it is a full-fledged software framework for research and application development in the audio and music domain with applicability also to the broader Multimedia domain. It offers a conceptual model; algorithms for analyzing, synthesizing and transforming audio signals; and tools for handling audio and music streams and creating cross-platform applications.

The CLAM framework is cross-platform. All the code is ANSI C++ and it is regularly compiled under GNU/Linux, Windows and Mac OSX using FLOSS (Free Libre Open Source Software) tools, such as automatic integrated building/testing/versioning systems, and agile practices such Test-Driven Development. (see [Amatriain, 2007b] and [Amatriain et al., 2002b], for instance)

We will explain the different CLAM components and the applications included in the framework in section 6.1.1. In section 6.1.2 we will also explain the rapid prototyping features that have been added recently and already constitute one of its major assets.

[1]The first object-oriented frameworks to be considered as such are the MVC (Model View Controller) for Smalltalk [Burbeck, 1987] and the MacApp for Apple applications[Wilson, 1990]. Other important frameworks from this initial phase were ET++[Weinand et al., 1989] and Interviews. Most of these seminal frameworks were related to graphics or user interfaces

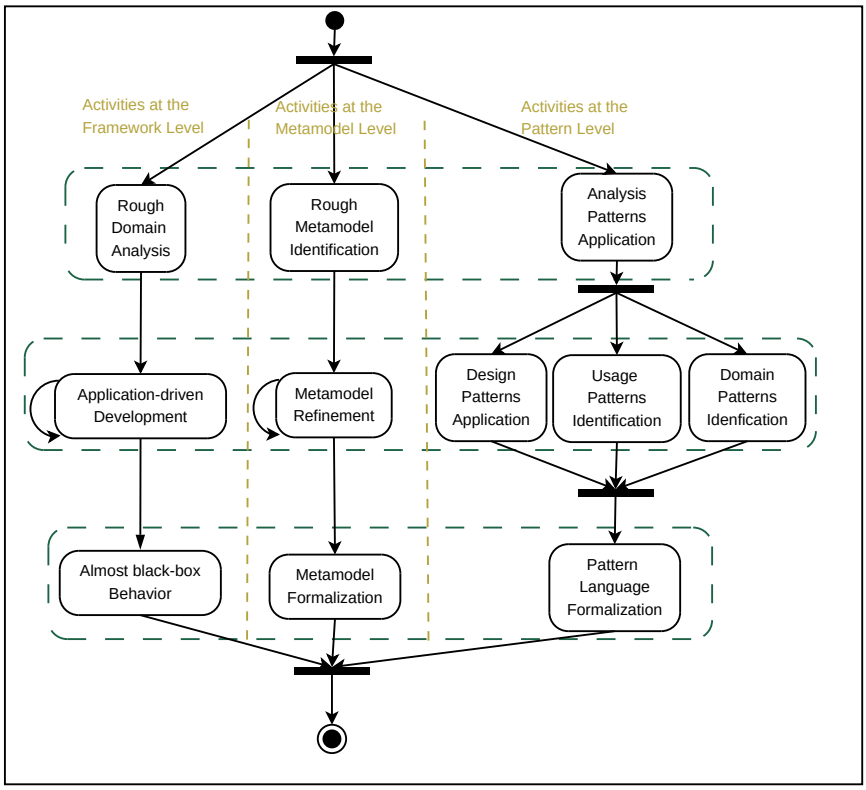


Figure 6.1: CLAM development process and related activities

In this sense, the framework is not only valuable for its features but also for other outputs of the process that can be considered as reusable components and approaches for the multimedia field. The process of designing CLAM generated reusable concepts and ideas that are formalized in the form of a general purpose domain-specific meta-model and a pattern language both of which are outlined in the next section.

During the CLAM development process several parallel activities have taken place (see figure 6.1). While some sought the goal of having a more usable framework, others dealt with also coming up with the appropriate abstractions and reusable constructs.

6.1.1 CLAM Components

As seen in figure 6.2 CLAM offers a processing kernel that includes an *infrastructure* and processing and data *repositories*. In that sense, CLAM is both a *black-box* and a *white-box* framework [Roberts and Johnson, 1996]. It is black-box

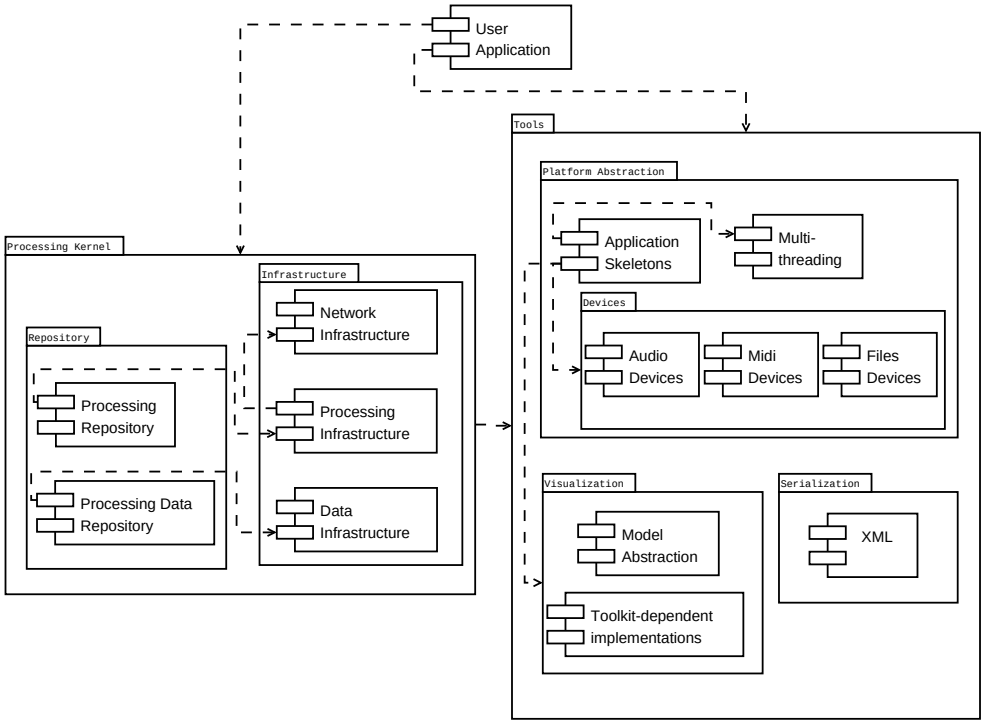


Figure 6.2: CLAM components. The CLAM framework is made up of a Processing Kernel and some Tools. The Processing Kernel includes an Infrastructure that is responsible for the framework white-box behavior and repositories that offer the black boxes. Tools are usually wrappers around pre-existing third party libraries. A user application can make use of any or all of these components.

because already built-in components included in the repositories can be connected with minimum or no programmer effort in order to build new applications. And it is *white-box* because the abstract classes that make up the infrastructure can be easily derived to extend the framework components with new processes or data classes.

Apart from the kernel, CLAM includes a number of tools for services such as audio input/output or XML serialization and a number of applications that have served as a testbed and validation of the framework.

In the next paragraphs we will review the CLAM infrastructure, repositories, and its tools.

The Infrastructure

The CLAM infrastructure is a direct implementation of the 4MPS meta-model, which has already been reviewed in section 3.2. And the multimedia dataflow patterns described in chapter 5.

Indeed, the meta-classes illustrated in figure 3.3 are directly mapped to C++ abstract classes in the framework (note that C++ does not accept meta-classes naturally). These meta-classes are responsible for the white-box or extensible behavior in the framework. When a user wants to add a new Processing or Data to the Repository a new concrete class needs to be derived from these classes.

The Repositories

The *Processing Repository* contains a large set of ready-to-use processing algorithms, and the *Data Repository* contains all the classes that act as data containers to be input or output to the processing algorithms.

The Processing Repository includes around 150 different Processing classes, classified in categories such as Analysis, ArithmeticOperators, or AudioFileIO.

Although the repository has a strong bias toward spectral-domain processing because of our research group's background and interests, there are enough encapsulated algorithms and tools so as to cover a broad range of possible applications.

On the other hand, in the Data Repository we offer the encapsulated versions of the most commonly used data types such as Audio, Spectrum, or Segment. It is interesting to note that all of these classes make use of the data infrastructure and are therefore able to offer services such as a homogeneous interface or built-in automatic XML persistence.

Tools

Apart from the infrastructure and the repositories, which together make up the CLAM *processing kernel* CLAM also includes a large number of tools that can be necessary to build an audio application.

All of this tools are possible thanks to the integration of third party open libraries into the framework. Services offered by these libraries are wrapped and integrated into the meta-model so they can be used as natural constructs (mostly Processing objects) from within CLAM. In this sense, one of the benefits of using CLAM is that it acts as a common point for already existing heterogeneous services [Amatriain, 2007b].

XML XML is used throughout CLAM as a general purpose storage format in order to store objects that contain data, descriptors or configurations [Garcia and Amatrian, 2001]. In CLAM a Spectrum as well as a Network configuration can be transparently stored in XML. This provides for seamless interoperability between applications allowing easy built-in data exchange.

GUI Just as many frameworks, CLAM had to think about ways of integrating the core of the framework tools with a graphical user interface that may be used as a front-end to the framework functionalities. In CLAM this is accomplished through the Visualization Module, which includes many already implemented widgets offered for the Qt framework. The more prominent example of such utilities are the *port monitors*: Widgets that can be connected to ports of a CLAM network to show its flowing data. A similar tool called Plots is also available for debugging data while implementing algorithms.

Platform Abstraction Under this category we include all those CLAM tools that encapsulate system-level functionalities and allow a CLAM user to access them transparently from the operating system or platform.

Using these tools a number of services, such as audio input/output, audio file formats, MIDI input/output, or SDIF file support, can be added to an application and then used on different operating systems with exactly the same code and always in observing the 4MPS meta-model.

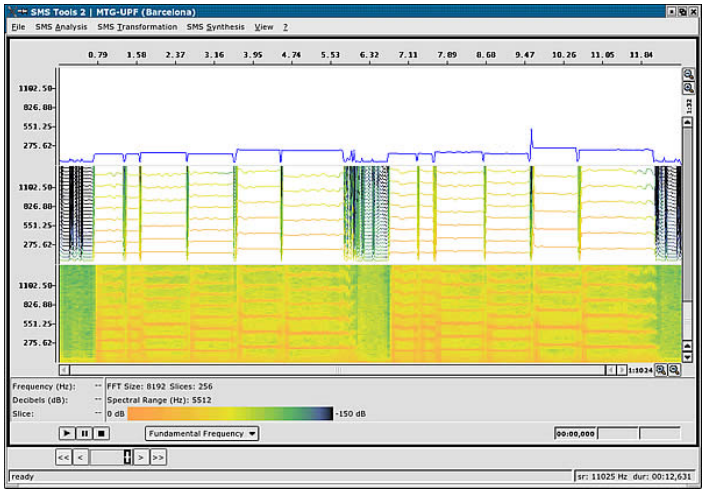


Figure 6.3: The SpectralTools graphical user interface. This application can be used not only to inspect and analyze audio files but also to transform them in the spectral domain.

CLAM Applications

The framework has been tested on —but also its development has been driven by— a number of applications. Many of these applications were used in the beginning to set the domain requirements and they now illustrate the feasibility of the meta-model, the design patterns and the benefits of the framework. In the following paragraphs we will present some of these applications.

Spectral Modeling Analysis/Synthesis One of the main goals when starting CLAM was to develop a replacement for a similar pre-existing tool.

This application (see GUI in figure 6.3) is used to analyze, transform and synthesize back a given sound. For doing so, it uses the Sinusoidal plus Residual model [Amatriain et al., 2002a]. The application reads an XML configuration file, and an audio file (or a previously analyzed SDIF file). The input sound is analyzed, transformed in the spectral domain according to a transformation score and then synthesized back.

The Annotator The CLAM Annotator [Amatriain et al., 2005] is a tool for inspecting and editing audio descriptors (see figure 6.4). The application can be used as a platform for launching custom extraction algorithms that analyze the signal and produce different kinds of descriptors. It provides tools for merging and

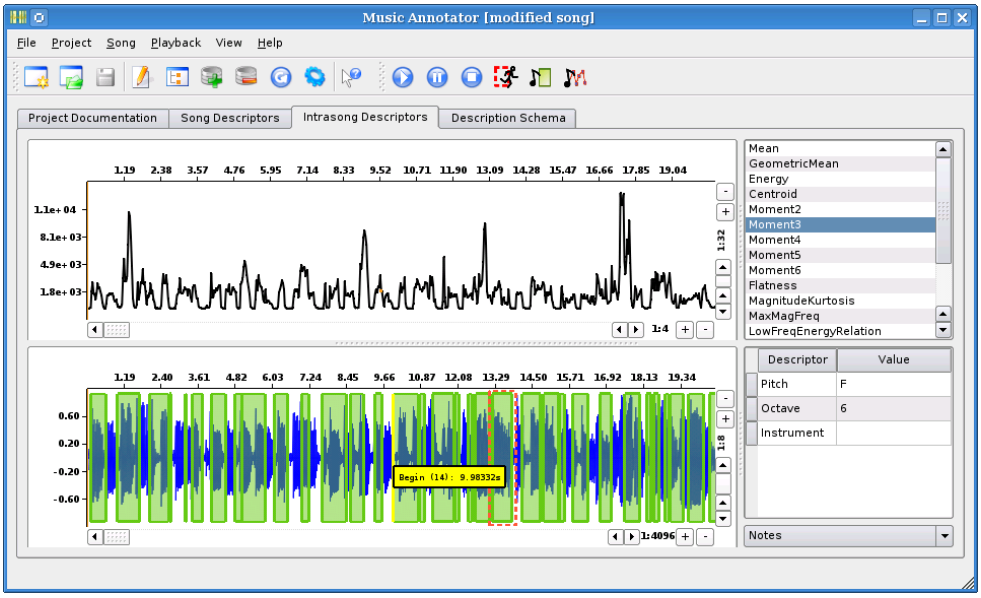


Figure 6.4: Editing low-level descriptors and segments with the CLAM Annotator. This tool provides ready-to-use descriptors such as chord extraction and can also be used to launch custom algorithms

filtering different source of descriptors that can be custom extractor programs or even remote sources from web services.

Descriptors can be organized at different levels of abstraction: song level, frame level, but also several segmentations with different semantics and attributes. The descriptors can be synchronously displayed or auralized to check their correctness. Merging different extractors and hand edited ground truth has been proved very useful to evaluate extractors' performance.

Others Many other sample usages of CLAM exist apart from the main applications included in the repository and described above.

For instance, *SALTO* is a software based synthesizer [Haas, 2001] that implements a general synthesis architecture configured to produce high quality sax and trumpet sounds. *SpectralDelay*, also known as CLAM's Dummy Test, was the first application implemented in the framework. It was chosen to drive the design in its first stages. The application implements a delay in the spectral domain: the input audio signal can be divided with CLAM into three bands and each of these bands can be delayed separately.

The repository also includes many smaller examples that illustrate how the

framework can be used to do particular tasks ranging from a simple sound file playback to a complex MPEG7 descriptor analysis.

6.1.2 CLAM as a Visual Prototyping Environment

So far we have seen that CLAM can be used as a regular application framework by accessing the source code. Furthermore, ready-to-use applications such as the ones presented in the previous section provide off-the-self functionality.

But latest developments have brought *visual building* capabilities into the framework. These allow the user to concentrate on the research algorithms and not on application development. Visual building is also valuable for rapid prototyping of applications and plug-ins.

CLAM's visual builder is known as the NetworkEditor (see figure 5.23). It allows to generate an application —or only its processing engine— by graphically connecting objects in a patch. Another application called Prototyper acts as the glue between a graphical GUI designing tool (such as Qt Designer) and the processing engine defined with the NetworkEditor.

Having a proper development environment is something that may increase development productivity. Development frameworks offer system models that enable system development dealing with concepts of the target domain. Eventually, they provide visual building tools that also improve productivity [Green and Petre, 1996].

