A Domain-Specific Metamodel for Multimedia Processing Systems

Xavier Amatriain

***Abstract*—In this paper, we introduce *4MPS*, a Metamodel for Multimedia Processing Systems. The goal of 4MPS is to offer a generic system metamodel that can be instantiated to describe any multimedia processing design. The metamodel combines the advantages of the object-oriented paradigm and metamodeling techniques with system engineering principles and graphical models of computation.**

4MPS is based on the classification of multimedia processing objects into two main categories: *Processing* objects that operate on data and controls, and *Data* objects that passively hold media content. Processing objects encapsulate a method or algorithm. They also include support for synchronous data processing and asynchronous event-driven *Controls* as well as a configuration mechanism and an explicit life cycle state model. Data input to and output from Processing objects is done through *Ports*. Data objects offer a homogeneous interface to media data, and support for metaobject-like facilities such as reflection and serialization.

The metamodel can be expressed in the language of graphical models of computation such as the *Dataflow Networks* and presents a comprehensive conceptual framework for media signal processing applications. 4MPS has its practical validation in several existing environments, including the author's CLAM framework.

***Index Terms*—Dataflow graphs, modeling, multimedia systems, object-oriented methods, systems engineering, visual languages.**

I. Introduction

IN THIS PAPER, we present a metamodel for designing multimedia processing software systems (i.e., multimedia systems that are designed to run preferably on software platforms and are signal processing intensive). Such systems share many constructs not only in the form of individual and independent design patterns but also at the overall system model level. This is even more so for application frameworks, which aim at supporting not just a single system but a set of conceptually related systems in the multimedia domain.

Furthermore, because of the particular characteristics of these kinds of systems, they are well suited for using graphical models of computation such as Dataflow Networks or Kahn Process Networks. But these models are not common practice for software engineers who are much more comfortable with the UML standard. And domain experts, usually focused on signal processing or knowledge discovery issues, usually know little both about graphical models of computation or UML.

Manuscript received October 3, 2006; revised April 19, 2007. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Lap-Pui Chau.

The author was with the Media Arts and Technology Program, University of California, Santa Barbara, CA 93111 USA. He is now with Telefonica I+D, Barcelona, Spain (e-mail: xar@tid.es).

Digital Object Identifier 10.1109/TMM.2007.902885

When designing CLAM (C++ Library for Audio and Music) we found out that although different frameworks for audio and music processing existed none seemed to fit our needs. Also, while most of them shared similar constructs there was no formal metamodel that could explain these relations or even offer a common vocabulary on which to base a new framework design. Similar solutions were even hard to compare because of a lack of common ground.

In this context, in this paper we propose a coherent metamodel that can be used to efficiently model any multimedia processing system and aims at offering a common high-level semantic framework for the domain. The metamodel uses the object-oriented paradigm and exploits the relation between this paradigm and the notion of graphical models of computation used in system engineering. Although the relation of the multimedia field to these concepts might seem obvious, a review of existing literature reveals that both object-orientation and graphical system modeling are often completely neglected from multimedia systems design.[1] In that sense we feel that a major contribution of the 4MPS metamodel is bringing these concepts to the spotlight in the context of multimedia systems.

Therefore 4MPS provides a common basis for the understanding of multimedia systems for experts with different background such as systems, software, or signal processing engineering and even multimedia artists. Because of this, the metamodel in general—and this paper in particular—aims at being understandable by a general audience by sometimes avoiding formal syntax from both systems and software engineering. Modeling a system using 4MPS takes a very basic knowledge in the previously mentioned areas. Nevertheless, and because of the way the metamodel is architectured, formal constructs are underlying any 4MPS model and can be accessed for further formal analysis from different viewpoints.

According to the 4MPS metamodel a generic multimedia processing system can be thoroughly and effectively described using simple object-oriented constructs. The metamodel offers a classification of objects in terms of their role in a multimedia processing system. Objects are classified into several main categories: processing objects, data containers, ports, and controls. 4MPS is closely related to Dataflow Process Networks, a graphical model of computation that has proven useful for modeling signal processing systems.

Therefore, the metamodel is on one hand related to pattern languages such as the one presented by Manolescu [5] or Arumi [6], but it adds modeling structure and architecture as well as formal background. On the other hand, it is related to system-level design models such as the one offered by the Ptolemy

[1]A study of comprehensive references (such as [1]–[4]) reveals the fact that none of them mention object-orientation or its benefits and there is only a single mention to Petri Nets but not to any other graphical MoC.

project [7], but it adds multimedia semantics. In that respect the main additions to traditional system-level models such as Kahn Process Networks or Dataflow Networks can be summarized in the following:[2]

- distinction between synchronous data flow and event-driven control flow;
- explicit life cycle for actors (Processing objects) and graphs (Networks);
- direct relation to object-orientation;
- availability of both dynamic and static composition;
- explicit representation of not only nodes and arcs but also data objects;
- clear domain-specific semantic for the elements in a processing graph and their internals.

The 4MPS metamodel can be instantiated by using a domain-specific visual language that will be presented throughout the paper. Nevertheless other approaches such as the use of XML or scripting languages for working with the metamodel are also possible and will be outlined.

The metamodel is basically the outcome of many years and iterations over multimedia, interactive, signal processing, real-time, and distributed systems, and its constructs are in fact architectural patterns found in several similar frameworks and environments. A high-level domain metamodel such as 4MPS is validated in a similar way to how a pattern language is: by identifying repeated and successful instances in the form of multimedia system models.

In Section II, we will start by defining some foundational concepts related to system modeling and graphical models of computation. Then we will complete our problem statement by defining the requirements of Multimedia Processing Systems. We will then, in the central part of the paper, define the 4MPS and its different components. In the next section we will experimentally validate the metamodel by presenting several implementations, success stories and evaluating its main dimensions. Finally, we will state our conclusions.

II. Background

A. Systems, Models, and Metamodels

The concept of *model* is closely related to that of *system*. As a matter of fact, we define a model as "an abstract representation of a given system." A system is "a large collection of interacting functional units that together achieve a defined purpose" [8]. It is made up of three main components: a goal, a set of things and/or rules, and the way this things and/or rules are organized or connected. The main goal when analyzing a system is to come up with a valid model for a given purpose. A model consciously focuses on some domain matters leaving others out.

A given system can be represented by many different models. These models may have different level of abstraction and purpose, and the "best" model is only the one that is the most useful for a particular application or purpose [9]. On the other hand a single model may have multiple interpretations, where the *interpretation* is defined as the relation of the model to the thing being modeled [10].

[2]As it will be later explained, many of these additions can be independently traced in different prior works, the novelty here being their grouping and their applicability to multimedia systems.

The current trend in software engineering of giving more value to models has given place to a methodology known as *model-driven development* (MDD) [11]. In traditional or code-driven development models are treated as simple sketches that are thrown away once the code is done, but in model-driven development the models themselves become the primary artifacts in the development process. The benefits reported for MDD are the same as those related to the usage of models in general, namely: enhanced productivity, portability, interoperability, and maintenance and documentation [12].

How does the object-oriented paradigm relate to models and systems? It turns out that one of the most commonly accepted definitions for "system" is surprisingly related to the software engineering corpus. In [9] the authors define: "A system is a set of objects together with relationships between the objects and between their attributes." This definition, that is largely referenced and commented in the literature about system theory, was made in 1956, much before the term object orientation was even coined.

When modeling a set of related systems belonging to the same domain, we realize that these models share many constructs. We are then able to generalize across these different models and come up with a model of what the set of related models should conform to. This is what we call a *metamodel*, a model of models. In the context of the software engineering community, the concept has been mostly used in relation to the Unified Modeling Language (UML) [13] and the Meta Object Facility (MOF) [14]. The UML language itself is used to describe the syntactic rules that all models written in UML must adopt thus defining a metamodel: a model of models.

Metamodeling is also sometimes understood as the definition of a semantic model for a family of related domain models. In this sense metamodeling is related to ontological engineering [15]. The properties of a well designed ontology are in fact the same as those of any software system including classes in an object-oriented design [16].

But while coming up with a formal and complete ontology or full-fledged metamodel is many times an unnecessary overhead, the software engineering community is accepting the practical benefits of metamodeling. In this context, many practitioners are praising the advantages of using *domain-specific* languages [17], that is modeling languages that are created to facilitate modeling within a particular domain.

But whatever our approach is, it is clear that we usually want to reuse our abstraction or modeling effort and try to come up with a model that is not only useful for the particular problem at hand ("system under study") but can be reused in similar situations, i.e., a metamodel.

B. Graphical Models of Computation

Models of computation (MoCs) are abstract representations of a family of related computer-based systems. Selecting the appropriate model of computation depends on the purpose and on the application domain; DSP applications, for instance, will generally benefit from dataflow models while control-intensive application mostly use finite state machine or similar models. A single paradigm, such as object-orientation or functional programming, may be used to model different models of computation and a single model of computation may be appropriate for different paradigms.

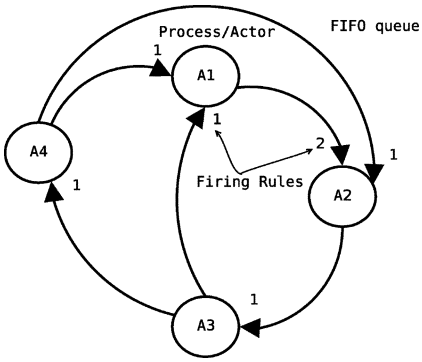

Fig. 1. Dataflow Process Network is a particular Graphical Model of Computation where nodes are *actors* that respond to *firing rules*. In this particular example A2 needs to receive 2 tokens from A1 and 1 from A4 in order to fire.