6.2

Real-Time Room Acoustics Simulation in 3D-Audio

6.2.1 Introduction

In this section we present a real-time 3D-audio system integrated in a digital cinema workflow. It is an interesting case study because its dataflow processing exhibits multi-rate, and whole system involves audio, video and 3D graphics. All the real-time processing runs on CLAM networks using an implementation of the TTSDF model presented in section 4.2. Therefore, this system uses and exemplifies both the TTSDF model an Dataflow Patterns presented in this thesis.

This 3D-audio system was developed within the Audio an Music group of Barcelona Media technology center. It was demonstrated in the context of the IP-RACINE project [2].

The role of our specific system, within the general workflow, is to generate immersing 3D-audio using room acoustics simulation techniques. It allows the sound sources and listener move in the virtual environment. The listener movements is interactively driven by the tracking system of the shooting camera. The system renders 3D-audio in real-time and offers a plausible reference with the visual environment. A nice feature of this system is that it can process multiple moving sound sources and listeners in a normal CPU.

In our first approach, a database of impulse-responses (IR's) with directionality information was computed offline (that is, before the real-time processing) for

[2]IP-RACINE European Union Integrated Project aims to secure the future of the European Cinema industry in the change from film to digital, improving the competitiveness of European Digital Cinema (DC) by developing workflow techniques for integrating the digital process "from scene to screen", and advancing the state of the art of digital cameras, virtual cinema studio production, cinema objects description, processing and post-production, and digital playout and display of sound and image for better user experience. IP-Racine research project consortium supported by DG InfSo of the European Commission, 2006.

each 3D environment. While it works well for small 3D environments and fixed source positions, the database solution does not scale well because its size grows exponentially on the density of source/listeners points to be computed. Therefore, the system was updated to support computing the IR's on-the-fly. The "quality" of the IR can be configured, which enables trading-off acoustic quality for IR's density and number of sources.

When running, the real-time system retrieves IR's corresponding to the sources and target positions, performs low-latency convolution between the IR's and the incoming audio, and smoothes IR transitions with cross-fades. Finally, the system is flexible enough to decode to any surround exhibition setup.

6.2.2 The "Testbed" Integrated System

The IP-RACINE project provided the opportunity for deploying our audio system and integrate it with other parts of the digital cinema workflow —being run by other partners of the consortium. This deployment took place during a IP-RACINE's "testbed", in December 2007. The testbed goal was to demonstrate how new technologies can be applied and integrated for the production an augmented-reality short film, involving all the processes from shooting to exhibiting.

The elements of that particular testbed consisted in a chroma shooting set, with four actors —a flamenco group of three musicians and a dancer—, a high-definition video camera with position and zoom tracking, several video and monitoring systems, and several other systems related to HD video processing, storing and exhibition.

The flamenco group was filmed while performing in playback over the music, which was pre-recorded in separate tracks at a studio. The role of our real-time audio system was to process each individual dry-recorded audio track obtaining a convincing spatialized sound. The result effectively generated the illusion of moving around the musicians in a (virtual) architectonic space.

The filming was done with a high-end high-definition prototype Thompson camera with zoom and position tracking, which enabled the real-time motion of a subjective listener within the scene. Using the tracking data, both augmented-reality video and immersive 3D-audio was generated in real-time allowing the director to not only preview the final scene while shooting but also pre-hear the 5.1 surround audio and HRTF-based binaural (with speakers), thus enabling artistic

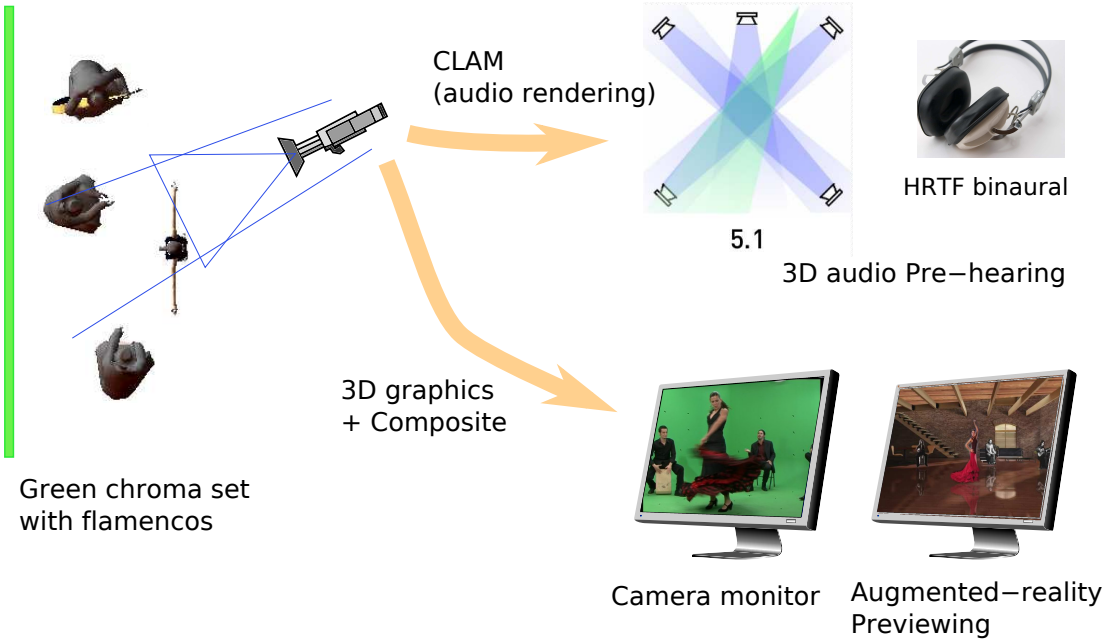


Figure 6.5: The IP-RACINE "testbed" setup: The shooting of an augmented-reality scene. The audio system is fed with the zoom and position tracking data of the camera, and renders 3D audio using room acoustics techniques based on ray-tracing and low-latency convolutions. The audio is exhibited both in 5.1 Surround and HRTF-based binaural format. The director not only previews the augmented-reality video but also pre-hears the 3D-audio, allowing her to take artistic decisions based on the final result.

decisions based on the final result. This set up is depicted in figure 6.5

The audio system also allows fine-tunning the audio rendering —for example, changing the size of the room, or the acoustics material— in a post-production environment. Additionally, it can run in offline mode enabling final renders with high quality IR's.

6.2.3 A 3D-Audio Dataflow Case Study

The processing core of the described system is done within the CLAM framework. It contains a good example of applicability of the TTSDF model and it's real-time capabilities. This application also shows how it takes advantage of the presented patterns.

Before diving into the details on how the TTSDF model and patterns are applied we will give some context on the 3D-audio domain and introduce the main CLAM networks used (which corresponds to the TTSDF graphs). However, this context is not key to understand how the model and patterns are applied. Therefore, readers not interested in the 3D-audio domain can safely skip the following paragraphs and continue in the (next) section 6.2.4.

The network depicted in figure 6.6 shows the processing core of the system. It produces 3D-audio from an input audio stream, plus the position of a source and a listener in a given 3D geometry —which can also be an empty geometry. If the room-simulation mode is enabled, the output audio contains reverberated sound with directional information for each sound reflection, therefore producing a sensation of being immersed in a virtual environment to the user.

The format of the output is Ambisonics of a given specified order [Gerzon, 1973, Malham and Myatt, 1995]. From the Ambisonics format, it is possible to decode the audio to fit diverse exhibition setups such as HRTF-based audio through headphones, standard surround 5.1 or 7.1, or other non-standard loudspeakers setups. Figure 6.8 shows a CLAM network that decodes first order Ambisonics (B-Format) to surround 5.0, whereas the network in figure 6.9 decodes B-Format to binaural.

Let us describe in more detail the main audio rendering network, depicted in figure 6.6. The audio scene to be rendered is animated by a processing which produces controls of source/listener positions and angles. This processing can be either an OSC[3] receiver or a file-based sequencer. The picture illustrates this

[3]Open Sound Control is a protocol for communication among computers, sound synthesizer and other multimedia devices, optimized for networking technologies [Wright, 1998]. It can be

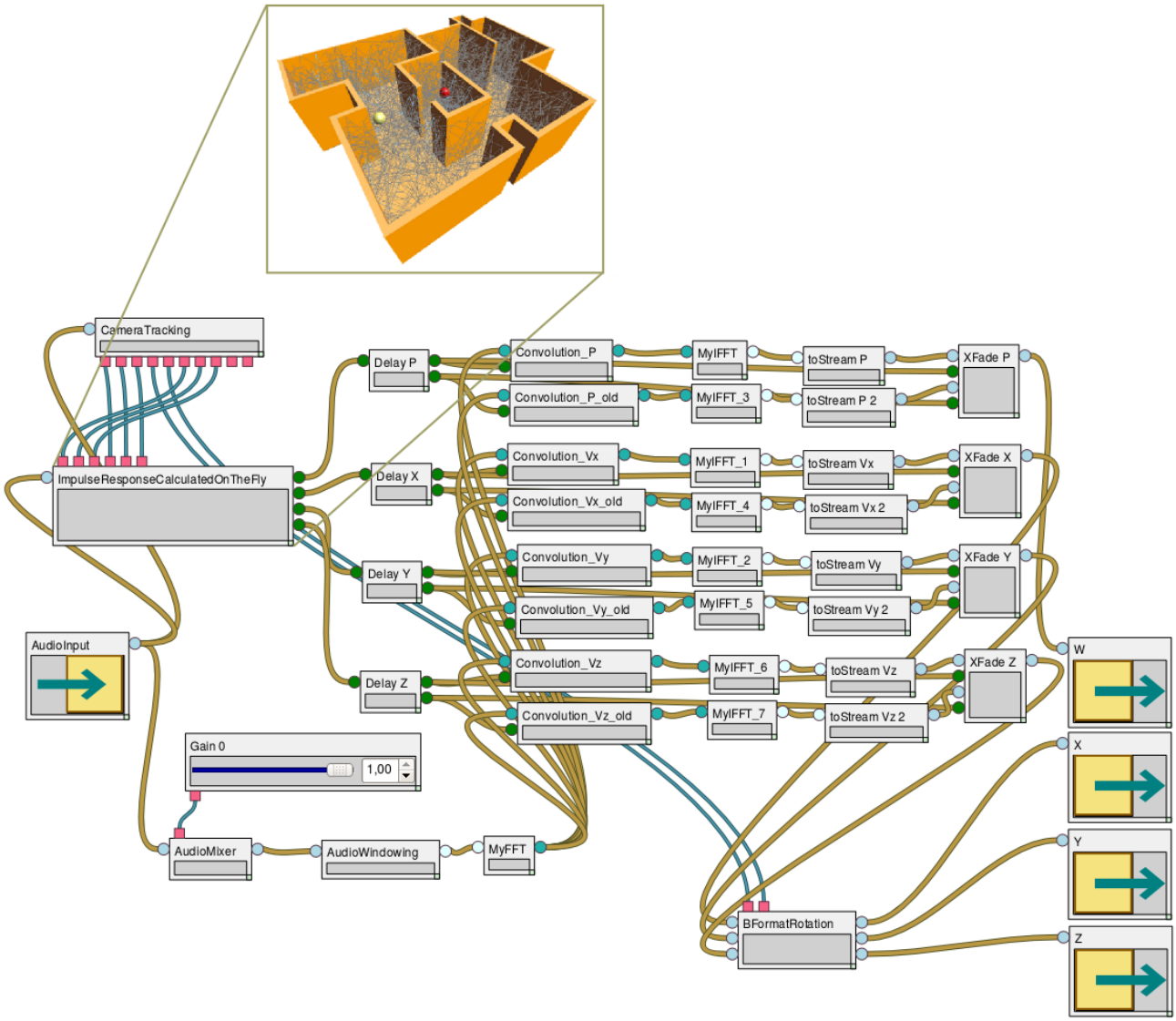


Figure 6.6: CLAM network that renders the audio in B-Format

second case, where the *CameraTracking* processing sequences controls from a file (for instance, exported from Blender) and uses an audio input for synchronization purposes.

The actual audio rendering process is done in two stages. The first stage consists on the computation of the acoustic impulse-response (IR) in Ambisonics format for a virtual room at the given source/listener positions. This process takes place in the *ImpulseResponseCalculatedOnTheFly* processing which outputs the IRs. Since IRs are typically larger than an audio frame they are encoded as a list of FFT frames.

The second stage consists on convolving the computed IRs, using the overlap-and-add convolution algorithm, which is depicted in figure 6.7 and explained in [Torger and Farina, 2001]. This process is implemented in the *Convolution* processing which takes two inputs: a FFT frame of the incoming audio stream and the aforementioned IR.

The IRs calculation uses acoustic ray-tracing algorithms , which take into account the characteristics of the materials, such as impedance and diffusion. The IR calculation is only triggered by the movement of the source or listener, with a configurable resolution.

First informal real-time tests have been carried out successfully using simplified scenarios: few sources (3), simple geometries (cube), and few rays (200) and rebounds (70). We are still in the process of achieving a physically consistent reverberation by establishing the convergence of our ray-tracer (i.e. making sure that we compute enough rays and enough rebounds to converge to a fixed RT60). Another future line is optimize the algorithm for real-time by reusing or modeling reverberation tails and only compute the early reflections by ray-tracing.

As the diagram shows, each B-Format component (W, X, Y, Z) of the computed IR is produced in a different port, and then processed in a pipeline. Each branch performs the convolution between the IR and the input audio, and smoothes the transitions between different IRs via cross-fades in the time domain. The need for such cross-fades requires doing two convolutions in each pipe-line.

One last source of complexity: since sources and listener can move freely, cross-fades are not enough. The result of an overlap-and-add convolution involves two IR's which, among other differences, present different delays of arrival in the direct sound and first reflections. When such differences are present, the overlap-and-add

thought as an improved version of the MIDI protocol.

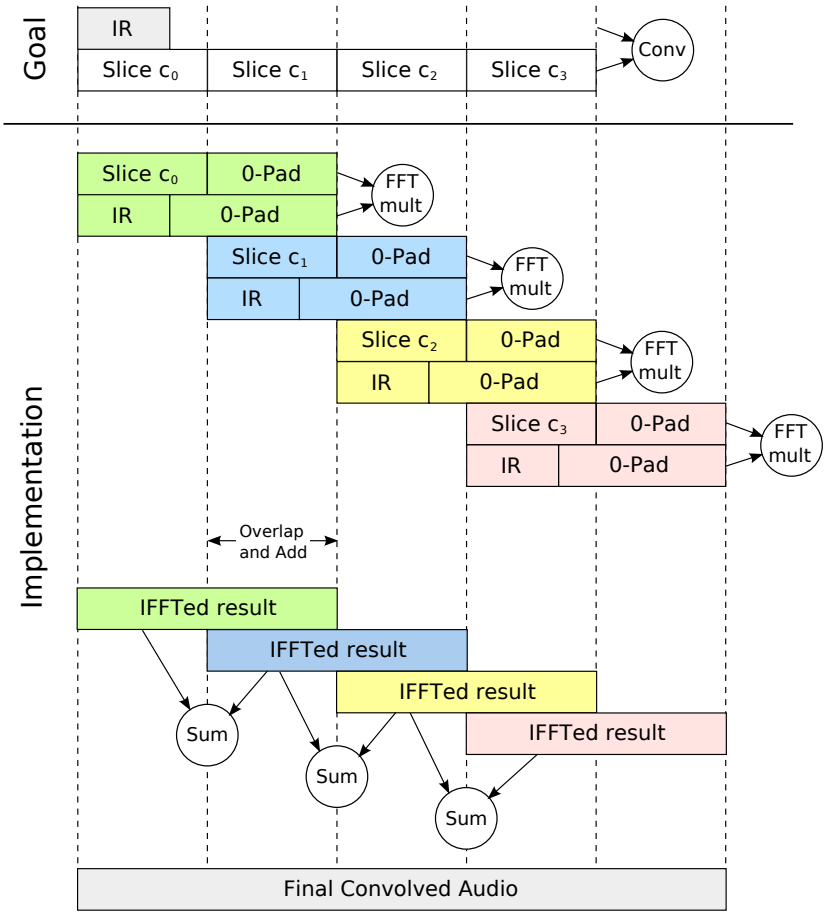


Figure 6.7: A simplification of the partitioned-convolution algorithm performed by the CLAM's "Convolution" processing.

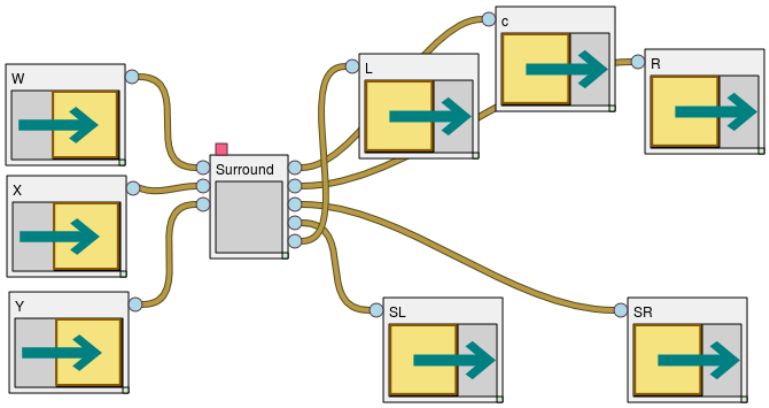


Figure 6.8: A CLAM network that decodes first order Ambisonics (B-Format) to Surround 5.0.

will have a discontinuity or clip, which the user notices as an annoying artifact.

This problem has been solved in this case by taking advantage of the two branches that were already needed for cross-fading the IR transition. The key point is to restrict how IRs change, so that only one branch can be clipped at a time. With this restriction, the problem can be solved by means of the *XFade* and *Delay* processings. The *Delay* processing produces two IR outputs: the first is just a copy of the received IR and the second is a copy of the IR received in the previous execution. To ensure that, at least, one overlap-and-add will be clip-free this processing will "hold" the same output when a received IR object only lasts one frame. The *XFade* reads the IR objects identifiers —and hence, its 4 input ports— and detects when and which branch is carrying a clipped frame, to decide which branch to select or to perform a cross-fade between the two.

In the last step, the listener's orientation is used to rotate the B-Format accordingly. The rotated output is then ready to be streamed to one or more decoders for exhibition systems.

6.2.4 Applying the TTSDF Model to the B-Format Rendering Network

The most interesting network regarding its processing execution scheduling is the B-Format renderer shown in figure 6.6. To analyse its TTSDF scheduling we start by simplifying the graph (figure 6.10) because most of the processings exhibit the same rate. Specifically, there are three different execution rates in the graph:

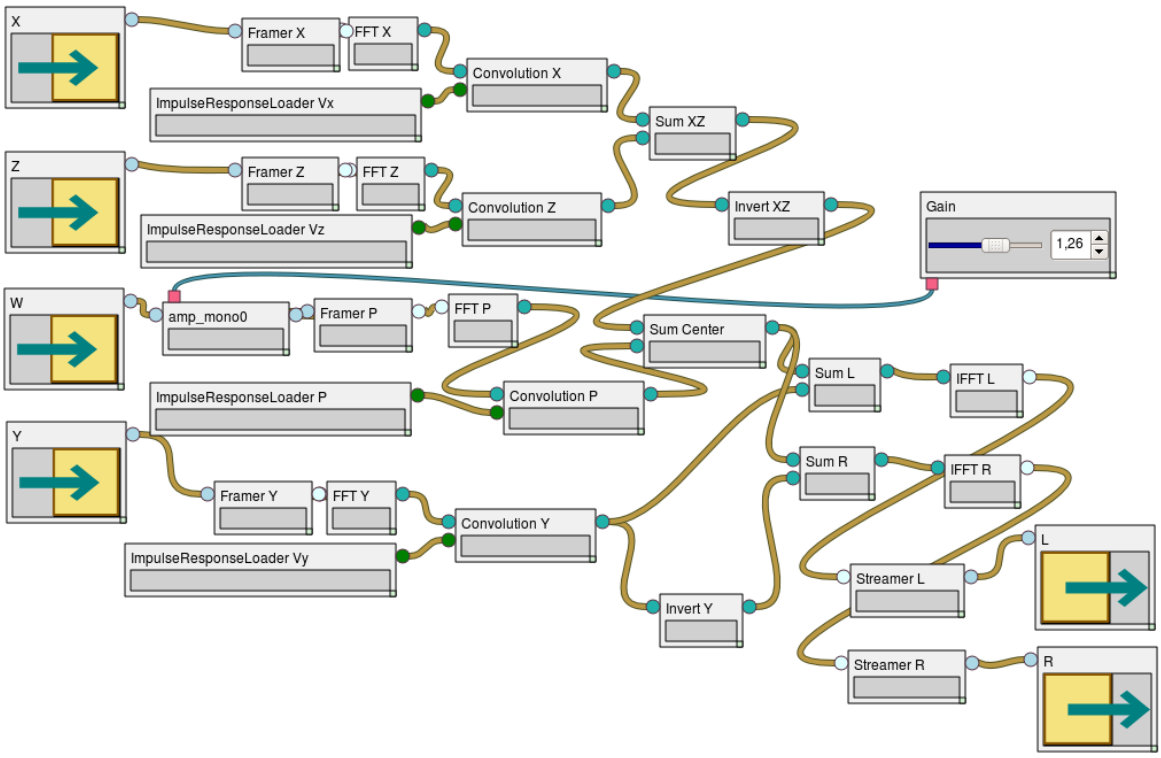


Figure 6.9: CLAM network that converts 3D-audio in B-Format to 2 channels binaural, to be listened with headphones. The technique is based in simulating how the human head and ears filters the audio frequencies depending on the source direction.

First, the *Input* and *Output*, which have a rate that depends on the fragment size[4] the audio driver uses in its callback. Second, the *CameraTracking*, which must run at 25 frames per second (that is, when its input port consumes 1920 samples with sample-rate of 48000 Hz). And third, all the rest: The *ImpulseResponseCalculatedOnTheFly* the *AudioWindowing*, the *FFT* etc.

In the simplified graph depicted in figure 6.10 we have removed most of the processings in the last category, only leaving the main pipeline, namely *AudioWindowing, FFT, Convolution, IFFT, OverlapAndAdd*. This simplification does not alter the complexity of the final scheduling but just makes it shorter and more understandable. It is important to note that though the processings in this pipeline share the same execution rate, their port rates (that is, the number of tokens consumed and produced per execution) differ. This is clear in the case of the *FFT*, which consumes 1280 tokens (audio samples) and produces a single spectrum object.

In this application, most of the port-rates are configurable by the domain expert who designs the network. The exception being the port-rate of the *Input* and *Output*, which are related to the audio driver configuration. In some audio subsystems architectures —like *CoreAudio* for Mac OS X and *PulseAudio* for Linux and other OS's— the user can adjust the fragment size of each callback in sample precision. In other architectures —like *Alsa*, for Linux or *PortAudio*, for multiple platforms— the allowed fragment sizes are much more restricted: the user can only choose between a power of two.

In the present example we've chosen a 256 samples fragment size for the callback[5] , and hence the *Input* and *Output* port rate. Apart from the input and output processings port rates, the other decisions that have a prominent effect on the execution rates and scheduling are the port-rate for the *AudioWindowing* and *CameraTracking* processings. In this example they have been set to 640 and 1920 samples respectively.

The following execution rates and scheduling is the result of executing the TTSDF scheduling algorithm presented in section 4.4. The parenthesis shows the execution of each callback. This is a sequential scheduling with no parallelization.

- Executions per period $\vec{q} = \{6, 2, 6, 6, 6, 15, 15, 6\}$ corresponding to nodes:

[4]The *fragment size* is the number of samples present in each processing callback. It is also known as *period* or just *buffer size*.

[5]which, at 48000 Hz, means a 5 milliseconds latency inherent to the callback

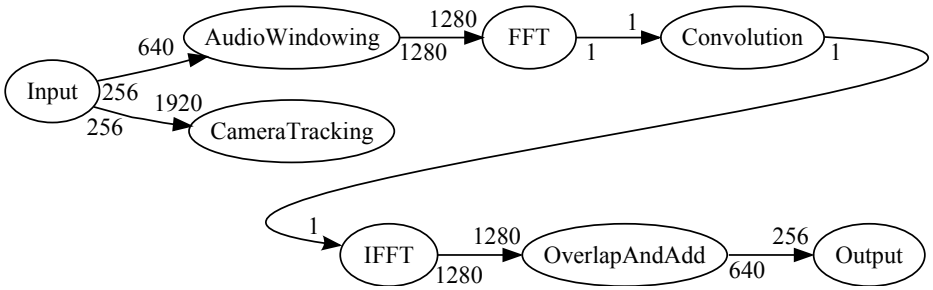
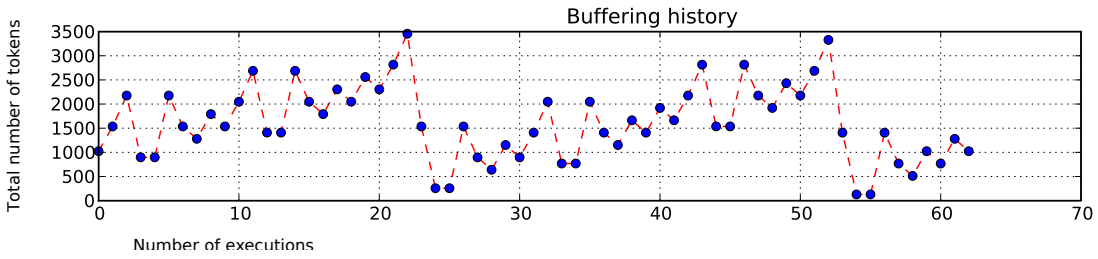


Figure 6.10: A simplified B-Format audio renderer graph its with port rates. Note that the *CameraTracking* node is a graph sink but is is not marked as output, in terms of the TTSDF model, because it does not relate to the driving time-triggered callback.

AudioWindowing, CameraTracking, Convolution, FFT, IFFT, Input, Output, OverlapAndAdd

- Time-triggered scheduling. Prologue + period: ($\mathbf{Input}_0$, $\mathbf{Output}_0$), ($\mathbf{Input}_1$, $\mathbf{Output}_1$) + ($\mathbf{Input}_0$, $AudioWindowing_0$, FFT_0, $Convolution_0$, $IFFT_0$, $OverlapAndAdd_0$, $\mathbf{Output}_0$), ($\mathbf{Input}_1$, $\mathbf{Output}_1$), ($\mathbf{Input}_2$, $AudioWindowing_1$, FFT_1, $Convolution_1$, $IFFT_1$, $OverlapAndAdd_1$, $\mathbf{Output}_2$), ($\mathbf{Input}_3$, $\mathbf{Output}_3$), ($\mathbf{Input}_4$, $\mathbf{Output}_4$), ($\mathbf{Input}_5$, $AudioWindowing_2$, $CameraTracking_0$, FFT_2, $Convolution_2$, $IFFT_2$, $OverlapAndAdd_2$, $\mathbf{Output}_5$), ($\mathbf{Input}_6$, $\mathbf{Output}_6$), ($\mathbf{Input}_7$, $AudioWindowing_3$, FFT_3, $Convolution_3$, $IFFT_3$, $OverlapAndAdd_3$, $\mathbf{Output}_7$), ($\mathbf{Input}_8$, $\mathbf{Output}_8$), ($\mathbf{Input}_9$, $\mathbf{Output}_9$), ($\mathbf{Input}_{10}$, $AudioWindowing_4$, FFT_4, $Convolution_4$, $IFFT_4$, $OverlapAndAdd_4$, $\mathbf{Output}_{10}$), ($\mathbf{Input}_{11}$, $\mathbf{Output}_{11}$), ($\mathbf{Input}_{12}$, $AudioWindowing_5$, $CameraTracking_1$, FFT_5, $Convolution_5$, $IFFT_5$, $OverlapAndAdd_5$, $\mathbf{Output}_{12}$), ($\mathbf{Input}_{13}$, $\mathbf{Output}_{13}$), ($\mathbf{Input}_{14}$, $\mathbf{Output}_{14}$)



- Optimal latency added by the TTSDF schedule prologue: 2 callback executions (512 samples).

It is interesting to compare the TTSDF scheduling above with a SDF scheduling below. Both have the same number of executions per period. Specific elements

of the TTSDF scheduling are: first, the separation between a preface and a periodic scheduling; and second the scheduling split in callback activations. We observe that the SDF scheduling adds more latency, since it initially execute 3 *Input*'s without a corresponding *Output*. Last, though the SDF scheduling could be adapted to a callback scheme by means of buffering in a separate thread (as shown in section 4.1), this adaptation does not guarantee the absence of gaps and jitter in the output.

- SDF scheduling (non Time-triggered) $\mathbf{Input}_0$, $\mathbf{Input}_1$, $\mathbf{Input}_2$, $AudioWindowing_0$, FFT_0, $\mathbf{Input}_3$, $Convolution_0$, $IFFT_0$, $\mathbf{Input}_4$, $OverlapAndAdd_0$, $AudioWindowing_1$, FFT_1, $\mathbf{Input}_5$, $\mathbf{Output}_0$, $Convolution_1$, $IFFT_1$, $\mathbf{Input}_6$, $\mathbf{Output}_1$, $OverlapAndAdd_1$, $\mathbf{Input}_7$, $\mathbf{Output}_2$, $AudioWindowing_2$, $CameraTracking_0$, FFT_2, $\mathbf{Input}_8$, $\mathbf{Output}_3$, $Convolution_2$, $IFFT_2$, $\mathbf{Input}_9$, $\mathbf{Output}_4$, $OverlapAndAdd_2$, $AudioWindowing_3$, FFT_3, $\mathbf{Input}_{10}$, $\mathbf{Output}_5$, $Convolution_3$, $IFFT_3$, $\mathbf{Input}_{11}$, $\mathbf{Output}_6$, $OverlapAndAdd_3$, $\mathbf{Input}_{12}$, $\mathbf{Output}_7$, $AudioWindowing_4$, FFT_4, $\mathbf{Input}_{13}$, $\mathbf{Output}_8$, $Convolution_4$, $IFFT_4$, $\mathbf{Input}_{14}$, $\mathbf{Output}_9$, $OverlapAndAdd_4$, $AudioWindowing_5$, $CameraTracking_1$, FFT_5, $\mathbf{Output}_{10}$, $Convolution_5$, $IFFT_5$, $\mathbf{Output}_{11}$, $OverlapAndAdd_5$, $\mathbf{Output}_{12}$, $\mathbf{Output}_{13}$, $\mathbf{Output}_{14}$
- Initial latency added by the SDF schedule (that is, number of input executions without a correspondent output execution): 3 callback activations.

Of course we could have chosen other port-rate settings that would have made the scheduling simpler. Actually, it is a common practice in many dataflow systems to only let the user choose port-rates that make the scheduling trivial. The main benefit of the TTSDF model is to overcome this shortcoming. It lets the domain expert choose any port-rate that makes sense from the signal processing point of view. If the port-rates are rate-consistent, the model ensures that it will run in real-time (provided enough processor resources) and with the optimum latency.

6.2.5 Applying the Dataflow Patterns to the B-Format Rendering Network

The 3D-Audio application analyzed in this section also exemplifies the use of the presented Pattern Language for Dataflow-Based Multimedia Processing Systems ("Dataflow Patterns" for short) —see chapter 5.

Though we find implementation examples for all the Dataflow Patterns in the CLAM framework infrastructure, here we take advantage of the previously

introduced B-Format renderer network (figure 6.6), to analyse how some of the patterns solve concrete design and implementation problems on this specific processing dataflow graph.

Typed Connections

One prominent characteristic of the B-Format renderer CLAM network diagram is that the ports are colored. The NetworkEditor —the user interface application for designing CLAM networks— only allows connecting ports with the same color, and hence, with the same type. Each color corresponds to a port's token C++ type. Also, the user gets a description of the port, including it's token type, when hovering the mouse pointer over the port.