For our purposes most useful models of computation belong to the category of "graphical MoCs." By *graphical* we are expressing the fact that the system can be explicitly modeled by a graph, a general mathematical construct formed by "arcs" and "nodes." MoCs are described by assigning a concrete semantic to arcs and nodes and by restricting the general structure of the graph (see Fig. 1 representing a Dataflow Process Network). What follows is a very brief description of the most important graphical MoCs in the context of this paper.[3]

Process Networks or **Kahn Process Networks** [20], [21] is a concurrent model of computation that was originally developed for distributed systems but has proven its convenience for signal processing systems. Process Networks are directed graphs where nodes represent *Processes* and arcs are infinite FIFO queues that connect these processes. Writing to a channel is non blocking (it always succeeds immediately) but reading is blocking. If a process tries to read from an empty input it is suspended until it has enough input data and the execution context is switched to another process or level.

Dataflow Networks [22] is a special case of Process Networks. In this model arcs also represent FIFO queues. But now the nodes of the graph represent *actors* (see Fig. 1). Instead of responding to a simple blocking-read semantics, actors use *firing rules* that specify how many tokens must be available on every input for the actor to fire. When an actor fires it consumes a finite number of tokens and produces also a finite number of output tokens.

Synchronous Dataflow Networks (SDF) is a special case of Dataflow Networks in which the number of tokens consumed and produced by an actor is known before the execution begins.

III. Multimedia Systems

Before we continue it is important to establish a clear definition of the problem statement. In order to do so we should analyze the domain and identify what are the main requirements of a generic Multimedia System.

Although any system containing text, graphics, video, audio and music—or any combination of the previous elements—can be considered a Multimedia system we will restrict our domain to the subset of Multimedia Processing Systems. This means that these systems will be signal processing intensive—leaving out, for instance, static multimedia authoring systems.

hese are the most important features of any Multimedia Processing System (MMPS).

- *Data and Process separation*: As Rao *et al.* point out [1] multimedia consists of "multimedia data" and a "set of instructions." In a MMPS it is important to be able to separate both concerns.
- *Stream oriented*: multimedia processing systems work on streams of data. These streams are made of atomic elements called *data tokens*. Most streams represent time varying signals so tokens have a more or less determinate time stamp. This means that a generic data flow has to be processed in a synchronous manner.[4]
- *Multiple data*: by definition in a typical system we will have multiple data types and streams that will be processed, transformed, and composed with.
- *Interaction*: one important feature of most multimedia systems is that they are more or less interactive. By this we mean that the system takes into account the user input in order to control or modify the execution flow. This means that, apart from the multiple data streams, we also have a separate event-driven control flow and this control flow may somehow influence the data flow.
- *System Complexity*: even trivial MMPS can yield complex models containing multiple data types, streams and control paths. There is a clear need for encapsulation, detail hiding and composition at different levels.
- *Software Orientation*: as Mandal points out [2] "computer-based processing is almost a necessity" for any multimedia system, let alone those we have defined as being *processing* intensive.

IV. 4MPS Metamodel for Multimedia Processing Systems

This section, which represents the core of the paper, presents the Metamodel for Multimedia Processing Systems, 4MPS for short. The goal of the metamodel is, beyond the fact of being formally correct, being accessible for the broad multimedia community. Because of this we have decided to use UML sparsely. Note though, that using the completely redesigned Activity Diagrams in UML 2.0 [23] and some multimedia extensions that have been introduced to the basic profile [24], 4MPS could indeed be described in terms of a UML profile or even better in terms of a MOF metamodel not to be constrained to deriving it from UML.[5]

4MPS provides the conceptual framework (i.e., metamodel) for a hierarchy of models of media data processing system architectures in an effective and generic way. The metamodel is an abstraction of many ideas developed in implementing the CLAM framework, studying related solutions, and discussing

[3]See [7], [18] or a general purpose Systems Engineering reference such as [19] for a more comprehensive review of different graphical MoCs.

[4]As Mandal [2] and Rao *et al.* [1] point out, a generic multimedia system combines both continuous and discrete data. Nevertheless, we consider the existence of nontemporal data irrelevant in the context of multimedia systems that are processing intensive.

[5]As the specification itself points out [13] "There is no simple answer for when you should create a new metamodel and when you instead should create a new profile."

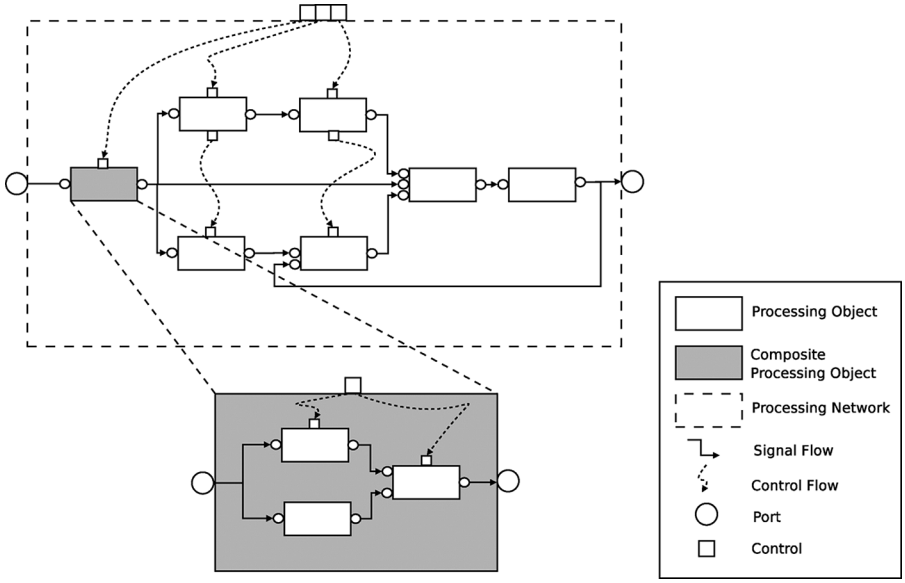

Fig. 2. Basic elements in 4MPS: Processing objects, represented by boxes, are connected through ports, round connection slots, defining a synchronous data flow from left to right. They are also connected through controls, square slots, defining in this case a vertical event-driven mechanism. Processing objects can be composed statically into Processing Composite objects or dynamically into Networks.

models with other authors. As a result of this process these abstractions do in fact reflect the commonalities of a family of related frameworks: those in the multimedia processing domain.

Using OMG metamodeling terminology [12] most of the following discussion related to 4MPS focuses on level M2 (metamodel). On the other hand M1 (models) are all the particular system models that use 4MPS (see Fig. 8, for instance) and M0 objects are the resulting multimedia applications or systems and the way they use particular run-time instances of the metamodel.

In the previous section we presented the separation of data and processes as being one of the most important requirements in Multimedia Processing Systems (MMPS). For that reason, 4MPS classifies objects into two main categories: objects that perform processing (*Processing Objects*), and objects that hold data used by the process (*Data Objects*).[6] Each of these categories forms a metaclass of the 4MPS Metamodel.

A 4MPS-derived model will be set in terms of Processing objects deployed as an interconnected Network where each Processing object can retrieve Data tokens and modify them according to some algorithm (see Fig. 3). A Network is on the one hand the grouping of a set of Processing objects with a common goal and on the other the logical entity in charge of scheduling and triggering the different Processing objects.

A. Processing Objects

The 4MPS Processing metaclass is the abstract encapsulation of a process following the object-oriented paradigm. We call any instance of a Processing subclass a Processing object.[7]

[6]Other secondary categories such as connecting and interfacing objects, and application or system-level objects also exist in the metamodel and will be reviewed later.

[7]Apart from the obvious relation to CLAM's Processing object, the abstraction is directly related to concepts present in many other frameworks such as Marsyas' *transformations*, OSW's and Kyma's *transforms*, *objects* in Max and *sound objects* in SndObj, *unit generators* in CSL and Aura, STK's *instruments*, *processes* in FORMES or *virtual processors* in VSDP [25].

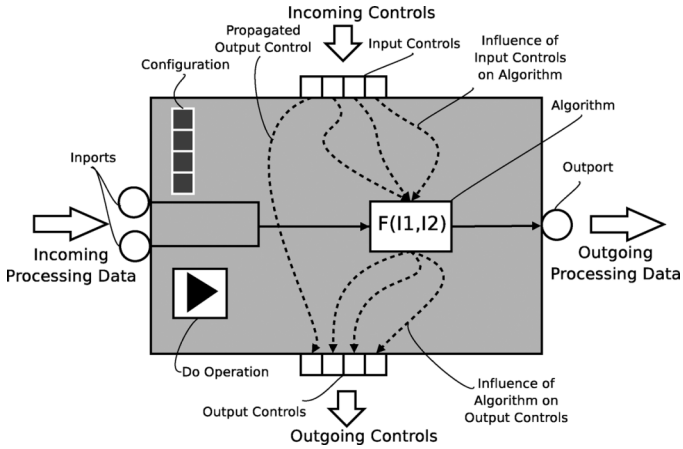

Fig. 3. Representation of the 4MPS Processing Metaclass. Data traveling from left to right enters through inports, is processed, and leaves through outports. Incoming controls can influence the process and/or can be forwarded through output controls. The Processing metaclass also has a configuration and a way to access its functionality through the Do operation.

The Processing objects are the main building blocks of a 4MPS modeled system. All processing in a 4MPS model must be performed inside a Processing object. Fig. 3 illustrates the different concepts that are encapsulated in the Processing metaclass. Its main components are a *configuration*, incoming and outgoing data *ports*, incoming and outgoing *controls* and any number of internal *algorithms*.