A dataflow designer can extend the used types very easily. A token type can be any C++ type. Adding a new type to CLAM's dataflow is just a matter of declaring the token type of a port[6] in a Processing class definition —which usually is compiled in a dynamic loadable library (or plugin). The key point is that the framework infrastructure is totally agnostic on the port types it has to manage.

Letting the framework treat ports homogeneously, while letting the processing access their tokens in a strongly typed manner, and at the same time, letting the ports be extensible by plugins, is a big design challenge. This design problem is addressed by the Typed Connections pattern (in section 5.2.4).

Driver Ports

The analyzed network uses both synchronous flow and asynchronous flow. In CLAM's NetworEditor synchronous (or stream) ports are represented with circles at the left and right side of a processing box. Event ports (or controls in CLAM's nomenclature) are represented with squares on top or bottom of a processing.

The firing rules of a processing only depends on the stream flow. In other words, a processing can fire only when the tokens at the synchronous ports inputs satisfies the established port-rates. On the other hand, the asynchronous flow made of events does not have any effect on when the processing fires. In this example, events can be seen as a controlled way to change attribute values in a target processing.

[6]The C++ syntax used for an input port is the following: `CLAM::InPort<MyClass> _inport;`. This construct is typically found as a concrete Processing class attribute declaration.

The design allowing this duality of flows is addressed by the Driver Ports pattern (in section 5.2.2).

Multi-rate Stream Ports

The same network have to deal with connections between ports with different port-rates —that is, number of tokens produced and tokens consumed. For example, the *AudioInput* processing outputs 256 samples on each execution, while the *AudioWindowing* processing consumes them at 640 samples per execution. The Multi-rate Stream Ports pattern (in section 5.3.2) addresses how to design the necessary buffers to connect such multi-rate ports.

Multiple Window Circular Buffer

In the analyzed network, some out ports are connected to multiple in ports. The *MyFFT* processing, for example, is connected to 8 convolution processings. The design problem here is how to design a buffering data structure that supports a single source of tokens with one writer and multiple readers, giving each one access to a contiguous subsequence of tokens. This problem is addressed by the Multiple Window Circular Buffer (in section 5.3.3).

Port Monitor

During the development process of the B-Format renderer dataflow, debugging aids were needed (and used) to assess the correctness of the algorithm. A typical situation consisted on feeding the network with a carefully chose small piece of audio, usually in loop, and then analyze the stream flowing through some of the out ports at run-time.

CLAM's NetworkEditor let the user add visual components like oscilloscopes and spectrogram connected to any suited out port. This was specially useful to analyze the B-Format impulse-responses in the spectral domain, the final (convolved) B-Format in the time domain, and the 5.1 surround output.

The design problem involved in this situation deals with the real-time programing restrictions. On one hand the flowing tokens have to be transferred to the visualization thread and the visualization should be fluid. On the other hand, the real-time dataflow processing cannot be locked nor preempted by the

visualization thread. This problem is addressed by the Port Monitor pattern (in section 5.4.2).

6.3 Visualization of audio streams in Streamcatcher

Though it's recent publication, the TTSDF model has already been used in other systems besides CLAM. Specifically, the model has been implemented by Martin Gasser, a researcher at the Austrian Research Institute for Artificial Intelligence (OFAI), and integrated it into "Streamcatcher", an application for visualizing audio streams arranged by similarity [Gasser and Widmer, 2008]. Additionally, this application implements one of the most innovative patterns of our pattern language, the Port Monitor. This application of the TTSDF model and Port Monitor pattern to a system which is totally independent from the one that motivated their development (that is, CLAM) gives an additional validation of the model and pattern language.

6.3.1 Context

Martin approached the CLAM developers in November 2008 explaining he was currently working on new scheduling strategies for dataflows that could cope better with real-time. This arrived in perfect timing because at that time the TTSDF model was just developed and documented, and it targeted the same real-time problems Martin was facing. Therefore, Martin became an early adopter of the TTSDF model, and provided a totally independent implementation of the model run-time and it's scheduling algorithm, as well as the Port Monitor pattern.

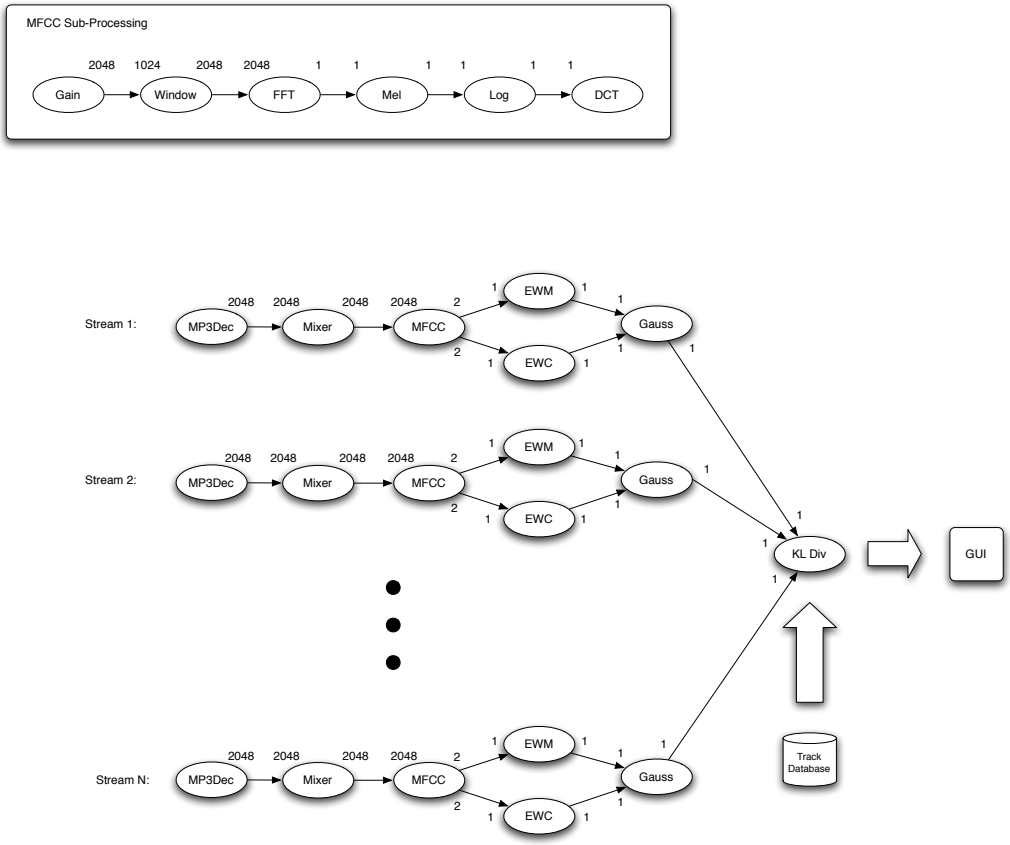


Figure 6.11: Streamcatcher general processing graph and its MFCC (mel-frequency cepstral coefficients) sub graph. Note that multi-rate is caused by audio windowing of the MFCC graph

6.3.2 Application of the TTSDF model and Port Monitor pattern in Streamcatcher

Streamcatcher in an application that allows to explore visualizations of online audio streams (e.g. radio streams). The visualization is defined by prototypical instances from personal music collections in a content-based approach. Streamcatcher generates an animated visualization that place similar audio streams together. The computation of music similarity and the visualization happen in real-time.

Streamcatcher runs on top of a lightweight dataflow framework called “Flower”. Figure 6.11 shows the processing TTSDF graph of Streamcatcher. The algorithm performed by this graph goes like follows: It decodes and analyzes internet MP3 streams, incrementally builds statistical models by using exponentially weighted

estimators for mean (*EWM* node) and covariance (*EWC* node), it continuously calculates a distance measure (KL divergence node) between each MP3 stream and a set of music clips supplied by the user, and visualizes the result by embedding stream representations in an animated 2D visualization of the music clip distances. To calculate the positions of the music clips, a multidimensional scaling algorithm is used.

Additionally to running the graph in real-time using the TTSDF static scheduling algorithm, Streamcatcher also makes use of one of our design patterns: The *KLDiv* and *Mixer* nodes implement the Port Monitor pattern at their out-ports. This design allows the application to show waveforms and similarity data in the visualization module running on the GUI thread, while communicating safely (in a lock-free way) with the audio-processing (TTSDF) thread.

More complex multi-rate applications are currently being developed with the Streamcatcher TTSDF framework (Flower). The new applications include a real-time vocal formant recognizer and a phase vocoder.

6.4 Summary

This chapter has presented three case studies that shows how the proposed time-triggered model and dataflow pattern language are used in real-life systems.

Both abstractions (the model and the pattern language) are implemented within the CLAM framework infrastructure. Therefore, CLAM and it's applications are presented as the first case study. The presence of successful applications implemented with the framework validates the abstractions. CLAM is a framework for rapid development of cross-platform audio and music applications. It is well described by the 4MPS meta-model reviewed in another chapter (see

section 3.2). This chapter has reviewed the framework components and its main applications.

The second case study is a specific multimedia system also developed with CLAM. It's underlying dataflow processing exhibits multi-rate, and exhibits an interesting time-triggered scheduling. Moreover, the whole system involves audio, video and 3D graphics. In this case study, we thoroughly analyse both the benefits of the model and the patterns.

The third and last case study is an application for visualizing audio streams arranged by similarity, built by researchers of the Austrian Research Institute for Artificial Intelligence (OFAI). The application is called Streamcatcher, and it's underlying dataflow framework, implements both the TTSDF model and the Port Monitor pattern. Interestingly, the OFAI application provides a totally independent implementation that gives an additional validation to the model and pattern.

Next chapter concludes this dissertation and gives future lines of research.

CHAPTER 7

Conclusions

The research condensed in this dissertation was motivated by the problems faced during years of activity building and analysing real-time multi-rate dataflow systems.

In this dissertation we have addressed two distinct but related problems: the lack of models of computation that can deal with continuous multimedia streams processing *in real-time*, and the lack of appropriate abstractions and systematic development methods that enables the effective implementation of such models of computation.

We have reviewed the background technologies upon which real-time multimedia systems are built, and the prior art that addresses similar problems. We have shown how existing abstractions that supports multi-rate dataflow processing fall short at covering the needs imposed by real-time restrictions, on the one hand; and neither provide systematic methodologies to build dataflow-based applications, on the other hand.

Next, we have proposed a solution for each problem: a model of computation (TTSDF) capable of running dataflow graphs within real-time constrains, and a pattern language that provides a systematic methodology to reuse design decisions

for building multimedia processing applications.

As a case study, we have presented an object-oriented framework for multimedia processing (CLAM), and specific applications, using the framework, that we have built. We have also presented a third party application and framework (Streamcatcher) from an Austrian research institute. All these frameworks and applications make extensive use of the contributed TTSDF model and pattern language.

We believe that multi-rate dataflows are still not widely used in multimedia domains today —with exceptions, like Marsyas [Tzanetakis, 2008] and CLAM frameworks—, mainly because of the previously stated lack of appropriate abstractions, and run-time frameworks. Leveraging the declarative style of dataflow combined with multi-rate and real-time capabilities promises more immediate and efficient systems —for example, by making good use of available multi-processors—, and, not less important, more expressive and human understandable abstractions.

Some of the multimedia sub-domains that we believe might take advantage of the contributed abstractions are: video coding and decoding, multi-frame-rate video processing, joint video and audio processing, audio processing in the spectral domain, real-time video and audio feature extraction using arbitrary sized token windows, and real-time computer graphics.

In the next sections we are going to, first, summarise the main contributions of the thesis, and second, present a detailed list of the contributions.

7.1

Summary of Contributions

This dissertation addresses the shortcoming of dataflow models of computation regarding real-time capabilities by formally describing **a new model that we**

named *Time-Triggered Synchronous Dataflow (TTSDF)* . The main innovation beyond the state of the art is that the TTSDF model schedules a dataflow graph in cyclic schedules that can be interleaved by several time-triggered "activations" in a way that inputs and outputs of the processing graph are regularly serviced.

We have shown that the TTSDF model has the same expressiveness (or equivalent computability) as the well known Synchronous Dataflow (SDF) model. We also have demonstrated that the model guarantees operating with the minimum latency and absence of gaps or jitter in the output. Finally, we have shown that it enables run-time load balancing between callback activations and parallelization.

The TTSDF model and the actor-oriented models in general are not and off-the-shelf solutions and do not suffice for building multimedia systems in a systematic and engineering approach. We have addressed this problem by proposing **a catalog of domain-specific *design patterns* organized in a *pattern language***. A pattern provides design reuse. It is a proven solution to a recurring generic design problem. It pays special attention to the context in which is applicable and the competing forces it needs to balance and the consequences of its application.

The thesis also contributes **an open-source framework (CLAM)** that implements and exemplifies all the previous abstractions and constructs. This framework is not only a platform for experimenting with models of computation and software design, but it is actually used in real-life projects in the multimedia domain, allowing domain experts to rapid prototype their real-time applications, using high-level and domain-specific abstractions, without needing to worry about implementation details.

7.2 Detailed Contributions

This thesis has presented a number of novel ideas that address existing limitations of dataflow models in respect of real-time, and the lack of systematic methods to develop dataflow-based multimedia systems. This section enumerates them all. The main contributions are emphasized with a bold font-style.

1. **An actor-oriented model of computation**, in the family of dataflow, that we have named *Time-Triggered Synchronous Dataflow (TTSDF)*. This model overcomes the timeliness limitations of existing dataflow models, and hence, adds real-time capabilities to the dataflow models. The TTSDF model has been extensively exposed in chapter 4. It's main properties are:
   (a) Combines actors associated with time with untimed actors
   (b) Retains token causality and dataflow semantics.
   (c) Statically (before run-time) schedulable.
   (d) Avoid jitter and uses optimum amount of latency. A superior bound for the latency is given by the model.
   (e) Fits naturally in callback-based architectures.
   (f) Enables static analysis for optimum distribution of run-time load among callbacks
   (g) The scheduling is parallelizable in multiple processors using well known techniques.
   (h) The callback-based scheduling algorithm is easily adaptable to other dataflow models (BDF, DDF)
2. Precise semantics for dataflow schedulings in callback-triggered architectures, which can be used in different models of computation. Presented in section 4.2.1.

3. A formal notation for the callback-triggered semantics, based on formal languages. This notation is useful to reason about schedulability of dataflow models. Presented in section 4.2.2.

4. **A design pattern language** for dataflow-based multimedia processing systems. It enables software developers to reuse tested design solutions into their systems, and thus, applying systematic solutions to domain problems with predictable trade-offs, and efficiently communicate and document design ideas.

   These characteristics represents a significant departure from other approaches that focus on reusing code (libraries, frameworks).

   The pattern language, has been presented in chapter 5. The pattern language, though not comprehensive in all the multimedia processing domain, focuses and is organized in the following four aspects :

   (a) *General Dataflow Patterns:* Address problems about how to organize high-level aspects of the dataflow architecture, by having different types of modules connections. Presented in section 5.2.

      i. *Semantic Ports* addresses distinct management of tokens by semantic.

      ii. *Driver Ports* addresses how to make modules executions independent of the availability of certain kind of tokens.

      iii. *Stream and Event Ports* addresses how to synchronize different streams and events arriving to a module.

      iv. *Typed Connections* addresses how to deal with typed tokens while allowing the network connection maker to ignore the concrete types.

   (b) *Flow Implementation Patterns:* Address how to physically transfer tokens from one module to another, according to the types of flow defined by the *general dataflow patterns*. Tokens life-cycle, ownership and memory management are recurrent issues in those patterns. Presented in section 5.3.

      i. *Propagating Event Ports* addresses the problem of having a high-priority event-driven flow able to propagate through the network.

ii. *Multi-rate Stream Ports* addresses how stream ports can consume and produce at different rates;

iii. *Multiple Window Circular Buffer* addresses how a writer and multiple readers can share the same tokens buffer.

iv. *Phantom Buffer* addresses how to design a data structure both with the benefits of a circular buffer and the guarantee of window contiguity.

(c) *Network Usability Patterns:* Address how humans can interact with dataflow networks without compromising the network processing efficiency. Presented in section 5.4.

i. *Recursive Networks* makes it feasible for humans to deal with the definition of big complex networks;

ii. *Port Monitor* addresses how to monitor a flow from a different thread, without compromising the network processing efficiency.

(d) *Visual Prototyping Patterns:* Address how domain experts can generate applications on top of a dataflow network, with interactive GUI, without needing programming skills. Presented in section 5.5.

i. *Visual Prototyper* Addresses how to dynamically build a graphical user interface that interacts with the underlying dataflow model. This pattern enables rapid applications prototyping using (but not restricted to) visual tools.

5. Showing that design patterns provide useful design reuse in the domain of multimedia processing systems. And showing that all the patterns can be found in different applications and contexts. The known uses of each pattern have been summarized in section 5.7.1.

6. Showing how design patterns are useful to communicate, document and compare designs of audio systems. The design communication value of patterns has been shown in section 5.7.2.

7. **An open-source framework** —CLAM— that implements all the concepts and constructs presented in the thesis: the TTSDF model of computation and all the proposed design patterns. CLAM enables multimedia domain experts to quickly prototype (that is: with visual tools, without coding)

real-time applications. Not all kind of applications but a significant part. CLAM is not only a test-bed for theoretic concepts of this thesis, but a tool actually used for real-world multimedia systems (specially in the music and audio sub-domain). CLAM and its applications have been reviewed as a case study for the time-triggered model and pattern language.

CLAM is freely available and, since it is free/libre software its source is available for study and modification. CLAM has been presented in section 6.1.

7.3

Open Issues and Future Work

7.3.1 Future Work in Time-Triggered Dataflows

The scheduling algorithm we have proposed have a lot of room for improvements and extensions. The most basic of these improvements is *balancing the run-time load among callback activations*. The current scheduling algorithm does not attempt to optimize the run-time load, and this is clear in the scheduling examples presented. Given that we have real-time restrictions, and each time-triggered callback must end before it's deadline, the scheduling should optimize for the worst-case callback activation. If individual actors run-time execution loads are given, such optimal scheduling should be easy to obtain —with a drawback of combinational cost.

New algorithms introducing trade-offs between optimizing the latency and the run-time load should be researched. The more latency we add to the scheduling, the more freedom the scheduler will have to ordering the firings, and hence, optimizing the run-time load. Of course, this trade-off should be left to the system designer.

The parallel scheduler has been summarized, but it should be implemented, and the aforementioned run-time balancing optimizations should also be studied for the parallel case.

Assuming that the run-time of each actor execution is known "a priori" is not very realistic. Dynamic techniques should be researched to estimate the actor's run-time.

Dataflow models execute actors atomically. When a dataflow network combines low-rate actors that performs expensive executions with high-rate actors that performs inexpensive executions, the run-time load is solely determined by the low-rate expensive actor. This is the case, for example, of real-time audio processing systems involving very long Fast Fourier Transforms. To overcome this problem, new techniques combining preemption of low-rate actors with dataflow schedulings should be researched.

Open architectures (such as audio or video plugins architectures) that leverage parallelism and multiprocessors but are still compatible with callbacks for input and output should be studied. Such architectures would not see the users callback as a black-box function, but would have access to the dataflow graph declaration.

7.3.2 Future Work in Multimedia Dataflow Patterns

The rich area of actor-oriented models of computation would benefit from the patterns techniques. Choosing between available models is hard. Patterns should be developed to facilitate this choice, emphasizing the forces and design consequences of each model. Similar to our pattern catalog for dataflow models, researching patterns to implement and enrich other models of computation would be useful.

Some actor-oriented related techniques such as Neuendorffer's *Reconfiguration of Actor-oriented Models* (see [Neuendorffer, 2005]) should also benefit from being reworked as a collection of patterns.

Empirical Quantitative Evaluation

Carrying out empirical evaluation of the teaching value of patterns, involves evaluating how individuals perform on technical problems when they know the patterns and when they do not. Due to the technical and specialized nature of our patterns, we believe that finding a significant sample of individuals would be plainly impossible.

Another kind of empirical study suitable for patterns takes a "Darwinian" approach. Consists on collecting data from many software projects under development: comments in the code, logs of Version Control Systems (such as CVS, CVS, GIT, etc.), mailing-lists discussions, and the like. Then automatically analyze and search for traces of pattern usage. Some other metrics can then capture the activity and "health" of the project. Thus, projects that use patterns and projects that do not can be compared.

Actually, this approach has been used in several studies on *general* patterns in open-source projects [Hahsler, 2004]. Interestingly enough, they detect a clear correlation between the adoption of patterns and the amount of development activity. This kind of quantitative study for the presented patterns would be very interesting. However, it requires many projects adopting those patterns, and this takes time.

Growing the Pattern Language

The presented work focuses on dataflow-based infrastructure for multimedia systems. However, there are many multimedia systems sub-domains where design patterns should be mined. Collecting a complete pattern catalog is a cumbersome work far beyond of a single person effort. On the contrary, it is more appropriate (and fun) to address as a living thing evolved by a community, similarly to the Wikipedia project.

The presented pattern language could be further refined and completed. It could be improved, for example, by splitting some of its bigger patterns into smaller ones. The following is a brief description of possible future patterns that would fit into the pattern language. It should be assessed that they hold the needed pattern qualities. Nevertheless, the list gives the directions where the catalog could grow.

1. *Controller Module* Addresses how user events from the (low-priority) out-of-band partition can enter into the (high-priority) in-band partition and propagated through the event ports connections.

2. *Dynamic repository* Addresses how applications can incorporate new modules without needing a rebuild, while keeping all modules organized in hierarchical structure, using meta-data.

3. *Configuration Time* Addresses how expensive changes, like memory allocation, in modules can be done, without effecting the network execution, while providing error handling.

4. *Configuration Object* Addresses how modules can be configured by the user without having to define user interfaces for each module, while allowing configurations persistence.

5. *Enhanced Types* Addresses how to separate the type parameters from the data itself for an efficient processing, while allowing automatic type compatibility checking and rich data representation.

6. *Partitioned streaming* Addresses how the flow in a network can be partitioned in high-level orchestrated tasks, when big buffers are not viable. For example, a process needs a large number of consecutive tokens in its input to start producing its stream. User intervention might be needed to define the partition points. Outputs of every phase are stored in data pools. (Examples can be found in Unix pipes, i.e. *cat* | *sed* | *sort* | *sert*, and the D2K framework.)

7. *Stream time propagation* Addresses how stream-tokens associated time can be propagated though the network given that channels buffers (and possibly the modules internally) causes latency.

7.4

Additional Insights

Open-Source

We have found that patterns and open-source work synergystically. Open-source boosts patterns on three fronts: One, open-source projects give material to pattern writers for new patterns to mine; two, they are a source of concrete code examples to people learning and using patterns; and three, they provide the necessary data for a quantitative evaluation of their effectiveness in real-life projects.

On the other hand, patterns also boost open-source. As [Seen et al., 2000] observes the design pattern adoption score very high in open-source developers. Specifically, patterns require no infrastructure investment, they can be adopted bottom-up and visible pattern adoption advertises competence. All three properties are certainly more important in an open-source environment than in a traditional company where the necessary infrastructure is provided and the management controls the development process. Another important aspect is about communication efficiency. The channels where communication takes place in open-source development (mailing lists, chats, forums...) are not appropriate for verbose design descriptions and they motivate the use of a concise vocabulary for communicate design ideas. Finally, our experience is that code documented with design patterns is many times easier to understand. Patterns are not mere design solution but they also carry a deeper knowledge in form of "consequences" or "forces resolution".

Final Conclusion

This thesis proposes abstractions taking three different forms: dataflow models, design pattern languages and frameworks. They all address the general challenges that experts in multimedia domain face: The features and complexity of

their applications are growing rapidly, while, real-time requirements and reliability standards increases.

We believe that these new abstractions represents a step forward in addressing this problems, and contributes to make real-time multimedia computing "grow up" out of the craftsman state into a genuinely scientific and more mature state.

APPENDIX A

Related Publications

This annex provides a list of published articles and authored open-source software.

1. **Olaiz, N. Arumí, P. Mateos, T. García, D. 2009.** *3D-Audio with CLAM and Blender's Game Engine* Proceedings of the 7th International Linux Audio Conference (LAC09); April 2009; Parma, Italy.

2. **Arumí, P and Amatriain, X. 2008.** *Time-triggered Static Schedulable Dataflows for Multimedia Systems* Proceedings of the Sixteenth Annual Multimedia Computing and Networking (MMCN'09); San Jose, California, USA.

3. **Arumí, P. García, D. Mateos, T. Garriga, A and Durany, J. 2008**, *Real-time 3D audio cinema for digital cinema*, Proceedings of the 2nd joint ASA-EAA conference Acoustics 08, Paris.

4. **Bailer, W. Arumí, P. Mateos, T. Garriga, A. Durany, J. and García, D. 2008** *Estimating 3D Camera Motion for Rendering Audio in Virtual Scenes*, 5th European Conference on Visual Media Production (CVMP 2008).

5. **Amatriain, X. Arumí, P. García, D. 2007.** *A framework for efficient and rapid development of cross-platform audio applications* **ACM Multimedia Systems Journal**

6. **García, D. Arumí, P. 2007.** *Visual prototyping of audio applications* Proceedings of 5th International Linux Audio Conference; Berlin, Germany

7. **Arumí, P. Sordo, M. García, D. Amatriain, X. 2006.** *Testfarm, una eina per millorar el desenvolupament del programari lliure* Proceedings of V Jornades de Programari Lliure; Barcelona

8. **García, D. Arumí, P. Amatriain, X. 2006.** *Extraccio d'acords amb l'Anotador de Musica de CLAM* Proceedings of V Jornades de Programari Lliure; Barcelona

9. **Amatriain, X. Arumí, P. García, D. 2006.** *CLAM: A Framework for Efficient and Rapid Development of Cross-platform Audio Applications* Proceedings of ACM Multimedia 2006; Santa Barbara, CA

10. **Arumí, P. García, D. Amatriain, X. 2006.** *A Dataflow Pattern Language for Audio and Music Computing* Proceedings of Pattern Languages of Programs 2006; Portland, Oregon

11. **Arumí, P. 2006.** *Towards a Pattern Language for Sound and Music Computing* Doctoral Pre-Thesis Work. UPF. Barcelona

12. **Arumí, P. Amatriain, X. 2005.** *CLAM, An Object-Oriented Framework for Audio and Music* Proceedings of 3rd International Linux Audio Conference; Karlsruhe, Germany

13. **Amatriain, X. Arumí, P. 2005.** *Developing Cross-platform Audio and Music Applications with the CLAM Framework* Proceedings of International Computer Music Conference 2005; Barcelona

14. **Arumí, P. García, D. Amatriain, X. 2003.** *CLAM, Una llibreria lliure per Audio i Musica* Proceedings of II Jornades de Software Lliure; Barcelona, Spain

15. **Amatriain, X. Arumí, P. Ramirez, M. 2002.** *CLAM, Yet Another Library for Audio and Music Processing?* Proceedings of 17th Annual ACM

Conference on Object-Oriented Programming, Systems, Languages and Applications; Seattle (USA)

16. **Arumí, P. 2002.** *CLAM : C++ Library for Audio and Music* Master Thesis UPC. FIB. Barcelona

A.1

Published Open-Source Software

List of authored and co-authored software. All these projects are currently maintained.

- *CLAM*, the audio framework (described in section 6.1) —Recipient of the 2006 ACM multimedia open-source award, and participant project of Google Summer of Code 2007 and 2008[1].
- *TestFarm*, the client-server continuous building and testing tool —written in Python[2].
- *MiniCppUnit*, a simple C++ xUnit testing framework[3].
- *Wiko*, the Wiki Compiler —written in Python [4].

[1]CLAM web: http://clam-project.org
CLAM statistics: http://www.ohloh.net/projects/8306?p=CLAM
[2]TestFarm web: http://www.iua.upf.edu/~parumi/testfarm/
[3]MiniCppUnit web:http://www.iua.upf.edu/~parumi/MiniCppUnit/
[4]Wiko web: http://www.iua.upf.edu/~dgarcia/wiko/

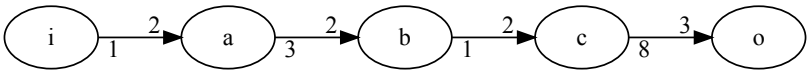
APPENDIX B

TTSDF Scheduling Examples

For each example we use the following conventions: Callback activations in the resulting scheduling are separated using parentheses "( )". The prologue and periodic parts are separated with a "+".

To allow comparing the TTSDF and SDF schedulings, we also give a non-time-triggered scheduling obtained with a SDF scheduling algorithm. Finally, we also give a diagram with the evolution of buffering during the periodic cycle. This diagram shows the total amount of tokens, summing all FIFO queues.

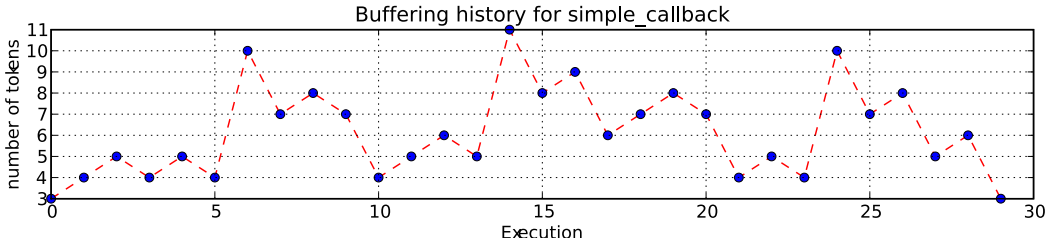
B.0.1 Simple TTSDF Pipeline



$$\Gamma = \begin{bmatrix} -2 & 0 & 0 & 1 & 0 \\ 3 & -2 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 \\ 0 & 0 & 8 & 0 & -3 \end{bmatrix}$$

- Executions per period $\vec{q} = \{4, 6, 3, 8, 8\}$ corresponding to nodes: *a, b, c, i, o*

- Time-Triggered scheduling. Prologue + period: ($\mathbf{i}_0$, $\mathbf{o}_0$), ($\mathbf{i}_1$, $\mathbf{o}_1$), ($\mathbf{i}_2$, $\mathbf{o}_2$) + ($\mathbf{i}_0$, a_0, b_0, a_1, b_1, c_0, $\mathbf{o}_0$), ($\mathbf{i}_1$, b_2, $\mathbf{o}_1$), ($\mathbf{i}_2$, a_2, b_3, c_1, $\mathbf{o}_2$), ($\mathbf{i}_3$, $\mathbf{o}_3$), ($\mathbf{i}_4$, a_3, b_4, $\mathbf{o}_4$), ($\mathbf{i}_5$, b_5, c_2, $\mathbf{o}_5$), ($\mathbf{i}_6$, $\mathbf{o}_6$), ($\mathbf{i}_7$, $\mathbf{o}_7$)

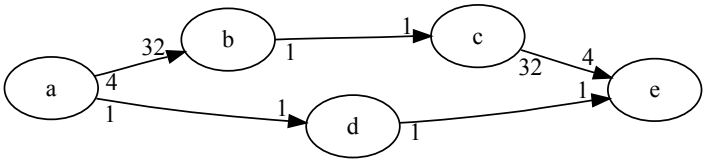


- Optimal latency added by the TTSDF schedule prologue: 3 callback activations.

- SDF scheduling (non Time-Triggered) $\mathbf{i}_0$, $\mathbf{i}_1$, a_0, b_0, $\mathbf{i}_2$, $\mathbf{i}_3$, a_1, b_1, c_0, $\mathbf{i}_4$, $\mathbf{o}_0$, b_2, $\mathbf{i}_5$, $\mathbf{o}_1$, a_2, b_3, c_1, $\mathbf{i}_6$, $\mathbf{o}_2$, $\mathbf{i}_7$, $\mathbf{o}_3$, a_3, b_4, $\mathbf{o}_4$, b_5, c_2, $\mathbf{o}_5$, $\mathbf{o}_6$, $\mathbf{o}_7$

- Initial latency added by the SDF schedule (that is, number of input executions without a correspondent output execution): 4 callback activations.