A Processing object is subject to two different flows: represented from left to right the *data flow* and from top to bottom the *control flow*. The data flow is synchronous and thus controlled by an external clock while the control flow is asynchronous and event-driven. This distinction is necessary in an MMPS as we need to distinguish between stream-oriented data processing and control events that do not occur at a particular rate. The

distinction between Data and Control is directly related to the *In-band Out-of-band Partitions* pattern [5].

When triggering the process we are asking the Processing object to access some incoming data and, using the encapsulated algorithm(s), transform it to some output data. A Processing object is able to access external data through its connection Ports. Input ports access incoming data and Output ports send outgoing data.[8] Pairs of ports can be connected defining a communication channel between Processing objects.

Besides receiving and sending data at a fixed rate through its Ports, a Processing object may also receive asynchronous Control events. These events affect the Processing object internal state and therefore are able to influence the result of the process itself. The Processing object can also broadcast events through its output Controls.

But all these mechanisms can be seen as auxiliary to the processing object main functionality, which is that of encapsulating one or more particular algorithms that work towards a clearly defined purpose. These algorithms are the ones actually in charge of processing the input data. The selection of one of the maybe alternative algorithms available is usually done upon configuration though some particular Processing objects may be able to implement a Strategy pattern [26] for dynamically selecting one algorithm or the other.

The execution of the Processing functionality is triggered by sending it a "Do" message. On the other hand, dynamic changes (and by dynamic we mean those that can be applied without the object having to transition from one of the main states to the other) will be triggered by the acknowledgement of an input control. The response of a Processing object to a Do message depends on the values of the data in the input ports but also on the internal state, which depends on the input controls received.

Processing Object Lifecycle: By analyzing the different messages that a Processing object receives and the way they influence its internal state we realize that Processing objects have an explicit and well-defined lifecycle. This same lifecycle, with minimal variations, can be found in many multimedia frameworks and applications.

The Processing object lifecycle is made up of the following main states: *Unconfigured*, *Ready*, and *Running* (see Fig. 4). While in the *Unconfigured* state the Processing object is waiting to be configured; in the *Ready* state it can be reconfigured or started; in the *Running* state, once the Processing object has been configured and started, the actual process can be executed; finally if the process needs to be suspended, in order to reconfigure for instance, the Processing object can be stopped. In any of those states the call to an unsupported message—for instance, Configure when in Running state—leads to an error and back to the Unconfigured state.

The messages that can be sent to a Processing object in order to change its state are: *Configure*, *Start*, *Do* and *Stop*.

Processing Configuration: A Processing Configuration is an object that contains values for a Processing object's non-execution variables, that is all the variable structural attributes that can only be changed when the Processing object is not in the Running state. The Configuration is used for setting the initial state of the Processing object. It may contain attributes related to any structural characteristic like the algorithms or strategy to be selected and values for structural parameters, such as the expected size of the incoming data, that can be used to also initialize the algorithms or internal tables. A Configuration though may also have initial values for non-structural attributes such as the controls.[9]

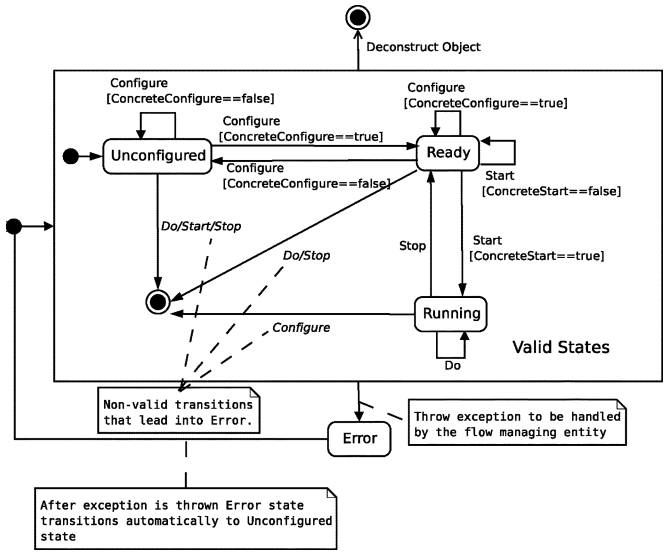

Fig. 4. Processing state diagram. A 4MPS Processing object has three well-defined states: unconfigured, ready and running. A Processing object has to first be correctly configured, and then started. It can only process when in the running state.

Data and Control Flow: While a Processing object is in the Running state it will receive, process, and produce two different kinds of information.

- Synchronous data: will be fed from and to the Ports every time a Do method is called. Processing objects consume Data through their input Ports and produce Data to their output Ports.
- Asynchronous data: will be fed from and to the Controls whenever a control event happens. They usually change the internal state/mode of the algorithm

Processing objects consume Data through their input Ports, process it and leave the result in their output Ports. The Data is consumed and produced in response to a call to the Do operation.

It is important to note that in 4MPS a data token is the atomic partition of any data that refers to an instant in time. Therefore, a whole spectrum, whatever its size, is a data token. But a chunk of audio is not considered a token as its data spreads over time, in this case each individual audio sample is a token in itself.

Processing objects, though, do not consume a unique, not even fixed, number of data tokens. Each Processing object may configure the size of the data chunk needed for a single execution. Also there is no forced relation between the region sizes of connected ports. An Outport many have a very small region while the connected Inports have larger reading regions. In this case the producing Processing object will have to be triggered several times before the reading Processing objects can proceed with an execution.

[8]In some frameworks, input ports are called *inlets* and output ports *outlets*.

[9]In Ptolemy [7] for instance, configuration related data is known as *parameter arguments*, while flow related data and control is known as *stream arguments*.

Ports in a Processing object may be also understood as intelligent pointers to where data is located (usually a memory location). If both the outport and one of the inputs are pointing to the same location, the Processing object is said to be processing *inplace*. Not all the Processing objects have the ability of processing inplace as this greatly depends on the algorithm that they encapsulate. Furthermore, input and output ports do not even have to be the same type: the transformation or process introduced by the processing object on the incoming data can be so structural that even the data kind may change (e.g., a Processing object may convert an input "tree" into and output "piece of furniture").

All interconnected Ports should be strongly typed and expect the same Data type. Although this condition may be somehow relaxed in some situations by allowing connections of polymorphic Data Ports, this is not common nor recommended. Actually, and as explained in , we want connections to be handled in a generic way and therefore introduce polymorphism at the graph level but we do not want to incur into performance penalty because of dynamic casting at run-time. This is solved by using the Typed Connections pattern and through the use of static polymorphism (e.g., C++ templates).

In the implementation layer such a structure is deployed through the Data Node, which is basically a phantom circular buffer with multiple reading/writing regions. Again see [6] for more details on this design pattern.

But, apart from the synchronous data flow, Processing objects can also respond to an asynchronous flow of events. This mechanism is encapsulated in the concept of Controls. A Processing object may have any number of input Controls and output Controls. An input Control affects the non-structural aspect of the Processing object state. This means that the reception of an input Control does not produce a transition from one of the three main states to another. On the other hand, output Controls maybe generated at any time although it is usual for them to be the result of one of the two following cases: (1) a response to a received input Control or (2) a result of a particular algorithm that apart from (or instead of) producing output data also generates a number of asynchronous events.

A control event is transmitted from the output control to the connected input controls as soon as it is generated. The new control value overwrites the previously existing one even if it has still not been read.[10] Controls must also have a clear initial value, which in many cases may be set by a configuration parameter. The initialization of the control value is performed in the Start transition so that every time that the Stop/Start cycle is followed the control is able to return to its initial value.

Controls should be simple datatypes such as integers, floats or boolean. Any structure more complex than that should not be sent as a Control the reason being that controls are sent asynchronously and continuously whenever they generate or are modified. The control associated to a slider, for instance, can be transmitted several times before actually affecting the result of the process, which will happen next time that the Processing Do is called. It is a waste of resources to transmit complex structures with this mechanism. Nevertheless, it is easy to add dedicated Processing objects that convert from Control to Data and vice versa.

Kinds of Processing Classes: Generators, Sinks and Transforms: In the Kahn Process Network model of computation, two kinds of subgraphs are of special importance: *data sources* and *data sinks* [21]. In [5] the *Data Flow Architecture* pattern classifies modules into *sinks*, *filters*, and *sources*. In a similar way in 4MPS we identify *Generating* Processing objects, which are data sources, and *Sink* Processing objects, which only consume data. *Transforms* represent the general case of Processing objects that have both input and output ports.

B. Data Objects

All data in a 4MPS model is contained in *Data* objects, a concept that is related to the *Payloads* pattern explained in [5]. This is also sometimes referred to as a content or record object.

4MPS Processing objects can only process Data objects. To be fully usable in such a context, a Data class must offer a number of services, namely: introspection or the ability to respond about its own structure; homogeneous interface; encapsulation; persistence; composition; and display facilities. Note that many of these requirements are related to the services offered by MOF meta-objects [14].

Data and Value Attributes: Attributes in a Data class can be classified into *data attributes* and *value attributes*. Data attributes basically act as data containers and they are usually of complex types such as arrays.

Value attributes act as auxiliary information related to the data attributes. They are always of simple types such as integer or floating point numbers. Value attributes can in turn be divided into *informative value attributes* and *structural value attributes*. Informative value attributes are simple value containers that are used to interpret the Data content. A modification in such attributes does not imply a change in the related data. Conversely structural value attributes are meant to inform but also to modify the internal structure of Data objects.

For example, a *Spectrum* Data class could have the following attributes.