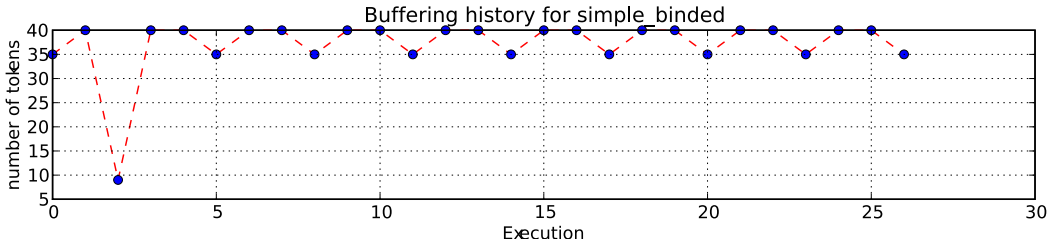
B.0.2 Split and Reunify



$$\Gamma = \begin{bmatrix} 4 & -32 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 32 & 0 & -4 \\ 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

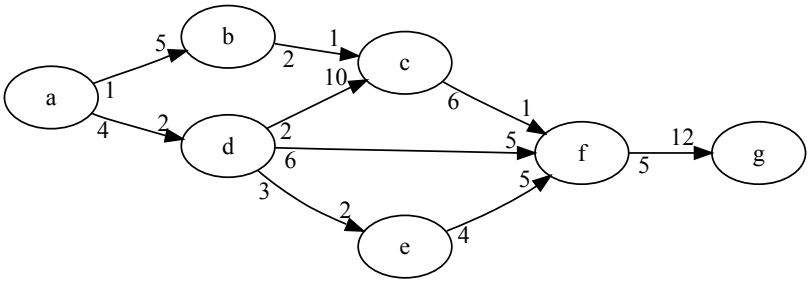
- Executions per period $\vec{q} = \{8, 1, 1, 8, 8\}$ corresponding to nodes: *a, b, c, d, e*

- Time-Triggered scheduling. Prologue + period: ($\mathbf{a}_0$, $\mathbf{e}_0$), ($\mathbf{a}_1$, $\mathbf{e}_1$), ($\mathbf{a}_2$, $\mathbf{e}_2$), ($\mathbf{a}_3$, $\mathbf{e}_3$), ($\mathbf{a}_4$, $\mathbf{e}_4$), ($\mathbf{a}_5$, $\mathbf{e}_5$), ($\mathbf{a}_6$, $\mathbf{e}_6$) + ($\mathbf{a}_0$, b_0, c_0, d_0, $\mathbf{e}_0$), ($\mathbf{a}_1$, d_1, $\mathbf{e}_1$), ($\mathbf{a}_2$, d_2, $\mathbf{e}_2$), ($\mathbf{a}_3$, d_3, $\mathbf{e}_3$), ($\mathbf{a}_4$, d_4, $\mathbf{e}_4$), ($\mathbf{a}_5$, d_5, $\mathbf{e}_5$), ($\mathbf{a}_6$, d_6, $\mathbf{e}_6$), ($\mathbf{a}_7$, d_7, $\mathbf{e}_7$)



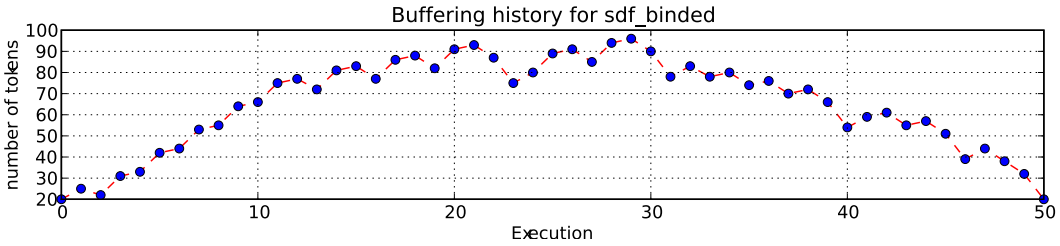
- Optimal latency added by the TTSDF schedule prologue: 7 callback activations.
- SDF scheduling (non Time-Triggered) $\mathbf{a}_0$, d_0, $\mathbf{a}_1$, d_1, $\mathbf{a}_2$, d_2, $\mathbf{a}_3$, d_3, $\mathbf{a}_4$, d_4, $\mathbf{a}_5$, d_5, $\mathbf{a}_6$, d_6, $\mathbf{a}_7$, b_0, c_0, d_7, $\mathbf{e}_0$, $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$, $\mathbf{e}_4$, $\mathbf{e}_5$, $\mathbf{e}_6$, $\mathbf{e}_7$
- Initial latency added by the SDF schedule (that is, number of input executions without a correspondent output execution): 7 callback activations.

B.0.3 Dense Graph



$$\Gamma = \begin{bmatrix} 1 & -5 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & -2 & 0 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -10 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & -5 & 0 \\ 0 & 0 & 0 & 3 & -2 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 4 & -5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & -12 \end{bmatrix}$$

- Executions per period $\vec{q} = \{5, 1, 2, 10, 15, 12, 5\}$ corresponding to nodes: a, b, c, d, e, f, g
- Time-Triggered scheduling. Prologue + period: $(\mathbf{a}_0, \mathbf{g}_0)$, $(\mathbf{a}_1, \mathbf{g}_1)$, $(\mathbf{a}_2, \mathbf{g}_2)$, $(\mathbf{a}_3, \mathbf{g}_3)$ + $(\mathbf{a}_0, b_0, d_0, e_0, d_1, e_1, d_2, e_2, d_3, e_3, d_4, e_4, c_0, d_5, e_5, f_0, d_6, e_6, f_1, d_7, e_7, f_2, \mathbf{g}_0)$, $(\mathbf{a}_1, d_8, e_8, f_3, d_9, e_9, f_4, \mathbf{g}_1)$, $(\mathbf{a}_2, c_1, e_10, f_5, e_11, f_6, e_12, f_7, \mathbf{g}_2)$, $(\mathbf{a}_3, e_13, f_8, e_14, f_9, \mathbf{g}_3)$, $(\mathbf{a}_4, f_10, f_11, \mathbf{g}_4)$



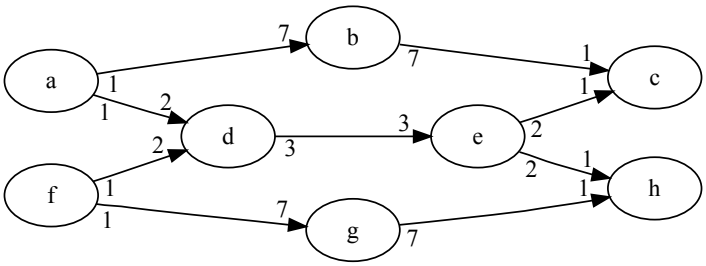
- Optimal latency added by the TTSDF schedule prologue: 4 callback activations.

- SDF scheduling (non Time-Triggered) $\mathbf{a}_0$, d_0, e_0, $\mathbf{a}_1$, d_1, e_1, $\mathbf{a}_2$, d_2, e_2, $\mathbf{a}_3$, d_3, e_3, $\mathbf{a}_4$, b_0, d_4, e_4, c_0, d_5, e_5, f_0, d_6, e_6, f_1, d_7, e_7, f_2, $\mathbf{g}_0$, d_8, e_8, f_3, d_9, e_9, f_4, $\mathbf{g}_1$, c_1, e_10, f_5, e_11, f_6, e_12, f_7, $\mathbf{g}_2$, e_13, f_8, e_14, f_9, $\mathbf{g}_3$, f_10, f_11, $\mathbf{g}_4$

- Initial latency added by the SDF schedule (that is, number of input executions without a correspondent output execution): 4 callback activations.

B.0.4 Graph with Optimum Between Bounds

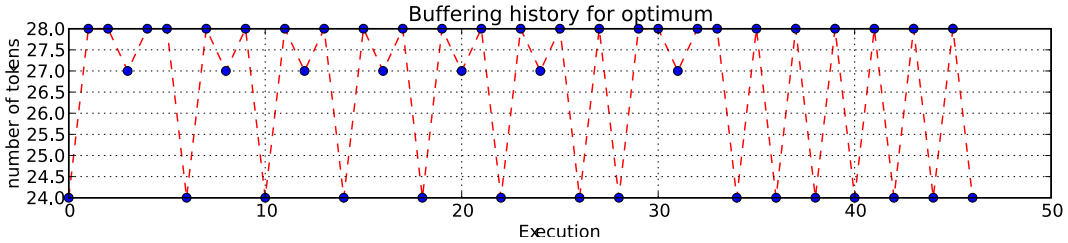
Example and name taken from [Ade et al., 1997].



$$\Gamma = \begin{bmatrix} 1 & -7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 7 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & -3 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & -1 \\ 0 & 0 & 0 & -2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & -1 \end{bmatrix}$$

- Executions per period $\vec{q} = \{14, 2, 14, 7, 7, 14, 2, 14\}$ corresponding to nodes: $a,\ b,\ c,\ d,\ e,\ f,\ g,\ h$

- Time-Triggered scheduling. Prologue + period: ($\mathbf{a}_0$, $\mathbf{f}_0$, $\mathbf{c}_0$, $\mathbf{h}_0$), ($\mathbf{a}_1$, $\mathbf{f}_1$, $\mathbf{c}_1$, $\mathbf{h}_1$), ($\mathbf{a}_2$, $\mathbf{f}_2$, $\mathbf{c}_2$, $\mathbf{h}_2$), ($\mathbf{a}_3$, $\mathbf{f}_3$, $\mathbf{c}_3$, $\mathbf{h}_3$), ($\mathbf{a}_4$, $\mathbf{f}_4$, $\mathbf{c}_4$), $\mathbf{h}_4$), ($\mathbf{a}_5$, $\mathbf{f}_5$, $\mathbf{c}_5$), $\mathbf{h}_5$) + ($\mathbf{a}_0$, $\mathbf{f}_0$, b_0, d_0, e_0, g_0, $\mathbf{c}_0$), $\mathbf{h}_0$), ($\mathbf{a}_1$, $\mathbf{f}_1$, d_1, e_1, $\mathbf{c}_1$, $\mathbf{h}_1$), ($\mathbf{a}_2$, $\mathbf{f}_2$, d_2, e_2, $\mathbf{c}_2$, $\mathbf{h}_2$), ($\mathbf{a}_3$, $\mathbf{f}_3$, d_3, e_3, $\mathbf{c}_3$, $\mathbf{h}_3$), ($\mathbf{a}_4$, $\mathbf{f}_4$, d_4, e_4, $\mathbf{c}_4$, $\mathbf{h}_4$), ($\mathbf{a}_5$, $\mathbf{f}_5$, d_5, e_5, $\mathbf{c}_5$, $\mathbf{h}_5$), ($\mathbf{a}_6$, $\mathbf{f}_6$, $\mathbf{c}_6$, $\mathbf{h}_6$), ($\mathbf{a}_7$, $\mathbf{f}_7$, b_1, d_6, e_6, g_1, $\mathbf{c}_7$, $\mathbf{h}_7$), ($\mathbf{a}_8$, $\mathbf{f}_8$, $\mathbf{c}_8$, $\mathbf{h}_8$), ($\mathbf{a}_9$, $\mathbf{f}_9$, $\mathbf{c}_9$, $\mathbf{h}_9$), ($\mathbf{a}_{10}$, $\mathbf{f}_{10}$, $\mathbf{c}_{10}$, $\mathbf{h}_{10}$), ($\mathbf{a}_{11}$, $\mathbf{f}_{11}$, $\mathbf{c}_{11}$, $\mathbf{h}_{11}$), ($\mathbf{a}_{12}$, $\mathbf{f}_{12}$, $\mathbf{c}_{12}$, $\mathbf{h}_{12}$), ($\mathbf{a}_{13}$, $\mathbf{f}_{13}$, $\mathbf{c}_{13}$, $\mathbf{h}_{13}$)

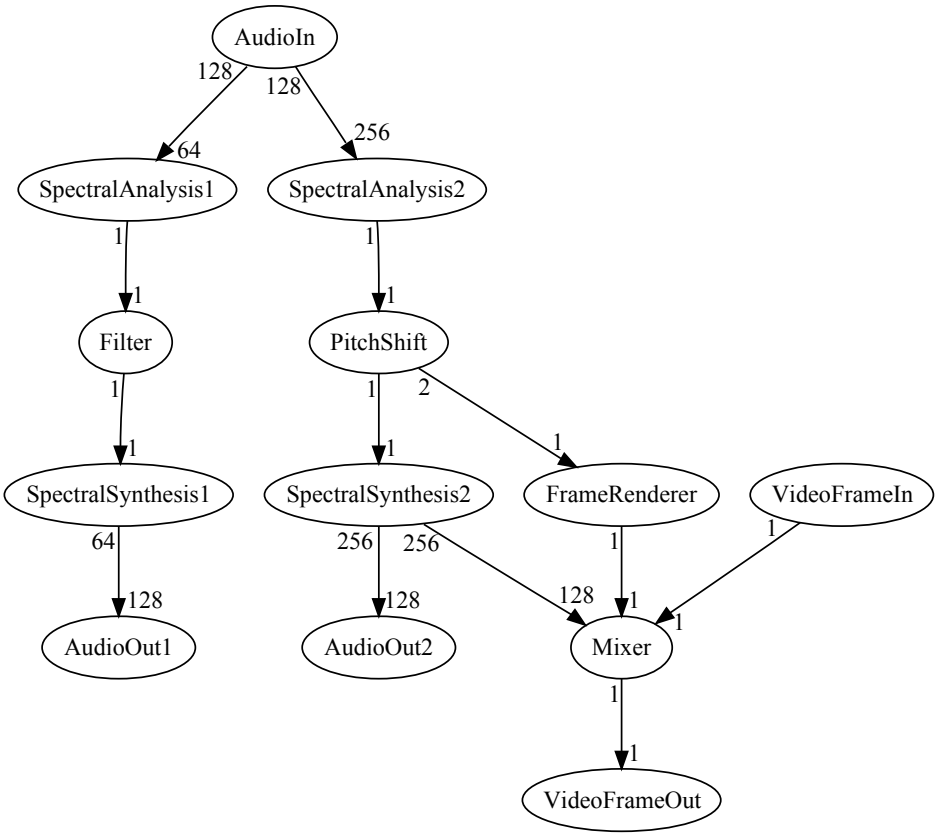


- Optimal latency added by the TTSDF schedule prologue: 6 callback activations.

- SDF scheduling (non Time-Triggered) $\mathbf{a}_0$, $\mathbf{f}_0$, $\mathbf{a}_1$, $\mathbf{f}_1$, $\mathbf{a}_2$, d_0, e_0, $\mathbf{f}_2$, $\mathbf{a}_3$, $\mathbf{f}_3$, $\mathbf{a}_4$, d_1, e_1, $\mathbf{f}_4$, $\mathbf{a}_5$, $\mathbf{f}_5$, $\mathbf{a}_6$, b_0, $\mathbf{c}_0$, d_2, e_2, $\mathbf{f}_6$, g_0, $\mathbf{h}_0$ (completed first output), $\mathbf{a}_7$, $\mathbf{c}_1$, $\mathbf{f}_7$, $\mathbf{h}_1$, $\mathbf{a}_8$, $\mathbf{c}_2$, d_3, e_3, $\mathbf{f}_8$, $\mathbf{h}_2$, $\mathbf{a}_9$, $\mathbf{c}_3$, $\mathbf{f}_9$, $\mathbf{h}_3$, $\mathbf{a}_{10}$, $\mathbf{c}_4$, d_4, e_4, $\mathbf{f}_{10}$, $\mathbf{h}_4$, $\mathbf{a}_{11}$, $\mathbf{c}_5$, $\mathbf{f}_{11}$, $\mathbf{h}_5$, $\mathbf{a}_{12}$, $\mathbf{c}_6$, d_5, e_5, $\mathbf{f}_{12}$, $\mathbf{h}_6$, $\mathbf{a}_{13}$, b_1, $\mathbf{c}_7$, $\mathbf{f}_{13}$, g_1, $\mathbf{h}_7$, $\mathbf{c}_8$, d_6, e_6, $\mathbf{h}_8$, $\mathbf{c}_9$, $\mathbf{h}_9$, $\mathbf{c}_{10}$, $\mathbf{h}_{10}$, $\mathbf{c}_{11}$, $\mathbf{h}_{11}$, $\mathbf{c}_{12}$, $\mathbf{h}_{12}$, $\mathbf{c}_{13}$, $\mathbf{h}_{13}$

- Initial latency added by the SDF schedule (that is, number of input executions without a correspondent output execution): 7 callback activations.