- *Data Attributes*: two buffers, one for the magnitude and another one for the phase.
- *Value Attributes*
  - — *Structural*: size, when this changes the buffers are resized.
  - — *Informative*: spectral range, a change in its value does not imply any structure change.[11]

C. Scalable Composition With 4MPS Objects: Networks and Processing Composites

The 4MPS metamodel offers different mechanisms for composing networks of Processing objects. In this section we will show their features and intent. Composition in 4MPS is not actually required; any model can be fully specified by a number of independent Processing objects. Nevertheless, building

[10]Although this is the default simple behavior for controls in 4MPS more elaborate mechanisms, such as control buffering, can be implemented at the receiving Processing Object if necessary. Note though that these buffering schemes will also require of an appropriate time-stamping mechanism for controls. This, although possible, is not necessary in the general case when controls are considered to have a coarse temporal grain. All these extensions will therefore not be addressed in this paper.

[11]Note though that even in this trivial example we could decide to treat this as a structural attribute and execute re-sampling algorithms when this is changed. Ultimately the developer has to decide on how to treat value attributes.

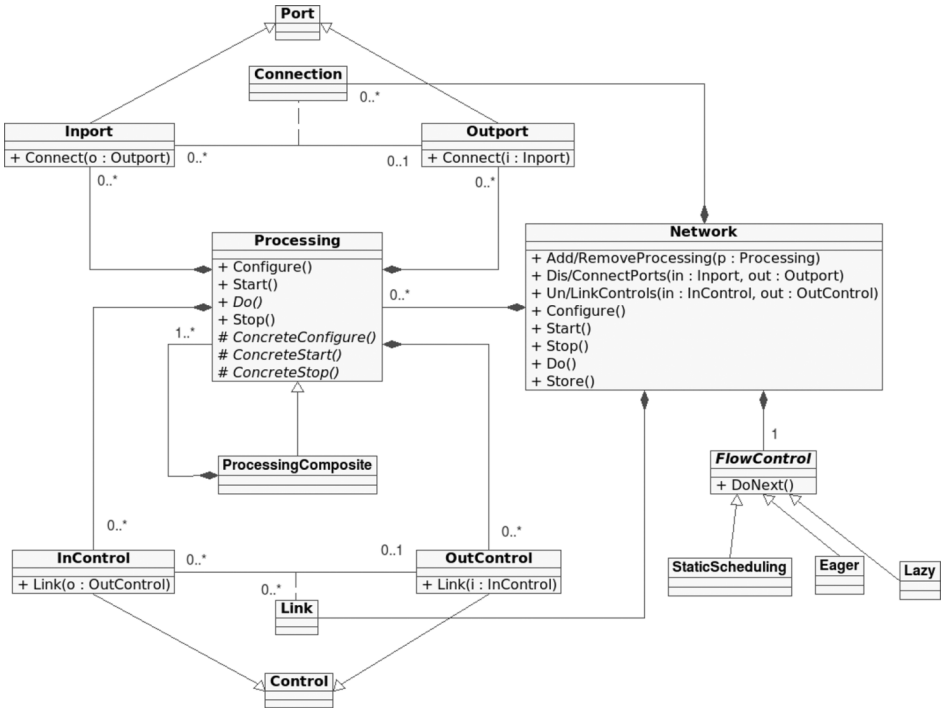

Fig. 5. This figure describes a (partial) metamodel of Network using a class diagram. This metamodel (as a language) is then instantiated when creating networks with the network editor (see Figure refcap: Network–Editor–Graphical). Note that a 4MPS Network is a dynamic run-time composition of Processing objects that contains not only Processing instances but also a list of connected Ports and Controls.

self-contained sub-models offers a number of advantages, namely 1) *reuse*; 2) *complexity and detail hiding*, sub-models can act as intermediate layers that hide complexity from the user; 3) *flow control automation*, by building coherent and homogeneous compositions we are able to apply standardized approaches to automatic flow control; and 4) *efficiency and optimization*, a composition can be a specialized grouping of Processing objects with a specific purpose and goal that can therefore be optimized.

The two mechanisms for composing with Processing objects in a 4MPS model are *Networks* and *Processing Composites*. Networks are dynamic compositions that can be modified at run time in any way by adding new Processing objects or modifying connections. Processing Composites are static compositions that are built at compile time and cannot be modified thereafter. These 4MPS constructs were already graphically illustrated in Fig. 2.

Note that both mechanisms address the first of the three benefits previously enumerated. But while Networks promote the ability to use automatic flow control (benefit number 2), Processing Composites strive for efficiency (benefit number 3).

In general terms, the dynamic composition offered by Networks follows the *Dataflow Architecture* pattern while the static composition in Processing Composites follows the *Adaptive Pipeline* pattern [5].

Processing Composites: A Processing Composite is a static composition of 4MPS Processing objects. A Processing Composite is manually implemented by a developer and might be fine-tuned for efficiency. The list of included Processing objects and their connections may not be modified at run-time.[12] It is a direct implementation of the Composite pattern [26].

A Processing Composite object has one parent that acts as the *composite* and any number of children that act as *components*. Any child can in turn be the parent of other Processing objects recursively defining different levels of composition. A Processing Composite object is seen as a regular Processing object from the outside, even in a Network-like context.

Networks: A 4MPS Network is a set of interconnected Processing objects that collaborate for a common goal and can be modified at run-time. As illustrated in the UML class diagram in Fig. 5, it can be seen as a set of Processing objects with connected input and output ports and input and output controls.

A Network has a dynamic list of Processing objects that can be updated at run-time. In order to define the process graph, the network must keep track of the list of connected pairs of input and output Ports. An output Port may have any number of input Ports connected to it while an input port can only be connected to a single output Port.

Apart from the process graph defined by interconnected Ports, a Network has another graph defined by Controls and their connections.

A Network has compositional properties so it can be made of interconnected Networks that in turn have internal Networks, etc.

[12]As a matter of fact we may use the control mechanism in order to change internal connections or bypass particular Processing objects in a Composite; this is rather a side effect of the flexibility of the metamodel rather than an important inherent feature.

D. 4MPS as a Graphical Model of Computation

4MPS offers an object-oriented metamodel for multimedia processing systems. Nevertheless, the consequence of applying object-oriented modeling techniques to the multimedia domain not surprisingly yields a Graphical Model of Computation. It is important to stress, though, that graphical MoCs in 4MPS are a consequence of the metamodeling process and not the opposite: 4MPS was not built as a semantic layer on top of preconceived graphical MoCs.

In order to identify what is the Graphical MoC that best suits our metamodel we will first summarize its main properties.

1) In the 4MPS graphical MoC the main nodes in the graph correspond to Processing objects. 2) By connecting Ports from different Processing objects we are defining the arcs in the graphical model. 3) These arcs in the graph are interpreted as theoretically unbounded FIFO queues[13] where data tokens are written and read. 4) Processing objects produce/consume from the FIFO queues in a synchronous manner, when told to do so by the flow control entity. 5) The data consumption/production rate of Processing objects is not fixed. 6) This consumption rate may, in some cases, vary at run time. 7) By connecting controls from different Processing objects we are also defining a secondary set of arcs; these Controls are transmitted using an event-driven mechanism as soon as they are generated.

The first four properties confirm the fact that 4MPS's graphical MoC is related to Process or Dataflow Networks. This comes to no surprise as those are the models that have been traditionally found most appropriate for signal processing related applications.

The fact that Processing objects can produce/consume different quantities of data tokens and that quantity is explicit leads us to observe that our model can be more easily assimilated to Dataflow Networks [22]. The "firing rules" of Dataflow Networks are translated into region sizes in the 4MPS models. The quantity of data tokens to be produced or consumed by a Processing object Port is specified by giving its region a particular size. This is precisely what Dataflow Networks firing rules usually specify.

Synchronous Dataflow Networks have firing rules that are statically defined so, in order to have regions re-sizeable at run-time like those found in 4MPS we have to turn to Dynamic Dataflow Networks. Note though, that 4MPS does not require all modeled applications to have dynamically re-sizeable regions so we will in many cases find 4MPS models conveying to the SDF MoC.

After the previous discussion we may conclude that 4MPS's graphical MoC is a "**(possibly Dynamic) Dataflow Network**."

In the general case the control mechanism in 4MPS will not introduce any difference in that respect as control events cannot modify "structural" aspects of the Processing Objects. Nevertheless, in some specific systems these control events may indeed need to interact on the firing rules therefore modifying the regular data flow. Although we will not favor these interactions, these models do deserve our special attention.

Several models exist for mixing control and data flow [27]–[29] the one that is best suited to our particular situation is that of "Context-aware Process Networks" [30]. As pointed out by the authors, adding asynchronous coordination introduces indeterminacy but this can be *oraclised* (isolated) into the control stream.

Scheduling: One of the advantages of having identified the underlying graphical model of computation is that we can then use related techniques to analyze the behavior of the metamodel. Of particular interest to us is the choice of an appropriate scheduling technique or algorithm.

Scheduling Process or Dataflow Networks is a nontrivial issue. It is beyond the scope of this paper to give an overview of such techniques and we refer the reader to the related literature [21], [31]–[33]. It is important to note though, that the goals of any practical scheduler are to offer a complete output and to achieve this with bounded memory (note that the boundness condition is imposed by implementation issues, not the theoretical model). Unfortunately, boundness and termination are undecidable in the general case.

As we said before, in the most simple case, a 4MPS model can be understood as a Synchronous Dataflow Network (SDF). This assimilation will stand valid as long as the ports do not change their window size in run-time and controls do not interfere with data flow or graph structure.

SDFs can in many cases be scheduled statically by building a periodic schedule. This method should of course be preferred whenever feasible as it will yield a more efficient system at run-time and speed up execution. The problem is that it is not always possible to construct such a schedule [31]. First, deadlocks have to be ruled out. Secondly the order of the so-called *topology matrix* has to be one less the number of actors in the graph. (See [33] for a complete explanation of how this matrix is defined and its different properties.) In those cases, the network can determine a complete cycle consisting of "f" firings for each Processing object.

If a static schedule can not be decided—this being probably the general case—we should resort to any run-time scheduling strategy. Different algorithms are found in the literature, broadly classified into eager (a.k.a demand-driven or pull) and lazy (a.k.a. data-driven or push). Using a *lazy* policy the execution thread starts by the outermost Processing object in the Network (the one whose output Port corresponds to the output of the Network). If this Processing object checks that it does not have enough input data it hands the control to the Processing object whose output Port is connected to its inputs and so on. In the *eager* version, the process starts with the Processing object that acts as the input to the chain.

If the network under study has indeed dynamic run-time changing firing rules we are then left no choice but to turn to dynamic on-line scheduling algorithms as the one found in [31].

E. 4MPS Networks, Run-Time Execution, and Domain-Specific Languages

Although so far our description has been mostly concerned with the structural properties of the metamodel components, the metamodel does also in fact offer an execution model. Our goal is for the metamodel to yield models and ultimately systems that are both efficient and flexible at run-time.

[13]Although the general Process Network graphical model uses unbounded FIFO queues it is well-known that it is usually possible to construct bounded memory implementations [22].

In previous sections we described a 4MPS Network as mostly a compositional tool. Nevertheless, this metaclass is much more important than that as a final 4MPS system can if fact be seen as a 4MPS Network—what we call the *Top Network*. Therefore, the run-time properties of the system will be those of a Network. To start with, and because as already explained a Network can in fact be seen as a Processing object, a Network also has an explicit three-state life cycle as the one illustrated in Fig. 4. In Fig. 5 we see that a Network also has the necessary operations to transition from one state to the next. When Configuring, Starting, or Stopping a Network ,we are in fact Configuring, Starting, and Stopping all the Processing objects in it.

Once in the Running state, the Network is in charge of calling the execution of the Do operation in all its Processing objects. In order to do so though, a particular run-time strategy has to be followed: either we have a static schedule that keeps a list of nodes to execute or we decide for any type of dynamic scheduling such as eager or lazy execution. As already explained the choice of the scheduling policy is not easy and depends on the topology of the network as well as the requirements of the system.

Because of this, responsibility over scheduling policy should be decoupled from regular run-time Network operations. In 4MPS (see Fig. 5) this responsibility is delegated to the FlowControl abstract metaclass that is in charge of keeping a pointer to the next Processing object that will be executed. Subclassing this metaclass allows for different flow control policies to be implemented offering a flexible yet efficient and ready-to-use run-time model.

Yet another important property of a 4MPS Network is its ability to be stored in a format that can later be processed and understood (see method *Store* in Fig. 5). Although the graphical representation of the metamodel is usually preferred, it sometimes is interesting for practical purposes to have a direct one-to-one mapping to a textual format. In the case of CLAM, XML was chosen as a general purpose storage format. Listing 1 (see Fig. 9) is an example of a text representation of a 4MPS Network.

Note that this Network description, which has a complete definition of an 4MS Network, can be considered as an XML-based domain-specific modeling language (DSML) in its own. This way we see that the metamodel is not coupled to a particular visual language but can in fact be instantiated by different DSMLs such as an XML dialect or a scripting language (see [34] for an example of how a scripting language can also be used to interact with a 4MPS based model). As a matter of fact, in the context of application frameworks such as CLAM, one can interact with the metamodel by directly manipulating classes in an OO language such as C++.

V. Metamodel Validation and Implementation

Evaluating a metamodel is not easy. A proper evaluation should include a combination of qualitative and quantitative studies. Although a thorough evaluation is beyond the scope of this paper, in this section we will outline the validity of the metamodel and the associated domain-specific modeling languages by using a combination of both approaches. And because in Software Engineering validation and implementation go hand in hand [35] we will use this same section to explain some of the metamodel realizations.

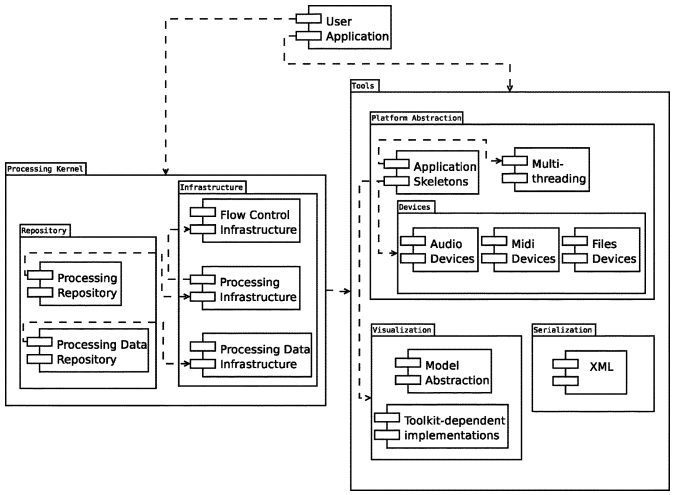

Fig. 6. CLAM components. A CLAM application makes use of both the CLAM Processing Kernel and its Tools. The Processing Kernel includes an Infrastructure, which is basically an implementation of the 4MPS, and a Repository of black-box ready-to-use algorithms and data. The Tools package contains components for services such as XML serialization, or cross-platform input/output.

We will first explain the realization of the metamodel in CLAM and a number of other frameworks. Then we will compare the proposed metamodel to other existing and related metamodels. Finally we will formally assess the validity of the framework by combining dimensions derived from the general Software Engineering corpus [35] and others borrowed from the Cognitive Dimensions framework [36], which has been specifically designed for evaluating visual languages.

A. CLAM

CLAM (C++ Library for Audio and Music) is an award-winning[14] framework that aims at offering extensible, generic and efficient design and implementation solutions for developing Audio and Music applications both for research an end-users. CLAM is both the origin and the main proof of concept of 4MPS. In the next paragraphs we will give a very brief presentation of the framework, please see [37] or [38] for more details.

CLAM offers a processing kernel that includes an *infrastructure* and Processing and Data *repositories* (see Fig. 6). In that sense CLAM is both a *black-box* and a *white-box* framework [39]. It is black-box because the already built-in and ready-to-use components included in the repositories can be connected with minimum programmer effort in order to build new applications. And it is *white-box* because the metaclasses that make up the infrastructure and are a direct implementation of the 4MPS metamodel can be easily refined to extend the framework components with new Processing or Data classes.

CLAM is *comprehensive* since it not only includes classes for processing but also for audio and MIDI input/output, XML serialization services, algorithm and data visualization and interaction, and multithreading handling. The framework deals with a wide variety of *extensible data types* that range from low-level signals (such as audio or spectrum) to higher-level semantic structures (such as musical phrase or segment).

[14]CLAM received the ACM 2006 award for the Best Open Source Multimedia Software.

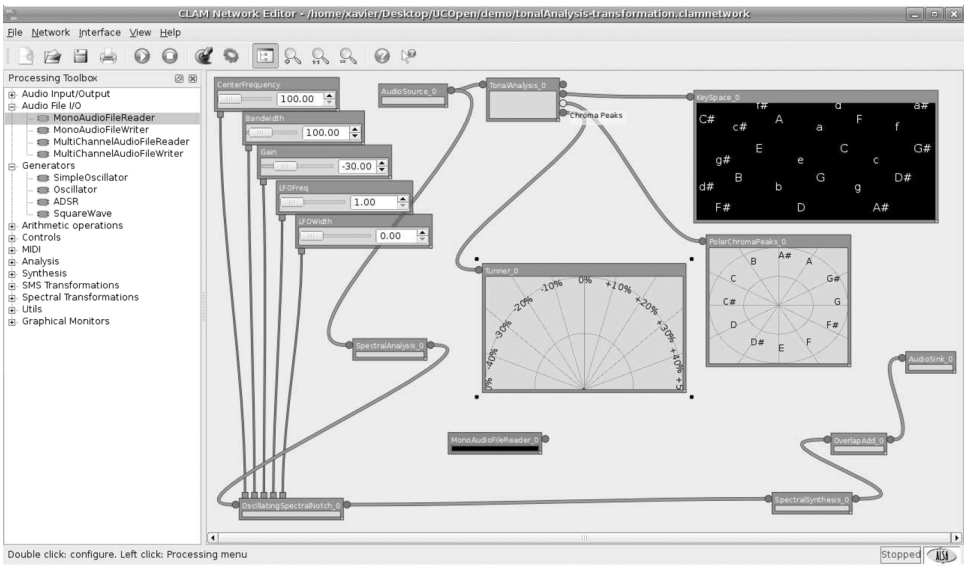

Fig. 7. Network Editor Graphical Interface. The Network Editor is CLAM's quintessential application. It is in fact a graphic representation of the 4MPS metamodel. The application allows to create new Processing Networks from pre-existing Processing objects and can be used as a rapid prototyping environment. In this example we have configured a network in which the incoming audio is analyzed in order to display its tonality content on one branch (three different visualization widgets are used on the upper right part of the network), and transformed with a frequency domain filter on another branch. Note how ports are represented by circles and controls by rectangles, both attached to Processing objects. Five sliders are used to control different aspects of the filter. The life cycle of the Processing objects is represented explicitly such as in the non connected one in the center of the network where the color represents that the object is *unconfigured*. Note that the top center Processing object has a different colored port. This is simply a visual cue that highlights a selected element while also offering a tooltip with the element name.

It is *cross-platform* as all the code is ANSI C++ and it is regularly compiled under Linux, Windows and Mac OSX using the most commonly used compilers. The framework can be used either as a regular C++ *class library* or as a *prototyping tool*. In the first mode, the user can extend, adapt or optimize the framework functionality in order to implement a particular application. In the second mode, the user can easily build a prototype application in order to test a new algorithm or processing application [40].