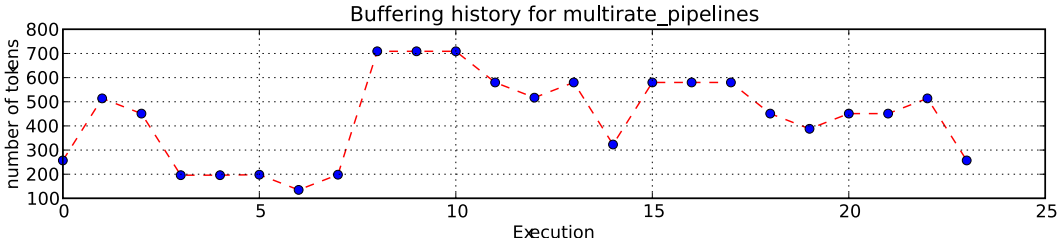
B.0.5 Audio and Video Multi-Rate Pipelines



$$\Gamma = \begin{bmatrix}
128 & 0 & 0 & 0 & 0 & 0 & 0 & -64 & 0 & 0 & 0 & 0 & 0 \\
128 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -256 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & -128 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 64 & 0 & 0 & 0 \\
0 & 0 & -128 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 256 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -128 & 0 & 0 & 0 & 0 & 256 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1
\end{bmatrix}$$

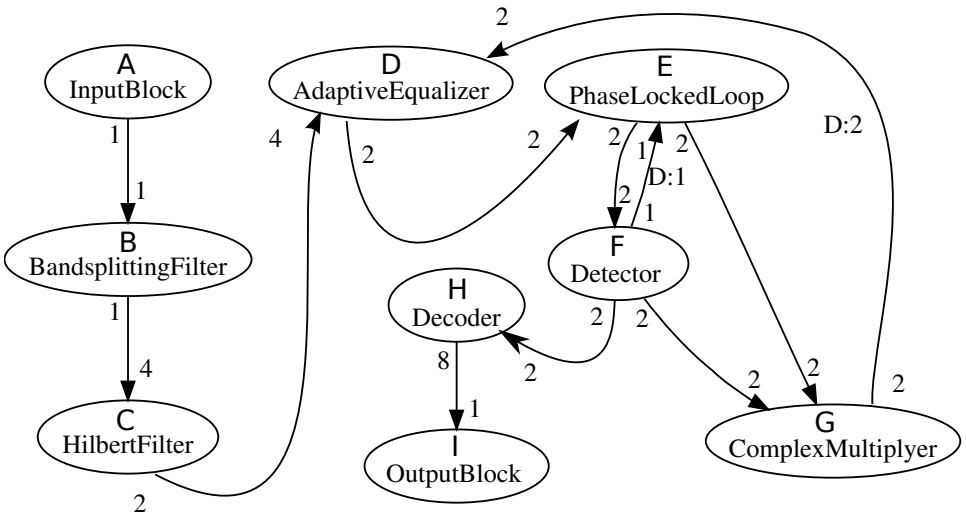
- Executions per period $\vec{q} = \{2, 2, 2, 4, 2, 2, 1, 4, 1, 4, 1, 2, 2\}$ corresponding to nodes: *AudioIn, AudioOut1, AudioOut2, Filter, FrameRenderer, Mixer, PitchShift, SpectralAnalysis1, SpectralAnalysis2, SpectralSynthesis1, SpectralSynthesis2, VideoFrameIn, VideoFrameOut*
- Time-Triggered scheduling. Prologue + period:

(**AudioIn**$_0$, **VideoFrameIn**$_0$, **AudioOut1**$_0$, **AudioOut2**$_0$, **VideoFrameOut**$_0$) + (**AudioIn**$_0$, **VideoFrameIn**$_0$, *SpectralAnalysis*1$_0$, *SpectralAnalysis*2$_0$, *Filter*$_0$, *PitchShift*$_0$, *SpectralAnalysis*1$_1$, *SpectralSynthesis*1$_0$, *SpectralSynthesis*2$_0$, *Filter*$_1$, *FrameRenderer*$_0$, *Mixer*$_0$, *SpectralAnalysis*1$_2$, *SpectralSynthesis*1$_1$, **AudioOut1**$_0$, **AudioOut2**$_0$), **VideoFrameOut**$_0$), (**AudioIn**$_1$, **VideoFrameIn**$_1$, *Filter*$_2$, *FrameRenderer*$_1$, *Mixer*$_1$, *SpectralAnalysis*1$_3$, *SpectralSynthesis*1$_2$, *Filter*$_3$, *SpectralSynthesis*1$_3$, **AudioOut1**$_1$, **AudioOut2**$_1$, **VideoFrameOut**$_1$)



- Optimal latency added by the TTSDF schedule prologue: 1 callback activations.
- SDF scheduling (non Time-Triggered) **AudioIn**$_0$, *SpectralAnalysis*1$_0$, **VideoFrameIn**$_0$, **AudioIn**$_1$, *Filter*$_0$, *SpectralAnalysis*1$_1$, *SpectralAnalysis*2$_0$, *SpectralSynthesis*1$_0$, **VideoFrameIn**$_1$, *Filter*$_1$, *PitchShift*$_0$, *SpectralAnalysis*1$_2$, *SpectralSynthesis*1$_1$, *SpectralSynthesis*2$_0$, **AudioOut1**$_0$, **AudioOut2**$_0$, *Filter*$_2$, *FrameRenderer*$_0$, *Mixer*$_0$, *SpectralAnalysis*1$_3$, *SpectralSynthesis*1$_2$, **VideoFrameOut**$_0$, **AudioOut2**$_1$, *Filter*$_3$, *FrameRenderer*$_1$, *Mixer*$_1$, *SpectralSynthesis*1$_3$, **VideoFrameOut**$_1$, **AudioOut1**$_1$
- Initial latency added by the SDF schedule (that is, number of input executions without a correspondent output execution): 2 callback activations.

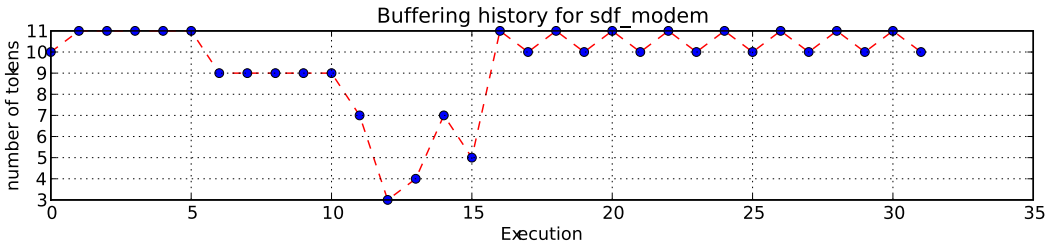
B.0.6 Simplified Modem



$$\Gamma = \begin{bmatrix} 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -4 & 0 & 0 & 0 \\ -4 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 \\ 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 2 \\ -2 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & -2 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 & 0 & -1 & 0 \end{bmatrix}$$

- Executions per period $\vec{q} = \{1, 8, 1, 1, 1, 2, 8, 8, 1\}$ corresponding to nodes: *AdaptiveEqualizer, BandsplittingFilter, ComplexMultiplyer, Decoder, Detector, HilbertFilter, InputBlock, OutputBlock, PhaseLockedLoop*

- Time-Triggered scheduling. Prologue + period: ($\mathbf{InputBlock}_0$, $\mathbf{OutputBlock}_0$), ($\mathbf{InputBlock}_1$, $\mathbf{OutputBlock}_1$), ($\mathbf{InputBlock}_2$, $\mathbf{OutputBlock}_2$), ($\mathbf{InputBlock}_3$, $\mathbf{OutputBlock}_3$), ($\mathbf{InputBlock}_4$, $\mathbf{OutputBlock}_4$), ($\mathbf{InputBlock}_5$, $\mathbf{OutputBlock}_5$), ($\mathbf{InputBlock}_6$, $\mathbf{OutputBlock}_6$) + ($\mathbf{InputBlock}_0$, $BandsplittingFilter_0$, $BandsplittingFilter_1$, $BandsplittingFilter_2$, $BandsplittingFilter_3$, $HilbertFilter_0$, $BandsplittingFilter_4$, $BandsplittingFilter_5$, $BandsplittingFilter_6$, $BandsplittingFilter_7$, $HilbertFilter_1$, $AdaptiveEqualizer_0$, $PhaseLockedLoop_0$, $Detector_0$, $ComplexMultiplyer_0$, $Decoder_0$, $\mathbf{OutputBlock}_0$), ($\mathbf{InputBlock}_1$, $\mathbf{OutputBlock}_1$), ($\mathbf{InputBlock}_2$, $\mathbf{OutputBlock}_2$), ($\mathbf{InputBlock}_3$, $\mathbf{OutputBlock}_3$), ($\mathbf{InputBlock}_4$, $\mathbf{OutputBlock}_4$), ($\mathbf{InputBlock}_5$, $\mathbf{OutputBlock}_5$), ($\mathbf{InputBlock}_6$, $\mathbf{OutputBlock}_6$), ($\mathbf{InputBlock}_7$, $\mathbf{OutputBlock}_7$)



- Optimal latency added by the TTSDF schedule prologue: 7 callback activations.

- SDF scheduling (non Time-Triggered) $\mathbf{InputBlock}_0$, $BandsplittingFilter_0$, $\mathbf{InputBlock}_1$, $BandsplittingFilter_1$, $\mathbf{InputBlock}_2$, $BandsplittingFilter_2$, $\mathbf{InputBlock}_3$, $BandsplittingFilter_3$, $HilbertFilter_0$, $\mathbf{InputBlock}_4$, $BandsplittingFilter_4$, $\mathbf{InputBlock}_5$, $BandsplittingFilter_5$, $\mathbf{InputBlock}_6$, $BandsplittingFilter_6$, $\mathbf{InputBlock}_7$, $BandsplittingFilter_7$, $HilbertFilter_1$, $AdaptiveEqualizer_0$, $PhaseLockedLoop_0$, $Detector_0$, $ComplexMultiplyer_0$, $Decoder_0$, $\mathbf{OutputBlock}_0$, $\mathbf{OutputBlock}_1$, $\mathbf{OutputBlock}_2$, $\mathbf{OutputBlock}_3$, $\mathbf{OutputBlock}_4$, $\mathbf{OutputBlock}_5$, $\mathbf{OutputBlock}_6$, $\mathbf{OutputBlock}_7$

- Initial latency added by the SDF schedule (that is, number of input executions without a correspondent output execution): 8 callback activations.

Bibliography

[Adam et al., 1974] Adam, T., Chandy, K., and Dickson, J. (1974). A Comparison of List Schedules for Parallel Processing Systems.

[Ade et al., 1997] Ade, M., Lauwereins, R., and Peperstraete, J. A. (1997). Data memory minimisation for synchronous data flow graphs emulated on dsp-fpga targets. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 64–69, New York, NY, USA. ACM Press.

[Agha, 1986] Agha, G. (1986). *Actors: A model of concurrent computation in Distributed Systems*. MIT Press, Cambridge, MA.

[Alexander, 1977] Alexander, C. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, USA.

[Amatriain, 2004] Amatriain, X. (2004). *An Object-Oriented Metamodel for Digital Signal Processing*. PhD thesis, Universitat Pompeu Fabra.

[Amatriain, 2007a] Amatriain, X. (2007a). A Domain-Specific Metamodel for Multimedia Processing Systems. *Multimedia, IEEE Transactions on*, 9(6):1284–1298.

[Amatriain, 2007b] Amatriain, X. (2007b). Clam: A framework for audio and music application development. *IEEE Software*, 24(1):82–85.

[Amatriain and Arumí, 2005] Amatriain, X. and Arumí, P. (2005). Developing cross-platform audio and music applications with the clam framework. In *Pro-*

ceedings of the 2005 International Computer Music Conferenc (ICMC'05). in press.

[Amatriain et al., 2002a] Amatriain, X., Bonada, J., Loscos, A., and Serra, X. (2002a). *DAFX: Digital Audio Effects (Udo Zölzer ed.)*, chapter Spectral Processing, pages 373–438. John Wiley and Sons, Ltd.

[Amatriain et al., 2002b] Amatriain, X., de Boer, M., Robledo, E., and Garcia, D. (2002b). CLAM: An OO Framework for Developing Audio and Music Applications. In *Proceedings of the 2002 Conference on Object Oriented Programming, Systems and Application (OOPSLA 2002)(Companion Material)*, Seattle, USA. ACM.

[Amatriain et al., 2005] Amatriain, X., Massaguer, J., Garcia, D., and Mosquera, I. (2005). The clam annotator: A cross-platform audio descriptors editing tool. In *Proceedings of 6th International Conference on Music Information Retrieval*, London, UK.

[Appleton, 1997] Appleton, B. (1997). Patterns and software: Essential concepts and terminology.

[Armstrong et al., 1996] Armstrong, J., Williams, R., Virding, M., and Wikstroem, C. (1996). *Concurrent Programing in Erlang*. Prentice-Hal.

[Aucouturier, 2006] Aucouturier, J. (2006). *Ten Experiments on the Modelling of Polyphonic Timbre*. PhD thesis, University of Paris 6/Sony CSL Paris.

[Aynsley and Long, 2005] Aynsley, J. and Long, D. (2005). Draft standard SystemC language reference manual. Technical report, Technical report, Open SystemC Initiative.

[Beck, 1988] Beck, K. (1988). Using pattern languages for object-oriented programs. In *ACM SIGPLAN*.

[Beck et al., 1996] Beck, K., Coplien, J. O., Crocker, R., Dominick, L., Meszaros, G., Paulisch, F., and Vlissides, J. (1996). Industrial experience with design patterns. In *Proceedings of the 18th International Conference on Software Engineering*, pages 103–114. IEEE Computer Society Press.

[Beck and Johnson, 1994] Beck, K. and Johnson, R. (1994). Patterns generate architectures. *Lecture Notes in Computer Science*, 821:139–149.

[Bencina and Burk, 2001] Bencina, R. and Burk, P. (2001). Port Audio: an Open Source Cross Platform Audio API. In *Proceedings of the 2001 International Computer Music Conference (ICMC '01)*. Computer Music Associaciation.

[Benveniste et al., 2003] Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. (2003). The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83.

[Bhattacharya and Bhattacharyya, 2001] Bhattacharya, B. and Bhattacharyya, S. (2001). Parameterized dataflow modeling for DSP systems. *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, 49(10):2408–2421.

[Booch, 1994] Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition edition.

[Borchers, 2000] Borchers, J. O. (2000). A pattern approach to interaction design. In *Symposium on Designing Interactive Systems*, pages 369–378.

[Boulanger et al., 2000] Boulanger, R. et al. (2000). *The Csound Book*. MIT press.

[Buck, 1993] Buck, J. (1993). *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California.

[Buck and Lee, 1994] Buck, J. and Lee, E. A. (1994). *Advanced Topics in Dataflow Computing and Multithreading*, chapter The Token Flow Model. IEEE Computer Society Press.