Success Stories: CLAM responded to a need for having a structured repository of signal processing tools focused on audio and music. For that reason, it has been used as an internal development framework since its very beginning.

Thus, CLAM applications have been developed and have been used as benchmarks to test the feasibility of the framework, and therefore the metamodel, under very different requirements. CLAM success stories are in fact 4MPS' success stories and partially validate the metamodel. In the following paragraphs we will review some of them (please refer to CLAM's website at www.clam.iua.upf.edu for further up-to-date information on the framework and its applications):

The *Network Editor* (see Fig. 7) is a graphical application for editing a `CLAM::Network` object. It can be considered in many ways as a 4MPS Visual building tool as the metamodel's concepts are indeed directly mapped into the tool. The output of the editor is an XML description of a CLAM 4MPS-compliant network. This description can be loaded from the *Prototyper*—another CLAM tool that can load qtDesigner visual designs and bind them with a processing network—in order to create a stand-alone application without having to write a single line of code (see [41] for more details).

The NetworkEditor has already been used as a tool that enables easy communication with domain experts with no software experience therefore proving the usefulness of the metamodel in these circumstances.

SMSTools is a CLAM application used to analyze, transform and re-synthesize sound in the spectral domain [42]. Apart from the interest in the application itself it is important to note that this is a direct port of an application that existed and was being used before the framework was born and the metamodel formulated. As a matter of fact, the development of the framework was driven by three applications that already existed and were ported through the process. Those applications included very different domain requirements (such as run-time efficiency or complex visualization) and therefore proved the validity of the metamodel in practical scenarios. Also porting these applications to a common framework and metamodel proved that they could be optimized more efficiently and the time to transfer knowledge to new developers was drastically reduced.

CLAM includes other applications such as *SALTO*, a software based brass synthesizer; *Spectral Delay*, a simple application that separates the input signal into three spectral bands each of which is then delayed separately (see the block diagram of the process in Fig. 8; or the *Annotator*, a tool for editing and annotating audio material [43]. Furthermore CLAM can be used to create plug-ins in several different formats such as VST or LADSPA.

B. Other Known Uses

The CREATE Signal Library (CSL, pron. "sizzle") [44] is a C++ class library for signal synthesis and processing. CSL's design is an instance of the 4MPS metamodel. CSL has evolved through several major re-writes since 1998 and although it was not originally designed having the 4MPS metamodel in mind, the latest releases have been influenced by it. As a matter of fact,

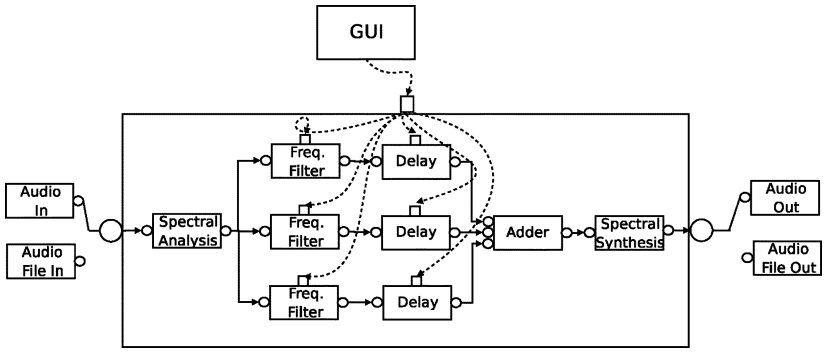

Fig. 8. Spectral Delay block diagram. Using the 4MPS we can easily model an application such as CLAM's Spectral Delay. In it, the input sound is divided into three bands and each band is then delayed independently.

release 4 of the framework is a complete rewrite that follows the 4MPS metamodel [45].

The kernel of CSL is a group of metaclasses that map directly onto the 4MPS metamodel of Processing, Data, and control I/O Ports. Referring back to Fig. 2, applications consist of networks of CSL processing objects interconnected via control and signal flows. The impact of the metamodel entails not just the class hierarchies, but also the object life-cycles and signal processing network composite models.

Another related framework that has been heavily influenced by the metamodel is Marsyas [46], an environment for developing music information retrieval applications.

Also the metamodel has recently been used to model a framework for audio spatialization [47]. And finally, to prove its applicability to the general multimedia domain and not only audio, it should be noted that the metamodel has been instrumental in the design of the artificial vision and hand recognition system described in [48].

C. Related Multimedia Frameworks and Metamodels

We have proven the usefulness of 4MPS in several cases of general purpose frameworks and applications. Nevertheless many of its constructs are found in many other similar frameworks in the multimedia domain.[15] The following is a list of some of the frameworks in multimedia, image processing and audio/music processing, that have been reviewed while designing the metamodel. Many of them are extensively reviewed in [18].

- Multimedia Processing Frameworks: Ptolemy [7], BCMT [49], MET++ [50], MFSM [51], VuSystem [52], Java I-HTSPN [53].
- Audio Processing Frameworks CLAM [40], CSL [44], Marsyas [54], STK [55], Open Sound World (OSW) [56], SndObj [57].
- Visual Processing Frameworks Khoros/Cantata [58](now VisiQuest), TiViPE [59], AVS [60], FSF [61].

[15]Although 4MPS was initially implemented in the Audio/Music domain, it is important to remember that the metamodel is not coupled to a particular data or processing style. In that sense, note that Audio and Music applications, such as the ones found in CLAM or CSL, are in general multimedia as they include audio signal data, symbolic data in the form of MIDI, scores or metadata, and graphics in the form of signal-related and symbolic dynamic presentations. Also, we are currently successfully applying the metamodel to the modeling of systems for image and html processing.

Most of these frameworks share some constructs with the 4MPS metamodel. The relation between some of these concepts (such as process orientation, separation of data and control, or static and dynamic composition of process networks) and 4MPS has been highlighted throughout the presentation of the Metamodel in Section IV. A thorough review of all such relations, though, is beyond the scope of this paper and we refer the interested reader to [18].

But when trying to compare these frameworks to the 4MPS metamodel at a higher level than that of the particular constructs (or patterns) we have to face the fact that, although every framework has an implicit metamodel, very few of them actually specify it even informally. In the following paragraphs we will briefly comment on those very few exceptions.

Francois' *Software Architecture for Immersipresence* (SAI) is related to his MFSM and FSF frameworks. It aims at offering "a software architecture model for designing, analyzing and implementing applications that perform distributed, asynchronous parallel processing of generic data streams" [51]. In many senses this metamodel shares many properties with 4MPS: SAI's *cells* are related to 4MPS' *Processing* objects while its *pulses* are our *Data*. SAI brings 4MPS implementation details such as the *Data Node* to the surface because of the need to distinguish between persistent and volatile data in a purely asynchronous model (this distinction is not necessary in 4MPS). Although this construct resembles Petri Nets the SAI metamodel is not related by the author to any formal system model. SAI is not formally related to object-orientation either. It also lacks many of the 4MPS highlights (different composition models, controls versus data, explicit life cycle and internal modeling of the graph elements...).

The Hierarchical Timed Stream Petri Nets model [53] is an extension of traditional Petri Nets related to the Java I-HTSPN framework. It is therefore highly coupled to this particular graphical MoC. The use of HTSPN does apply to control-intensive multimedia systems, especially those based on discrete time data, being particularly well suited for multimedia presentation and composition systems. However it is not suitable for the general case of Multimedia Processing Systems presented in this paper, which have not only timed interactions between data but also timed processes that modify data streams and dynamic interactions with those processes.

Other models such as OMMMA (object-oriented Modeling of Multimedia) [62] focus on modeling multimedia applications

from a traditional object-oriented perspective where the focus is put on data entities and processes are auxiliary properties of the data. OMMA delivers in the modeling of data entities and as such could be used for modeling in detail the Data entities in 4MPS. However, it falls into the shortcomings of UML already mentioned when it comes to modeling processes and flows with enough expressivity.

Finally, 4MPS shares some constructs with models and frameworks that are strictly DSP-oriented (see [63] for a comparison of many of them). Models such as the one presented in [64] are also built on the grounds of graphical models of computations such as Dataflow Networks for being this a natural model for signal processing systems.

In this context it is worth to mention the Ptolemy framework for probably being the best representative of these DSP software environments. Ptolemy focuses on heterogeneous system design [7]. As the authors explain, Ptolemy is "first and foremost a laboratory for experimenting with design techniques". In that sense, the framework is built on top of many different Models of Computation with the main goal being offering ways to interact with these models and combine them. While in 4MPS graphical MoC are a consequence of the design process and domain requirements in Ptolemy they are the origin of the framework itself. On the other hand, Ptolemy focuses mainly on embedded systems.

As opposed to 4MS Ptolemy and similar DSP-oriented frameworks do not offer any semantic layer related to a particular application domain because they aim at being "domain-independent." Ptolemy and 4MPS attack the problem of system design with different goals and from different perspectives, offering different layers to interact with. As a matter of fact, a 4MPS model could very well be built using the Ptolemy framework (note that Ptolemy has indeed been used for modeling image processing systems, for instance). But the framework itself, without any intermediate semantic layer, is hard to access by a Multimedia Engineer.

D. Evaluating the Metamodel

In order to finish our metamodel validation we will first briefly discuss the following high-level dimensions: *feasibility*, *completeness*, and *usability*. These dimensions are derived from the ones used in [65]. We will then review the different dimensions in the Cognitive Dimensions framework [36] and discuss their applicability to 4MPS.

Feasibility: This dimension refers to how practical are the abstractions in the metamodel and how well they fit the requirements in our particular domain. Is it feasible to implement efficient multimedia processing systems that are modeled using 4MPS? This should be the first question to address as any further discussion could be invalidated by a negative answer. In our case it has clearly been shown in the previous paragraphs that it is feasible to construct efficient working systems using 4MPS and that the metamodel can be implemented through frameworks in several ways and flavors.

Completeness: This dimension reflects two complementary questions. 1) Can *any* system in the domain be modeled with the metamodel that is proposed? 2) Can a system be completely modeled using and instance of the metamodel? In both cases answers are affirmative in respect to 4MPS. As shown in this section multimedia processing systems from the widest possible range have been modeled using 4MPS: from real-time audio effects to image recognition and hand-tracking algorithms; from off-line applications for content-extraction and metadata generation to prototyping environments. No application in the domain (even some previously existing that have been ported) has had to artificially adapt itself to the metamodel.

It is also important to note that there are tools that can help the user in dealing with the different levels of the metamodel, the model, and the final system. As presented in [41] CLAM offers tools for visually patching a 4MPS model and create a final standalone system without even writing a line of code.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<network id="ExampleNetwork">
  <processing id="Reader_0" type="MonoAudioFileReader">
    <SourceFile>
      <URI>/home/xavier/0001.wav</URI>
      <Header>
        <SampleRate>48000</SampleRate>
        <Samples>1101824</Samples>
        <Channels>2</Channels>
        <Length>22954.7</Length>
        <Format>WAV</Format>
        <Encoding>signed 16-bit</Encoding>
        <Endianess>Format Default</Endianess>
      </Header>
      <TextualDescriptors/>
    </SourceFile>
    <SelectedChannel>0</SelectedChannel>
    <Loop>1</Loop>
  </processing>
  <processing id="AudioOut_0" type="AudioOut">
    <Device>default:default</Device>
    <ChannelID>0</ChannelID>
    <FrameSize>512</FrameSize>
    <SampleRate>44100</SampleRate>
  </processing>
  <processing id="Sender_0" type="OutControlSender">
    <Min>0</Min>
    <Default>0</Default>
    <Max>1</Max>
    <Step>0.01</Step>
    <Representation>VSlider</Representation>
  </processing>
  <processing id="Vumeter_0" type="Vumeter"/>
  <port_connection>
    <out>Reader_0.Samples</out>
    <in>Vumeter_0.Input</in>
  </port_connection>
  <port_connection>
    <out>Reader_0.Samples</out>
    <in>AudioOut_0.Audio Input</in>
  </port_connection>
  <control_connection>
    <out>Sender_0.out</out>
    <in>Reader_0.Offset</in>
  </control_connection>
  <flowcontrol type="Lazy"/>
</network>
```

Fig. 9. Listing 1. Example of an XML-based 4MPS Network definition. First, Processing objects and their configurations are listed, then port and control connections, and finally the kind of flow control policy to use.

Usability: Is it easy to build new models and generate systems using 4MPS? Is the metamodel usable by third parties? Can existing projects easily be converted to the metamodel? This is probably the hardest question to answer especially because of the ambiguity of the word *easily*. In order to give a complete answer we will have to resort to the finer grain dimensions offered by the Cognitive Dimensions framework. Nevertheless, as a first take on the issue it is important to highlight that the metamodel has been adopted by third parties in several ways. Not only have third parties used the metamodel through the CLAM framework, but what is more interesting, some third party frameworks have embraced the metamodel. Also, existing applications have been ported to the metamodel through the CLAM framework.

Once the main questions have been addressed we will now concentrate on finer-grain dimensions included in the Cognitive Dimensions framework. Most of them are actually related to the Usability issue outlined above.

Closeness of Mapping: This refers to how directly entities in the problem domain are translated onto the software solution. As already reported in [36], visual dataflow-based languages help in improving this dimension. In the case of 4MPS this has proved to be even more so because of the addition of the secondary control flow and the direct representation of program objects in the visual language.

Viscosity: This dimension is defined as the "resistance to change" of a particular language or representation. As reported again in [36] visual languages have issues with viscosity because of the fact that it is not always easy to maintain a compact and understandable visual representation. While this is so, it is important to remember that the 4MPS metamodel does not rely only on its visual representation. In previous sections we also introduced an XML representation for 4MPS systems with a one-to-one mapping to the visual language. This allows the use of regular text-based tools and operations (i.e., "Replace All") that reduce the number of primitives needed for an action therefore reducing viscosity. Furthermore, as already explained, the metamodel can also be instantiated through scripting or object-oriented programming languages.

Hidden Dependencies: These occur when a part in a program is affected by another part without any explicit connection. In 4MPS all connections are explicit and no other communication is allowed between processes. Hidden dependencies are greatly reduced in comparison to traditional approaches.

Hard Mental Operations: This deals with how hard it is to understand the different steps involved in an operation and usually relates again to the difficulty in combining individual primitives. As reported in [36] this is directly related to the amount of possible primitives and the different interfaces/behaviors they offer. In 4MPS only one primitive exists (the Processing object) and the interface is limited by clear entry points—ports and controls. The fact that composability is also offered reduces greatly the number of hard mental operations in a particular layer.

Imposed Guess Ahead: This dimension refers to how much the user is forced to make a decision before the information is available. Graphical languages help improve this dimension [36] and 4MPS is no exception. It is for this same exact reason that the 4MPS visual editor implemented in CLAM has been used successfully as a prototyping tool in several settings.

Secondary Notation: This deals with how easy it is to add extra information or groupings that do not relate to the general program functionality. The nonexistence of this secondary notation has been reported as a major shortcoming in graphical languages [36]. The fact that the 4MPS metamodel can be represented both visually and through text or any other programming language allows for secondary notation to be used under any circumstance.

VI. Conclusion

We previously defined a Multimedia Processing System as presenting a clear *separation between data and processes*, and being *stream oriented*, capable of processing *multiple data types*, *interactive*, *complex*, and *software based*. The metamodel presented in this paper addresses all of these concerns and offers practical modeling solutions that can yield efficient applications with very different requirements.

A metamodel can be evaluated through the systems it generates and their usefulness in a particular domain. In Section V, we showed how the metamodel is validated experimentally and also through the evaluation of several formal dimensions. All of this leads to the conclusion that 4MPS provides a useful and clear way to model multimedia processing systems, and it offers an explicit graphical model of computation that helps in modeling complex systems and understanding them.

In Section II, we listed the main benefits of using models. But, as Willrich *et al.* note [53] maybe the two main benefits of describing a model with formal semantics are that the model can be checked and verified against design errors, and the model can be used to derive executable code. We have shown how 4MPS yields itself to formal analysis allowing for not only verification but also for implementing scheduling algorithms. Also we showed how the metamodel not only can be implemented in application frameworks such as CLAM but its visual language can be used to create a *visual builder* such as the Network Editor that allows to derive executable code from the system specification.

But besides the previous benefits, and perhaps most importantly, a metamodel as the one presented defines a common vocabulary and a conceptual meeting point for those interested in multimedia system modeling and software development. Multimedia system designers that know 4MPS are able to communicate their ideas better and more efficiently. Although there is still a long way to go to formally define our field we hope to have contributed in that direction.

Acknowledgment

The author wishes to thank the Developers and Multimedia Researchers for fruitful discussions which led to the metamodel presented in this paper: first and foremost, the CLAM Development Team (currently led by P. Arumi and D. Garcia, Universitat Pompeu Fabra), S. Pope (Project Leader for the CSL framework) for help in reviewing earlier versions of the metamodel and this paper, and E. Lee, G. Tzanetakis, R. Dannenberg, M. Puckette, and others, for contributing their reviews and suggestions.

References

[1] K. Rao, Z. Bojkovic, and D. A. Milovanovic, *Multimedia Communication Systems. Techniques, Standards and Networks*. Englewood Cliffs, NJ: Prentice-Hall, 2002.

[2] M. K. Mandal, *Multimedia Signals and Systems*. Norwell, MA: Kluwer, 2002.

[3] R. Earnshaw and J. Vince, Eds., *Multimedia Systems and Applications*. New York: Academic, 1995.

[4] S. Hashimoto, Ed., *Multimedia Modeling. Modeling Multimedia Information and Systems*. Singapore: World Scientific, 1995.

[5] D. A. Manolescu, "A dataflow pattern language," in *Proc. 4th Patt. Lang. Programm. Conf.*, 1997.

[6] P. Arumi, D. Garcia, and X. Amatriain, "A dataflow pattern language for sound and music computing," in *Proc. Patt. Lang. Programm. (PloP'06)*, 2006.

[7] C. Hylands *et al.*, "Overview of the Ptolemy Project," Tech. Rep., Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, 2003.

[8] W. Rowe, "Why system science and cybernetics?," *IEEE Trans. Syst. Cybern.*, vol. SSC-1, no. 1, pp. 2–3, Nov. 1965.

[9] A. Hall and R. Fagen, "Definition of system," in *Yearbook of the Society for the Advancement of General Systems Theory*. Ann Arbor, MI: General Systems, 1956.

[10] E. Seidewitz, "What models mean," *IEEE Softw.*, vol. 20, no. 5, pp. 26–32, Sep./Oct. 2003.

[11] S. J. Meller, A. M. Clark, and T. Futagami, "Model driven development," *IEEE Softw.*, Sep. 2003.

[12] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA: Addison-Wesley, 2003.

[13] ACM Object Management Group, Unified Modeling Language (UML) Specification: Infrastructure, Version 2.0 Mar. 2006.

[14] Object Management Group, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Nov. 2005.

[15] H. Knublauch, "Ontology-driven software development in the context of the semantic web: An example scenario with protege/owl," in *Proc. Int. Workshop Model-Driven Semantic Web*, Monterey, CA, 2004.

[16] V. Devedzic, "Understanding ontological engineering," *Commun. ACM*, vol. 45, no. 4, Apr. 2002.

[17] S. Cook, "Domain-specific modeling and model driven architecture," *MDA J.*, pp. 2–10, Jan. 2004.

[18] X. Amatriain, "An Object-Oriented Metamodel for Digital Signal Processing With a Focus on Audio and Music," Ph.D. dissertation, Univ. Pompeu Fabra, Barcelona, Spain, 2005.

[19] D. M. Buede, *The Engineering Design of Systems*. New York: Wiley, 1999.

[20] G. Kahn, "The semantics of a simple language for parallel programming," *Inf. Process.*, pp. 471–475, 1974.

[21] T. M. Parks, "Bounded Schedule of Process Networks," Ph.D. dissertation, Univ. California, Berkeley, 1995.

[22] E. Lee and T. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–799, May 1995.

[23] H. Storrle, "Semantics and verification of data flow in uml 2.0 activities," *Electron. Notes Theoret. Comput. Sci.*, vol. 127, no. 4, 2005.

[24] S. Sauer and G. Engels, "Extending uml for modeling of multimedia applications," in *Proc. IEEE Symp. Vis. Lang.*, 1999, pp. 80–87.

[25] D. K. Mellinger, G. E. Garnett, and B. Mont-Reynaud, "Virtual digital signal processing in an object-oriented system," in *The Well-Tempered Object. Musical Applications of Object-Oriented Software Technology*. Cambridge, MA: MIT Press, 1991, pp. 188–194.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns—Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[27] T. Grotker, R. Schoenen, and H. Meyr, "Unified specification of control and data flow," in *Proc. 1997 IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP'97)*, 1997, p. 271.

[28] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich, "Spi—A system model for heterogeneously specified embedded systems," *IEEE Trans. Very Large Scale Integration (VLSI)*, vol. 10, no. 4, pp. 379–389, Aug. 2002.

[29] O. Levia and C. Ussery, "Directed control data-flow networks: A new semantic model for the system-on-chip era," in *Proc. FDL'99*, Lyon, France, Sep. 1999, pp. 548–560.

[30] H. W. van Dijk, H. J. Sips, and E. F. Deprettere, "On context-aware process networks," in *Proc. . Int. Symp. Mobile Multimed. Appl. (MMSA'02)*, Dec. 2002.

[31] M. Geilen and T. Basten, "Requirements on the execution of kahn process networks," in *Proc. Eur. Symp. Programm. Lang. Syst.*, 2003, pp. 319–334.

[32] J. Buck and E. A. Lee, "The token flow model," in *Advanced Topics in Dataflow Computing and Multithreading*. New York: IEEE Computer Society Press, 1994.

[33] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, Jan. 1987.

[34] N. Burroughs, A. Parkin, and G. Tzanetakis, "Flexible scheduling for dataflow audio processing," in *Proc. Int. Comput. Music Conf.*, 2006, pp. 79–82.

[35] M. Zelkowitz and D. Wallace, "Experimental validation in software engineering," *Inform. Softw. Technol.*, vol. 39, no. 11, Nov. 1997.

[36] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *J. Vis. Lang. Comput.* vol. 7, no. 2, pp. 131–174, 1996 [Online]. Available: htpp://citeseer.ist.psu.edu/green96usability.html

[37] X. Amatriain, P. Arumi, and D. Garcia, "Clam: A framework for efficient and rapid development of cross-platform audio applications," in *Proc. ACM Multimed.*, 2006.

[38] X. Amatriain, "Clam: A framework for audio and music application development," *IEEE Softw.*, vol. 24, no. 1, pp. 82–85, Jan./Feb. 2007.

[39] D. Roberts and R. Johnson, "Evolve frameworks into domain-specific languages," in *Proc. 3rd Int. Conf. Pattern Languages Programm.*, Monticello, IL, Sep. 1996.

[40] X. Amatriain and P. Arumi, "Developing cross-platform audio and music applications with the CLAM framework," in *Proc. Int. Comput. Music Conf.*, 2005.

[41] D. Garcia, P. Arumi, and X. Amatriain, "Visual prototyping of audio applications," in *Proc. 2007 Linux Audio Conf.*, 2007.

[42] X. Amatriain, J. Bonada, A. Loscos, and X. Serra, "Spectral processing," in *DAFX: Digital Audio Effects (Udo Zoelzer, ed.)*. New York: Wiley, 2002, pp. 373–438.

[43] X. Amatriain, J. Massaguer, D. Garcia, and I. Mosquera, "The CLAM annotator: A cross-platform audio descriptors editing tool," in *Proc. 6th Int. Conf. Music Inform. Retrieval*, London, U.K., 2005.

[44] S. T. Pope and C. Ramakrishnan, "The create signal library ('Sizzle'): Design, issues and applications," in *Proc. Int. Comput. Music Conf. (ICMC'03)*, 2003.

[45] S. T. Pope, X. Amatriain, L. Putnam, J. Castellanos, and R. Avery, "Metamoodels and design patterns in CSL4," in *Proc. Int. Comput. Music Conf. (ICMC'06)*, 2006.

[46] S. Bray and G. Tzanetakis, "Implicit patching for dataflow-based audio analysis and synthesis," in *Proc. Int. Comput. Music Conf. (ICMC'05)*, 2005.

[47] J. Castellanos, "Design of a Framework for Spatial-Audio Rendering," M.S. thesis, Univ. California, Santa Barbara, 2006.

[48] J. Prats, "Blackfinger: Artificial Vision and Hand Recognition System," M.S. thesis, Univ. Pompeu Fabra (UPF), Barcelona, Spain, 2006.

[49] K. Mayer-Patel and L. Rowe, "Design and performance of the Berkeley continuous media toolkit," in *Proc. Multimed. Comput. Network.*, San Jose, CA, 1997, pp. 194–206.

[50] P. Ackermann, "Direct manipulation of temporal structures in a multimedia application framework," in *Proc. ACM Multimedia Conf.*, 1994.

[51] A. R. François and G. G. Medioni, "A modular middleware flow scheduling framework," in *Proc. ACM Multimedia'00*, Los Angeles, CA, Nov. 2000, pp. 371–374.

[52] C. J. Lindblad and D. L. Tennenhouse, "The VuSystem: A programming system for compute-intensive multimedia," *IEEE J. Select. Areas Commun.*, vol. 14, no. 7, pp. 1298–1313, Jul. 1996.

[53] R. Willrich, P. D. Saqui-Sannes, P. Senac, and M. Diaz, "Multimedia authoring with hierarchical timed stream Petri nets and java," *Multimed. Tools Applic.*, vol. 16, no. 1–2, pp. 7–27, 2002.

[54] G. Tzanetakis and P. Cook, "Marsyas3D: A prototype audio browser-editor using a large-scale immersive visual and audio display," in *Proc. Int. Conf. Auditory Display (ICAD)*, 2001.

[55] P. Cook, "Synthesis toolkit in C++," in *Proc. 1996 SIGGRAPH*, 1996.

[56] A. Chaudhary, A. Freed, and M. Wright, "An open architecture for real-time audio processing software," in *Proc. Audio Eng. Soc. 107th Conv.*, 1999, pp. 1–4.

[57] V. Lazzarini, "Sound processing with the SndObj library: An overview," in *Proc. 4th Int. Conf. Dig. Aud. Effects (DAFX'01)*, 2001.

[58] M. Young, D. Argiro, and S. Kubica, "Cantata: Visual programming environment for the khoros system," *Comput. Graph.*, vol. 29, no. 2, pp. 22–24, 1995.

[59] T. Lourens, "TiViPE—Tino's visual programming environment," in *Proc. 28th Annu. Int. Comput. Softw. Appl. Conf. (COMPSAC'04)*, 2004, pp. 10–15.

[60] C. Upson *et al.*, "The application visualization system: A computational environment for scientific visualization," *IEEE Comput. Graph. Applicat.*, vol. CGA-9, no. 4, pp. 32–40, Jul. 1989.

[61] A. R. François and G. G. Medioni, "A modular software architecture for real-time video processing," in *Proc. IEEE Int. Workshop Comput. Vis. Syst.*, Vancouver, BC, Canada, Jul. 2001, pp. 35–49.
[62] S. S. G. Engels, "Object-oriented modeling of multimedia applications," in *Handbook of Software Engineering and Knowledge Engineering*. Singapore: World Scientific, 2002, vol. 2, pp. 21–53.
[63] V. Zivkovic and P. Lieverse, "An overview of methodologies and tools in the field of system-level design," in *Proc. Embedded Processor Design Challenges*, 2002, pp. 74–88.
[64] J. Sztipanovits, G. Karsai, and T. Bapty, "Self-adaptive software for signal processing," *Commun. ACM*, vol. 41, no. 5, pp. 66–73, May 1998.
[65] S. Zachariadis, C. Mascolo, and W. Emmerich, "The satin component system-a metamodel for engineering adaptable mobile systems," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 910–927, Nov. 2006.

Xavier Amatriain studied telecommunications engineering at the Universitat Politecnica de Catalunya and received the Ph.D. degree in computer science and digital communication from the Universitat Pompeu Fabra, Bareclona, Spain.

He has recently joined the Telefonica I+D Research Center, Barcelona, Spain. Previously, he was Research Director at the CREATE center, University of California Santa Barbara, where he coordinated the Allosphere project (http://www.mat.ucsb.edu/allosphere). He is also Project Leader and coauthor of the CLAM Framework. He has authored more than 20 papers in journals and international conferences. His research interests include multimedia and signal processing system modeling, music information retrieval, software engineering, agile methodologies, and open source software development.