[Buck and Vaidyanathan, 2000] Buck, J. and Vaidyanathan, R. (2000). Heterogeneous modeling and simulation of embedded systems in El Greco. *International Conference on Hardware Software Codesign: Proceedings of the eighth international workshop on Hardware/software codesign*, 2000:142–146.

[Burbeck, 1987] Burbeck, S. (1987). Application programming in smalltalk-80: How to use model-view-controller (mvc). Technical report, Xerox PARC.

[Buschman et al., 1996a] Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996a). *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & Sons.

[Buschman et al., 1996b] Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996b). *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & Sons.

[Chaudhary et al., 1999] Chaudhary, A., Freed, A., and Wright, M. (1999). An Open Architecture for Real-Time Audio Processing Software. In *Proceedings of the Audio Engineering Society 107th Convention*.

[Cook and Scavone, 1999] Cook, P. and Scavone, G. (1999). The Synthesis Toolkik (STK). In *Proceedings of the 1999 International Computer Music Conference (ICMC99)*, Beijing, China. Computer Music Association.

[Coplien, 1998] Coplien, J. (1998). Software Design Patterns: Common Questions and Answers. *The Patterns Handbook: Techniques, Strategies, and Applications. Cambridge University Press, NY, January*, pages 311–320.

[Coplien and Schmidt, 1995] Coplien, J. and Schmidt, D. (1995). *Pattern languages of program design*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA.

[Dabney and Harman, 2001] Dabney, J. and Harman, T. (2001). Mastering SIMULINK 4.

[Dannenberg, 2004] Dannenberg, R. (2004). Combining visual and textual representations for flexible interactive audio signal processing. In *Proceedings of the 2004 International Computer Music Conferenc (ICMC'04)*. in press.

[Dannenberg and Brandt, 1996a] Dannenberg, R. B. and Brandt, E. (1996a). A Flexible Real-Time Software Synthesis System. In *Proceedings of the 1996 International Computer Music Conference (ICMC96)*, pages 270–273.

[Dannenberg and Brandt, 1996b] Dannenberg, R. B. and Brandt, E. (1996b). A Portable, High-Performance System for Interactive Audio Processing. In *Proceedings of the 1996 International Computer Music Conference (ICMC96)*, pages 270–273. International Computer Music Association.

[Davis et al., 2004] Davis, P., Letz, S., D., F., and Orlarey, Y. (2004). Jack Audio Server: MacOSX port and multi-processor version. In *Proceedings of the first Sound and Music Computing conference - SMC04*, pages 177–183.

[Dennis, 1974] Dennis, J. (1974). First version of a data flow procedure language, Programming Symposium. *Proceedings Colloque sur la Programmation*, pages 362–376.

[Douglass, 2003] Douglass, B. P. (2003). *Real-Time Design Patterns*. Addison-Wesley.

[Edwards, 1995] Edwards, S. (1995). Streams: a Pattern for "Pull-Driven. In Coplien, J. O. and Schmidt, D. C., editors, *Pattern Languages of Program Design*, volume vol.1, chapter 21. Addison-Wesley.

[Edwards et al., 2001] Edwards, S., Lavagno, L., Lee, E. A., and Sangiovanni-Vincentelli, A. (2001). Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Readings in Hardware/Software Co-Design*.

[Edwards and Tardieu, 2005] Edwards, S. and Tardieu, O. (2005). SHIM: a deterministic model for heterogeneous embedded systems. *Proceedings of the 5th ACM international conference on Embedded software*, pages 264–272.

[Eker et al., 2003] Eker, J., Janneck, J., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y. (2003). Taming heterogeneity-the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144.

[Fielding, 2000] Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

[Fong, 2000] Fong, C. (2000). Discrete-Time Dataflow Models for Visual Simulation in Ptolemy II. *Master's Report, Memorandum UCB/ERL M*, 1.

[Foote, 1988] Foote, B. (1988). Designing to Facilitate Change With Object Oriented Frameworks. Master's thesis, University of Illinois at Urbana Champaign.

[Fowler et al., 1999] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Gambier, 2004] Gambier, A. (2004). Real-time Control Systems: A Tutorial. *Control Conference, 2004. 5th Asian*, 2.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Gao et al., 1992] Gao, G., Govindarajan, R., and Panangaden, P. (1992). Well-behaved dataflow programs for DSP computation. *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, 5.

[Garcia and Amatrian, 2001] Garcia, D. and Amatrian, X. (2001). XML as a means of control for audio processing, synthesis and analysis. In *Proceedings of the MOSART Workshop on Current Research Directions in Computer Music*, Barcelona, Spain.

[Gasser and Widmer, 2008] Gasser, M. and Widmer, G. (2008). Streamcatcher: Integrated Visualization of Music Clips and Online Audio Streams. In *ISMIR 08: Proceedings of the 9th International Conference on Music Information Retrieval*.

[Geilen and Basten, 2003] Geilen, M. and Basten, T. (2003). Requirements on the execution of kahn process networks. In *Proceedings of the 12th European Symposium on Programming, ESOP*.

[Gerzon, 1973] Gerzon, M. A. (1973). Periphony: With-height sound reproduction. *Journal of the Audio Engineering Society*, 21:2–10.

[Girault et al., 1999] Girault, A., Lee, B., and Lee, E. A. (1999). Hierarchical finite state machines with multiple concurrency models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(6):742–760.

[Gordon and Talley, 1999] Gordon, R. and Talley, S. (1999). *Essential JMF: Java media framework*. Prentice Hall PTR.

[Graham, 1991] Graham, I. (1991). *Object Oriented Methods*. Addison-Wesley.

[Gray, 2003] Gray, K. (2003). *Microsoft DirectX 9 programmable graphics pipeline*. Microsoft Press.

[Green and Petre, 1996] Green, T. R. G. and Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174.

[Haas, 2001] Haas, J. (2001). SALTO - A Spectral Domain Saxophone Synthesizer. In *Proceedings of MOSART Workshop on Current Research Directions in Computer Music*, Barcelona, Spain.

[Hahsler, 2004] Hahsler, M. (2004). A quantitative study of the application of design patterns in java.

[Halbwachs, 1998] Halbwachs, N. (1998). Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16.

[Henzinger et al., 2001] Henzinger, T., Horowitz, B., and Kirsch, C. (2001). Giotto: A Time-Triggered Language for Embedded Programming. *Embedded Software: First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October 8-10, 2001: Proceedings*.

[Henzinger et al., 2003] Henzinger, T., Horowitz, B., and Kirsch, C. (2003). Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99.

[Hewit, 1977] Hewit, C. (1977). Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–363.

[Hewitt and Baker, 1977] Hewitt, C. and Baker, H. (1977). Actors and Continuous Functionals. *IFIP Working Conf. on Formal Description of Programming Concepts, August*.

[Hylands et al., 2003] Hylands, C., Lee, E., Liu, J., Liu, X., Neuendorffer, S., Xiong, Y., Zhao, Y., and Zheng, H. (2003). Overview of the Ptolemy Project. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berklee, California.

[Johnson and Jennings, 2001] Johnson, G. and Jennings, R. (2001). *LabVIEW Graphical Programming*. McGraw-Hill Professional.

[Judkins and Gill, 2000] Judkins, T. and Gill, C. (2000). A Pattern Language for Designing Digital Modular Synthesis Software.

[Kahn and MacQueen, 1977] Kahn, G. and MacQueen, D. (1977). Coroutines and Networks of Parallel Processes. *Information Processing 77, Proceedings of IFIP Congress*, 77(7):993–998.

[Kerihuel et al., 1994] Kerihuel, A., Mcconnell, R., and Rajopadhye, S. (1994). Vsdf: synchronous data flow for vlsi. In *Circuits and Systems, 1994., Proceedings of the 37th Midwest Symposium on*, volume 1, pages 389–392vol.1.

[Kohler, 1975] Kohler, W. (1975). A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems. *IEEE Transactions on Computers*, 24(12):1235–1238.

[Larman, 2002] Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice-Hall, second edition.

[Lawrence, 2003] Lawrence, J. (2003). Standards-Orthogonality of verilog data types and object kinds. *IEEE Design & Test of Computers*, 20(5):94–96.

[Lazzarini, 2001] Lazzarini, V. (2001). Sound Processing with the SndObj Library: An Overview. In *Proceedings of the 4th International Conference on Digital Audio Effects (DAFX '01)*.

[Ledeczi et al., 2001] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. (2001). The Generic Modeling Environment. *Workshop on Intelligent Signal Processing, Budapest, Hungary, May*, 17.

[Lee and Zhao, 2007] Lee, E. and Zhao, Y. (2007). Reinventing computing for real time. *Lecture Notes in Computer Science*, 4322:1.

[Lee, 1999] Lee, E. A. (1999). Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7(1):25–45.

[Lee, 2002] Lee, E. A. (2002). Embedded software. *Advances in Computers*, 56:56–97.

[Lee and Messerschmitt, 1987a] Lee, E. A. and Messerschmitt, D. G. (1987a). Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35.

[Lee and Messerschmitt, 1987b] Lee, E. A. and Messerschmitt, D. G. (1987b). Synchronous data flow. *Proc. of the IEEE.*, 75(9):1235–1245.

[Lee and Parks, 1995] Lee, E. A. and Parks, T. (1995). Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801.

[Liu and Layland, 1973] Liu, C. and Layland, J. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61.

[Liu, 2000] Liu, J. (2000). *Real-Time Systems*. Prentice Hall.

[Malham and Myatt, 1995] Malham, D. and Myatt, A. (1995). 3-D sound spatialization using ambisonic techniques. *Computer Music Journal*, 19(4):58–70.

[Manolescu, 1997] Manolescu, D. A. (1997). A Dataflow Pattern Language. In *Proceedings of the 4th Pattern Languages of Programming Conference*.

[McCartney, 2002] McCartney, J. (2002). Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68.

[Meunier, 1995] Meunier, R. (1995). The Pipes and Filter Architecture. In Coplien, J. O. and Schmidt, D. C., editors, *Pattern Languages of Program Design*, volume vol.1, chapter 22. Addison-Wesley.

[Mok, 1983] Mok, A. K. (1983). *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology Cambridge, MA, USA.

[Murata, 1989] Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77.

[Neuendorffer and Lee, 2004] Neuendorffer, S. and Lee, E. A. (2004). Hierarchical Reconfiguration of Dataflow Models. In *Conference on Formal Methods and Models for Codesign (MEMOCODE)*.

[Neuendorffer, 2005] Neuendorffer, S. A. (2005). *Actor-Oriented Metaprogramming*. PhD thesis, EECS Department, University of California, Berkeley.

[Orlarey et al., 2004] Orlarey, Y., Fober, D., and Letz, S. (2004). Syntactical and semantical aspects of Faust. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 8(9):623–632.

[Papadopoulos and Arbab, 1998] Papadopoulos, G. and Arbab, F. (1998). Coordination models and languages. *Advances in Computers*, 46(329-400):76.

[Parks, 1995] Parks, T. M. (1995). *Bounded Schedule of Process Networks*. PhD thesis, University of California at Berkeley.

[Pastrnak et al., 2004] Pastrnak, M., Poplavko, P., de With, P., and Farin, D. (2004). Data-flow timing models of dynamic multimedia applications for multiprocessor systems. In *System-on-Chip for Real-Time Applications, 2004.Proceedings. 4th IEEE International Workshop on*, pages 206–209.

[Perry, 1993] Perry, D. (1993). VHDL. *Mcgraw-Hill Series On Computer Engineering*, page 390.

[Pope and Ramakrishnan, 2003] Pope, S. T. and Ramakrishnan, C. (2003). The Create Signal Library ("Sizzle"): Design, Issues and Applications. In *Proceedings of the 2003 International Computer Music Conference (ICMC '03)*.

[Posnak and M., 1996] Posnak, E. J. Lavander, R. G. and M., H. (1996). Adaptive pipeline: an object structural pattern for adaptive applications. In *The 3rd Pattern Languages of Programming conference*, Monticello, IL, USA.

[Prechelt et al., 1998] Prechelt, L., Unger, B., Philippsen, M., and Tichy, W. (1998). Two controlled experiments assessing the usefulness of design pattern information during program maintenance.

[Puckette, 1991] Puckette, M. (1991). Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*.

[Puckette, 1997] Puckette, M. (1997). Pure Data. In *Proceedings of the 1997 International Music Conference (ICMC '97)*, pages 224–227. Computer Music Association.

[Puckette, 2002] Puckette, M. (2002). Max at Seventeen. *Computer Music Journal*, 26(4):31–43.

[Roberts and Johnson, 1996] Roberts, D. and Johnson, R. (1996). Evolve Frameworks into Domain-Specific Languages. In *Procedings of the 3rd International Conference on Pattern Languages for Programming*, Monticelli, IL, USA.

[Seen et al., 2000] Seen, M., Taylor, P., and Dick, M. (2000). Applying a crystal ball to design pattern adoption. *tools*, 00:443.

[Shaw, 1996] Shaw, M. (1996). Some Patterns for Software Architecture. In Vlissides, J. M., Coplien, J. O., and Kerth, N. L., editors, *Pattern Languages of Program Design*, volume vol.2, chapter 16. Addison-Wesley.

[Stallings, 1998] Stallings, W. (1998). *Operating systems: internals and design principles*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

[Stankovic, 1988] Stankovic, J. (1988). Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *Computer*, 21(10):10–19.

[Steinmetz, 1995] Steinmetz, R. (1995). Analyzing the multimedia operating system. *IEEE MultiMedia*, 2(1):68–84.

[Steinmetz, 1996] Steinmetz, R. (1996). Human perception of jitter and media synchronization. *Selected Areas in Communications, IEEE Journal on*, 14(1):61–72.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Software*. ACM/Addison-Wesley.

[Taymans et al., 2008] Taymans, W., Baker, S., Wingo, A., S., B. R., and Kost, S. (2008). GStreamer application development manual 0.10.19.

[Torger and Farina, 2001] Torger, A. and Farina, A. (2001). Real-time partitioned convolution for Ambiophonics surround sound. *Applications of Signal Processing to Audio and Acoustics, 2001 IEEE Workshop*, pages 195–198.

[Tzanetakis, 2008] Tzanetakis, G. (2008). *Intelligent Music Information Systems: Tools and Methodologies*, chapter Marsyas-0.2: a case study in implementing Music Information Retrieval Systems, pages 31–49.

[Tzanetakis and Cook, 2002] Tzanetakis, G. and Cook, P. (2002). *Audio Information Retrieval using Marsyas*. Kluewe Academic Publisher.

[van Dijk et al., 2002] van Dijk, H. W., Sips, H. J., and Deprettere, E. F. (2002). On Context-aware Process Networks. In *Proceedings of the International Symposium on Mobile Multimedia & Applications (MMSA 2002))*.

[Vlissides, 1998] Vlissides, J. (1998). *Pattern Hatching, Desing Patterns Applied*. Addison-Wesley.

[Wadge and Ashcroft, 1985] Wadge, W. and Ashcroft, E. (1985). *LUCID, the dataflow programming language*. Academic Press Professional, Inc. San Diego, CA, USA.

[Walli, 1995] Walli, S. (1995). The POSIX family of standards. *StandardView*, 3(1):11.

[Wang and Cook, 2004] Wang, G. and Cook, P. (2004). ChucK: a programming language for on-the-fly, real-time audio synthesis and multimedia. *Proceedings of the 12th annual ACM international conference on Multimedia*, pages 812–815.

[Weinand et al., 1989] Weinand, A., Gamma, E., and Marty, R. (1989). Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2).

[Wilson, 1990] Wilson, D. A. (1990). *Programming With Macapp*. Addison-Wesley.

[Wright, 1998] Wright, M. (1998). Implementation and Performance Issues with Open Sound Control. In *Proceedings of the 1998 International Computer Music Conference (ICMC '98)*. Computer Music Association.

[www-CLAM,] www-CLAM. CLAM website: http://clam-project.org.

[www-PatternsEssential,] www-PatternsEssential. Pattern and Software: Essential Concepts and Terminology, http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

[Yu-Kwong, 1996] Yu-Kwong, I. (1996). Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors.