Frameworks Generate Domain-Specific Languages: A Case Study in the Multimedia Domain

Xavier Amatriain and Pau Arumi

Abstract—We present an approach to software framework development that includes the generation of domain-specific languages (DSLs) and pattern languages as goals for the process. Our model is made of three workflows—framework, metamodel, and patterns—and three phases—inception, construction, and formalization. The main conclusion is that when developing a framework, we can produce with minimal overhead—almost as a side effect—a metamodel with an associated DSL and a pattern language. Both outputs will not only help the framework evolve in the right direction, but will also be valuable in themselves. In order to illustrate these ideas, we present a case study in the multimedia domain. For several years, we have been developing a multimedia framework. The process has produced a full-fledged domain-specific metamodel for the multimedia domain, with an associated DSL and a pattern language.

Index Terms—Domain-specific architectures, visual programming, life cycle, CASE.

1 Introduction

The benefits of modeling languages are well established in engineering. The basic idea is to come up with a set of commonly accepted concepts, notations, and rules in order to better express problems and solutions. This way we are mimicking the flexibility of a natural language, in which a vocabulary, syntax, and grammar allow the expression of complex ideas by combining atomic semantic and functional units.

UML filled a gap in software engineering by standardizing a generic modeling language. The existence of such a flexible and general-purpose modeling language is good enough in many cases. In others, however, we would prefer to have a more tailored and specialized tool, even if that means compromising flexibility and genericity. Furthermore, the generic UML is sometimes distant from the concepts and tools used in some particular domains. Having domain experts adopt UML is not always the best solution: We would like our generic software engineering tools and concepts to adapt to these particular domains.

DSL aims at solving these issues by offering a comprehensive modeling language tailored to a particular domain. General modeling techniques and practices are combined with a thorough domain analysis. The result is a subset or subclass of those general techniques, tools, and practices that better fit the particular application domain.

DSLs, as with any language, are made of a vocabulary, a grammar, and a syntax. The vocabulary, and the grammar and syntax at the abstract level, are provided by an associated metamodel. But the concrete syntax will depend on the way the particular DSL is implemented, e.g., whether it is a visual or textual-based DSL. In other words, a given metamodel which includes every component of a DSL except for its concrete syntax might be implemented or realized through different concrete syntaxes.

There are two common approaches to define a DSL: Start from the general-purpose UML and constrain and refine its usage to better embrace domain specificities, or use a generic metamodeling tool in order to define a new modeling language that relates to domain modeling concepts (see Mernik et al.'s survey [26] for more information on standard approaches to DSL design).

Here, we present a different approach that consists of integrating the definition of a DSL in the development process of a software framework. Rather than proposing this as a general-purpose approach to defining a DSL, our goal is to highlight that both ideas are so intimately related that when developing a domain framework using recorded best practices, it is possible to obtain a full-fledged DSL with very little added overhead: Frameworks generate domain-specific languages. The resulting DSL will fall in the category of *embedded* DSLs. An embedded DSL, as defined by Hudak [18], is a DSL that is derived from a general-purpose programming language, inheriting the infrastructure and tailoring it to the specific domain.

Our approach to DSL design is part of a comprehensive framework development process model that aims at not only producing a high-quality framework, but also at providing a domain-specific language and a pattern language. Those outputs may well become more valuable than the framework itself since they can be reused beyond the framework. Our proposed process model explicitly distinguishes three workflows—framework, metamodel, and patterns—and three phases—inception, construction,

- *X. Amatriain is with Telefonica Research, Via Augusta, 177, Barcelona 08021, Spain. E-mail: xavier@amatriain.net.*
- *P. Arumi is with Barcelona Media, Av. Diagonal, 177, planta 9, 08018 Barcelona, Spain. E-mail: pau.arumi@barcelonamedia.org.*

Manuscript received 17 June 2009; revised 23 Nov. 2009; accepted 28 Feb. 2010; published online 26 Mar. 2010.
Recommended for acceptance by P. Strooper.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2009-06-0157.
Digital Object Identifier no. 10.1109/TSE.2010.48.

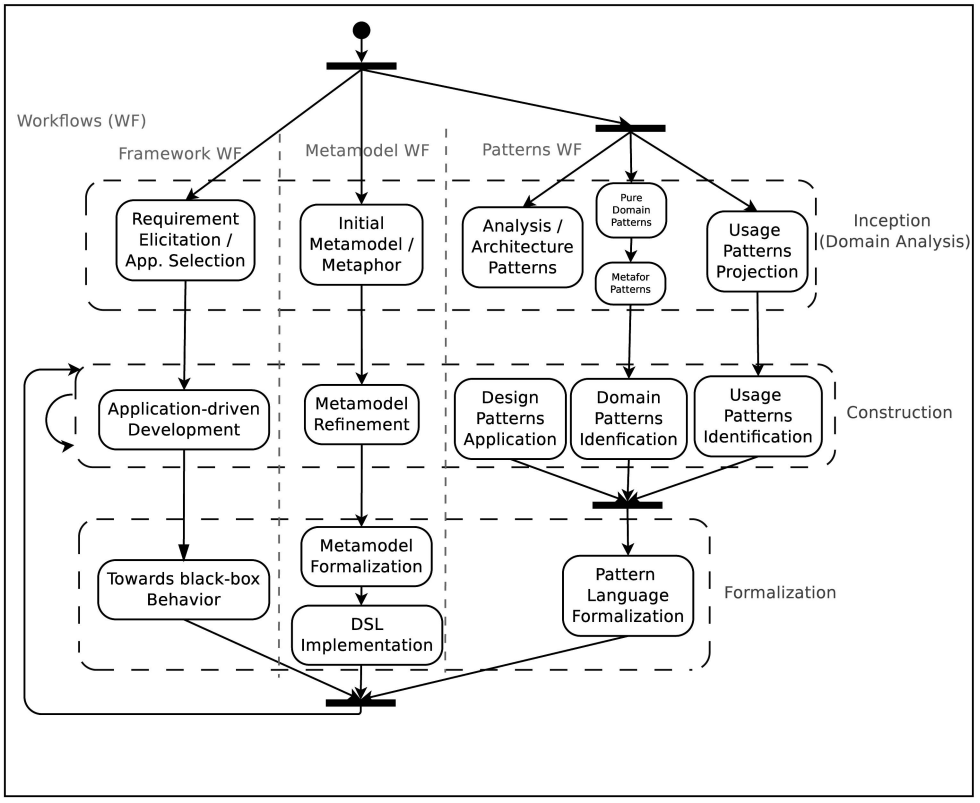

Fig. 1. The main activities in our framework development process model that includes three workflows—framework, metamodel, and patterns—and three phases—inception, construction, and formalization.

and formalization—in the development process. We also promote the iterative nature of process so that all outputs—framework, DSL, and pattern language—are derived incrementally, feeding from the evolution of the other products. We describe all of these workflows and phases, including their interactions, in Section 2.

In some sense, our approach resembles the one outlined by Roberts and Johnson [28] in their collection of framework patterns. We review their approach, together with others related process models, in Section 3.

In order to illustrate these ideas, we present a case study in the multimedia domain in Section 4. For several years, we have been developing CLAM, a framework for audio and multimedia (see Section 4.1). The process has produced a full-fledged domain-specific metamodel (presented in Section 4.2), a DSL with several concrete syntaxes (see Section 4.3), and a pattern language (see Section 4.4).

2 A DSL-Oriented Framework Development Process

In this section, we propose a general-purpose process model for framework development. The goal of such a process will be—beyond the obvious design of a well-constructed and useful framework—the generation of a domain-specific language (or several) and a pattern language. The iterative formulation of a domain metamodel drives the design process, which can be considered a particular approach to Model-Driven Development (MDD) [25] in the context of framework development.

The proposed process model is mainly derived from the authors' experience in developing the CLAM framework. The details of this particular process and its outcomes will be explained in Section 4. We shall now focus on formulating a general-purpose approach based on those learnings.

Fig. 1 illustrates the main activities in the proposed framework development process. The process is divided into three separate threads or *workflows*: *Framework Workflow*, *Metamodel Workflow*, and *Patterns Workflow*. Each of these workflows encompasses and frames activities at a particular abstraction level.

We also show three distinct phases in the development process, which we call *Inception*, *Construction*, and *Formalization*.[1] Each phase hosts different activities with a different goal in mind. However, note that we provide several iteration points. For instance, the Construction phase, where most of the development will occur, should be completely incremental with iterations as short as possible. Also, once the formalization phase is concluded, we do not expect the framework to remain unchanged. For this reason, we provide the model with an iteration point that leads back into the Construction phase.

Our proposed model is in fact favorable to *agile* principles. Among other things, we advocate for a simple upfront analysis and design, and an iterative approach for the rest of the process. Our process can therefore be considered as a concrete implementation of *agile metamodeling* [6] in the context of framework development.

1. Although these names bear some resemblance to the Rational Unified Process, our model is not related to the RUP framework.

The novelty in our process model is not so much in the way a particular workflow is addressed since, in each of them, we rely on well-known techniques and practices. The novelty is in the fact that the three workflows are addressed in parallel, and we expose an explicit relation and communication channels between all three workflows.

2.1 Framework Workflow

The framework workflow hosts all of the activities that focus on delivering a useful framework in the *traditional* sense [13], [29]. Therefore, our proposed approach in this workflow relies on applying well-known best practices for framework development. In particular, we base our process model on the following principles:

- small initial investment,
- iterative and incremental development,
- application-driven framework design,
- strive for black-box behavior.

In the next paragraphs, we will detail how these principles are put into practice into each of the process phases.

2.1.1 Inception: *Framework Requirement Elicitation*

We do not favor a big upfront analysis and design phase. If this is seldom advisable when developing any kind of application, it is even less so when developing a framework. The number of use cases and breadth of the problem make the unknowns grow exponentially.

We cannot aim at capturing the details of a whole domain during this initial phase. However, there are important activities that we should undertake in order to obtain a first rough domain analysis. When developing a regular system, we usually identify *actors* that will interact with our system. It is important to bear in mind that in a framework, a special, and most important, form of actors will be the concrete applications that will be developed within the framework. Therefore, one of the basic activities in this phase will be to identify a set of relevant applications that play the role of the actors by defining some initial use cases. Ideally, we want around three to four applications that give a good enough sample of the problem space, i.e., they should have requirements as different as possible and represent as many of the targeted use cases as possible. In CLAM, for instance, we choose an application with strong real-time requirements, another one with a complex process flow, and a final one with a focus on a complete and flexible user interface (see Section 4.1 for more details). The initial selection of applications to drive framework development is captured by the *Three Examples* pattern in Roberts and Johnson's catalog [29].

Ideally, the applications we choose to drive our framework design will already be developed and fully functional. The goal when choosing these applications is to treat them as if they were to be refactored into the framework infrastructure. A welcome side effect of choosing working applications is that these will have developers with a good understanding of the domain. These developers should help us understand the domain and should become our on-site clients on the technical side.

The inception phase should be as short as possible. However, finding the set of relevant applications and understanding their requirements is not a trivial task.

2.1.2 Construction: *Application-Driven Framework Design*

In the *inception* phase, we chose applications with the idea of refactoring them into the framework, while the framework itself is designed to host the requirements posed by them. However, although some applications might yield themselves to being progressively refactored into a framework, from our experience it is usually better to rewrite them from scratch. Sample applications set the basis for the requirements, but these requirements evolve during the process. Driving applications might well be treated as *throw-away prototypes*. If the process is done correctly, the framework will give birth to many more successful applications, and even the original driving applications should be improved if still needed.[2]

Framework design is an ongoing and continuous refinement; each new iteration is likely to impose new requirements and challenges onto it. But, after some iterations, the core framework development should become minimal. This idea of *core framework* is similar to Bosch et al.'s model of framework development, where they distinguish between *core framework* development and *internal increments* [13]. But in our process model, we promote internal increments that end up delivering the core framework as a consequence, rather than trying to fix the core in an initial upfront design.

2.1.3 Formalization: *Black Box Behavior*

The goal of the framework workflow should be to come up with a framework that is as close as possible to a black-box behavior. A black-box framework is one in which you can develop applications by simply plugging components together without the need to use inheritance and understand the internal behavior of the classes [21].

In order to reach the ideal completely black-box behavior, we would need to offer ready-to-use versions of any component that could be needed in the framework domain. This might be feasible if the framework is focused on a small enough subdomain. However, most of the time our goal will be to minimize, rather than to eliminate, the need for white-box intervention.

In this phase, once the framework infrastructure (or core) is stable enough, we will deliver a black-box version of the most useful components. As an alternative, we can sometimes design *pluggable objects* [28], classes that can be easily parameterized in order to change the behavior, therefore, reducing the number of classes we need to effectively cover the problem space.

A fully functional black-box framework is very close to a DSL. If we have followed a similar progression in the metamodel workflow, all we need to do is add an appropriate syntax and tools to interact with the model by plugging components. As a matter of fact, Johnson and Roberts' *Visual Builder* [28], which is a natural evolution of the black-box framework, can be considered a visual domain-specific language.

2. Many times the functionality that was originally offered by a focused application is offered as a service in the framework.

2.2 Metamodel Workflow

While the goal of the framework workflow is to come up with an appropriate set of tools for application developers in the domain, the goal of this metamodel workflow is to come up with the appropriate *concepts*. In a similar way, we will start with an open and evolving metaphor that will turn into a more concrete domain metamodel [16] as iterations go. The final goal will be to turn this metamodel into a DSL with several syntaxes that offer the users direct access to the metamodel concepts. Ideally, one of these syntaxes will be of a graphical form, therefore becoming the *visual builder* in the framework.

2.2.1 Inception: *Domain Analysis or the First Metamodel*

One of the main activities in the inception phase is to perform some sort of domain analysis. The main activities involved in domain analysis are:

1. domain characterization,
2. data collection,
3. data analysis,
4. classification, and
5. evaluation of the domain model [7].

These activities will take place regardless of the approach. However, time and effort devoted to each task will vary. For instance, in approaches that lean toward agile methods like ours, activities 3 and 4 will tend to be minimized in the initial phase, since they are expected to occur concurrently and iteratively during the development phase. Furthermore, when developing a framework, we want the domain analysis to be application driven. Therefore, "domain characterization" will be mainly done by identifying the appropriate applications that cover the problem space. And "data collection" will imply analyzing these applications and understanding both their semantics and requirements. Finally, note that our model evaluation will be done through the framework implementation itself.

In the framework workflow, our goal was to select those applications that will help us define the requirements and will be used to drive the framework development. In this metamodel workflow, however, we are concerned with activities that deal with modeling domain concepts and constructs. As a result, a first and rough metamodel should be verbalized. This first metamodel can be, in most senses, treated as the "system metaphor" [34] and should be used to better communicate between framework developers and the potential users. It should set a common understanding of what is being built and with what purpose by defining particular terminology and procedures.

This metaphor should be treated as an "evolving metamodel." We do not expect it to be a ground truth and its evolution will in fact mark its refinement. We should expect this metamodel to become more concrete and also multilayered in each iteration.

At this stage, a few issues should already be taken into account when defining this metamodel:

- The reason we call this a *domain metamodel* as opposed to a regular domain model is that the concepts should not be tailored to a particular application but rather capture the whole range of possible applications that the framework is targeting in the given domain.
- One of the main objectives of sketching this metamodel is to better communicate with the domain experts that are targeted as users. Remember that this will become the shared metaphor.
- Because of the previous point, it is important to choose appropriate names and to consult with the input of domain experts when doing this analysis.
- Yet, it is important to remember—and explicitly state—that this is an evolving metamodel. Some of the preconceived ideas about the domain might prove wrong or inexact during the framework development process. We—and the domain experts—should be prepared to *embrace change*.

2.2.2 Construction: *Metamodel Refinement*

The metamodel refinement takes place throughout the framework development and in some sense will still be active as long as the framework keeps evolving. However, most of the metamodel refinement takes place in the Construction phase, when white-box components (i.e., abstract classes) are refactored to accommodate new requirements in the driving applications.

Metamodel refinement should aim at making the first rough approach more concrete and broad at the same time. Concrete because particular applications and the framework implementation will have to be correctly explained by the metamodel and broad because the more applications and use we give to the framework, the more we will be testing the metamodel validity and its scope.

Unfortunately, metamodel testing cannot be automated in any way since it deals with verifying that conceptual constructs and concepts used in the evolving metaphor are valid as the framework design evolves. Metamodel testing at this stage consists on the following activity: Given the current iteration in the framework development process, verify that all concepts and constructs in the white-box framework infrastructure can be expressed simply by using the evolving metaphor. Make sure that all driving applications can also be interpreted in terms of the metamodel. Pay special attention to changes introduced in the latest framework increment. If any of the previous is not verified, iterate over metamodel/metaphor until it can naturally explain the current state in the framework. Other complementary activitites for metamodel and DSL validation that require having a more stable metamodel will be described in the formalization phase for this same workflow. However, those activities can also be integrated with the iterations.

Metamodel refinement is not a sequential activity that needs to be executed at the end of each iteration—adding the possibility of blocking the development process. It is an activity in the metamodel workflow and should therefore be carried out in parallel to any framework development. Furthermore, we have explained the natural and obvious flow of communication from the framework to the metamodel. However, sometimes—especially in later iterations, once the metamodel is more stable—it is the metamodel that will remain fixed and enforce some changes in the next iterations of the framework if this has drifted from the metamodel without having a good enough reason for doing so.

2.2.3 Formalization: *Metamodel Consolidation and DSL Implementation*

Increments at the metamodel level will lead to a formalized metamodel in the Formalization phase. But, even this formalization might need to adapt to further refactorings as the framework evolves.

In any case, once we have a fairly stable metamodel and the framework white-box behavior has been defined, we should be ready to realize this metamodel through a DSL. Again, given the iterative nature of our proposed model, it may well be that a DSL already exists even if in a preliminary form. But, at this stage, the implementation of a DSL will simply require deciding on a given notation that reflects the metamodel and implementing the tool to interact with it.

One of the most difficult parts of this metamodel workflow is to integrate proper testing and validation. We now describe some of the activities that have proven useful to validate the metamodel and associated DSL. Although we describe them as if they were to occur at the end of the workflow, as a sort of *acceptance test*, many of them can and should be integrated into the iterations so that the validation is also performed iteratively.

As we showed in the evaluation of our multimedia metamodel and DSL [4], a proper evaluation of a metamodel should include a combination of qualitative and quantitative studies. But, because in software engineering, validation and implementation go hand in hand [37], we base part of the evaluation in the concrete implementation of the metamodel.

A first and necessary condition to prove the validity of a metamodel is to prove that it is *implementable* and its implementation can be used to develop a variety of systems. Another necessary condition to prove the usefulness of the metamodel is to show that it can help understand systems outside the framework that originated it. We can do this by explaining how similar systems in the same domain can be explained or even synthesized using the metamodel and its DSL.

Finally, we can formally assess the validity of the framework by combining dimensions derived from the general software engineering corpus [37] with others borrowed from the Cognitive Dimensions Framework [17]. Higher-level dimensions can be used to validate and test the metamodel, while lowever level dimensions, such as the ones in the Cognitive Dimensions, can help us validate the DSL.

Some high-level dimensions that can be use to validate a metamodel are [36]: *feasibility*, *completeness*, and *usability*. *Feasibility* refers to how practical the abstractions in the metamodel are and how well they fit the requirements in our particular domain. *Completeness* reflects two complementary questions: 1) Can *any* system in the domain be modeled with the metamodel that is proposed and 2) can a system be completely modeled using an instance of the metamodel? And *Usability* tries to answer questions such as whether it is easy to build new models and generate systems using the metamodel, whether the metamodel is usable by third parties, and whether existing projects can easily be converted to the metamodel.

The Cognitive Dimensions Framework [17] includes finer-grain dimensions that can be used to validate the DSL. This framework has been specifically designed for evaluating visual languages, but many of its dimensions are applicable to nonvisual DSLs. The framework includes dimensions such as: *Viscosity*, *Hidden Dependencies*, *Hard Mental Operations*, *Imposed Guess Ahead*, and *Secondary Notation*.

2.3 Patterns Workflow

Patterns should be instrumental throughout the framework development process. As a matter of fact, they should influence the development process in many ways. Not only do we promote the use of different kinds of patterns, but we also introduce the idea that patterns should be one of the expected outputs of the process. A complete pattern language should be the perfect complement for the full-fledged metamodel and DSL obtained in the metamodel workflow. Patterns help us document frameworks [20], but not only that: Using patterns allows us to understand the framework as a composition of patterns [35].

In the following paragraphs, we will highlight how patterns are used, discovered, and formulated in the different phases of this pattern workflow.

2.3.1 Inception: *Patterns for Analysis*

The first thing we need to be aware of and understand are the so-called *meta patterns*. According to Pree: "Meta patterns are a set of design patterns that described how to construct a framework independently of the domain" [27]. An example of meta patterns that should be reviewed before starting the task of designing a framework are those by Roberts and Johnson [29]. Understanding these meta patterns will be the first use of patterns in our pattern workflow.

On the other hand, it should be obvious that this inception phase is the natural host for all sort of analysis patterns. These patterns can be grouped into three different categories: 1) general analysis patterns, 2) specialized analysis patterns, and 3) domain patterns [30].

However, generally applicable analysis patterns, independent of the domain, are hard to find. And when they exist, they tend not to give a very concrete solution to an analysis problems. A good example are some of the patterns in the GRASP catalog [22]. Also related are general purpose architectural patterns, such as those in the POSA catalog [14]. These can indeed be considered a form of analysis patterns since they can define a high-level starting point for our design.

But what we are most interested in our process model is uncovering domain patterns. These domain patterns will follow an analysis process made of three different but related activities:

- **Pure Domain Patterns Elicitation**: By talking to domain experts and regular stakeholders, our goal is to identify patterns that are used in the domain. Patterns should be, as always, a recurring solution to a problem in a context. At this stage, these domain patterns can be completely detached from any software solution since they simply explain how domain experts address those issues and situations that we will be covering in our framework.
- **Metaphorical Patterns Description**: Given the metaphor (i.e., rough domain metamodel) described in

the metamodel workflow, we already want to identify related patterns in this phase. This basically entails translating the pure domain patterns elicited in the previous step to the concepts and constructs of the driving metaphor.

- **Usage Patterns Projection**: By choosing the driving applications in the framework workflow, we are defining what we deem are relevant framework use cases. As a matter of fact, an initial assessment of the requirements posed by the driving applications will already uncover a number of recurring usage needs. At this point, we are likely to identify some recurring problems and contexts but not their solution. Still, it is a good practice to record these usage patterns and bear them in mind in the next phases.

Some of these uncovered domain patterns might be related to preexisting patterns in the neighboring domains. Therefore, we should not forget to support our findings by searching for pattern catalogs describing similar or related domains. In the multimedia pattern language, we present in Section 4.4, for instance, we were able to reuse and adapt patterns coming from related dataflow languages.

2.3.2 Construction: *Pattern Evolution*

The main goal for the construction phase is to evolve the different patterns that were identified in the inception phase. First, patterns related to the metaphor should progressively evolve into metamodel patterns with the final goal of being able to understand the metamodel as a composition of these patterns. As a matter of fact, some of these metamodel patterns might at some point become a part of the metamodel since they will be defining concepts and constructs that are inherent to the metamodel itself.

On the other hand, this is the phase where usage patterns will be formulated and become concrete. The different iterations in the framework workflow will end up offering solutions to the usage problem-context pairs identified in the inception phase. Usage patterns will also help evolve the framework and the metamodel in the right direction.

The construction phase is also the natural place where standard design patterns will be used. Ideally, these patterns will not only be used, but will also help define the low-level design for some of the metamodel patterns. As a matter of fact, some of the metamodel patterns might be a sort of specialized design patterns for the given domain.

2.3.3 Formalization: *Pattern Language Formalization*

As seen in the previous phases, both the metamodel and the framework itself give place to a number of patterns that include domain patterns, metamodel patterns, specialized design patterns, and usage patterns. This does indeed prove the feedback between the three workflows.

Our final goal in this workflow is to define a *generative pattern language*,[3] i.e., a pattern language that not only explains rules of arrangement, but also allows users to create endless combinations [2]. Therefore, patterns should aim at being coherent. However, they should also be useful in isolation and in different settings than the ones defined by the metamodel. Patterns in our final generative pattern language may address high-level architectural problems, offer solutions to a particular usage of the framework, or give low-level design details for some of the components.

3. Note that the use of the word "language" in this context is not consistent with the formal notion of language as in Domain Specific Language. We have chosen to use the expression "Pattern Language" for consistency with existing literature. In this context, a Pattern Language should be understood, as defined by Johnson, as "a set of patterns, each of which describes how to solve a particular kind of problem" [20].

3 Related Approaches

Roberts and Johnson proposed an approach to framework development that is somewhat similar to ours [28]. As a matter of fact, and to the best of our knowledge, it is the only approach that touches upon the three workflows included in our process model. They propose basing framework development in a series of patterns that cover different phases of the development process. For instance, the *Three Examples* pattern advocates for identifying three applications early on to drive the design. The main phases according to this pattern language would be *white box*, *black box*, *visual builder*. This is indeed similar to our proposed approach in which a DSL should be the ultimate goal of the development process—as a matter of fact, the DSL can be understood as the sum of the *visual builder* and the *language tools* pattern also included in their catalog.

However, there are important differences between our model and that of Robert and Johnson. First, although the authors do set a path to go from initial framework design to a DSL, they do not establish the definition of the DSL as a goal of the process itself. As a matter of fact, they consider that many frameworks will never make it to the final stage, so they will never have the need to provide any kind of DSL. The authors do not distinguish different workflows—although they do explain that some patterns are concurrent—or identify phases and activities. More importantly, perhaps, they do not establish a clear relation between parallel activities. Also, it is difficult to interpret how to fit in the iterative nature of the process into their pattern language. Finally, and although their main concern is the patterns workflow, they do not consider how patterns at different levels may be needed. For instance, they do not discuss the need for design patterns or domain-specific patterns. In any case, we do not see our process model as opposed to this pattern language. Our approach is more comprehensive, detailed, and structured, but it is completely compatible with Roberts and Johnson patterns, and complements them in many ways.

Van Deursen et al. [32] explain that a typical development of a DSL involves three steps with a number of phases:

- Analysis:
  1. identify problem domain,
  2. gather domain knowledge,
  3. cluster knowledge into small number of concepts, and
  4. design a DSL that describes applications in the domain.

- Implementation:
  1. construct a library that implements the concepts and
  2. design and implement a compiler that translates DSL programs to library calls.
- Use: 1 write DSL programs for all applications.

Although this approach bears some resemblance tp our three phases, there is a very important conceptual difference in that it promotes an *analysis-first* approach to DSLs. Also, there is no notion of iteration between phases. Most importantly, it does not provide any relation between the development of the DSL and that of the framework.

Aksit et al. [1] propose a four-phase approach to building frameworks using domain models. First, they model the top-level structure of the framework using the so-called *knowledge graphs*. Second, they refine each node in the graph into an acyclic subknowledge graph called *knowledge domain*. Third, they identify which nodes can be included together in the top-level knowledge graph. Finally, they map knowledge domains into OO concepts. This approach is interesting in that it addresses both the framework and metamodel workflows. But, the model is focused so much on formalities that its practical applicability in the general case is not clear.

Van Deursen notes the similarity between DSLs and OO frameworks in his case study on the financial domain [31]. He concludes that when developing a DSL from scratch, it makes sense to do it by extending an OO framework. Using a DSL in the context of a framework development has, according to van Deursen, the following advantages: 1) It is a guide to the framework design since any construct that does not fit naturally into the DSL should probably not be in the framework either, 2) it encourages black box, as opposed to white box, behavior in frameworks, and 3) it gives more abstract access to the framework, encapsulating even the language used to develop the framework. The author's conclusions when relating frameworks and DSLs are in essence very similar to ours except that the starting point is different: We advocate for a process that integrates metamodel and framework workflows since the beginning, while Van Deursen starts from the premise that a preexisting framework can benefit from a DSL.

Bonachea et al. [12] present a practical case study of developing a DSL for costumer user profiling. They advocate for a completely iterative process model, with iteration occurring between most activities. They report executing the following activities:

1. interview domain experts,
2. develop models,
3. write programs that observe the models by hand,
4. design the language,
5. write programs using the language,
6. implement runtime system and language compiler.

In particular, they stress the importance of keeping domain experts involved during the whole process. Although this approach focuses only on our metamodel workflow, it is interesting since it highlights the iterative nature of the DSL nature, which is also a very important conclusion of our approach.

Cleaveland [15] proposes a process model for building *application generators*, which are a particular case of DSL in which a compiler translates high-level specifications into a regular low-level programming language. The process they propose is made of seven steps:

1. recognize domains,
2. define domain boundaries,
3. define an underlying model,
4. define the variant and invariant parts,
5. define specification input,
6. define products, and
7. implement the generator.

According to Cleaveland, all but the last step are led by domain analysts. His proposed approach can be seen as a serialization of our concurrent metamodel and framework workflows in which initial activities are more related to the metamodel and final ones to the framework.

Yacoub and Ammar describe a pattern-oriented approach to build software systems known as POAD (*Pattern-oriented Analysis and Design*)[35]. In particular, they focus on how to use design patterns starting already in the analysis phase. They identify two approaches to using patterns: *stringing*, in which patterns are glued together to compose a design, and *overlapping*, in which the same class can belong to several patterns. POAD advises for the use of the stringing approach at the higher levels of abstractions while allowing for overlapping patterns in the detail design of the lower levels. Although this approach is a good approximation to our patterns workflow, there are several differences. For instance, the POAD approach only deals with the use of preexisting design patterns assuming that a pattern library exists when the analysis phase starts. The main goal of the analysis phase is in fact to select the most appropriate patterns, which are later integrated into the model in the design phase. Also, analysis patterns are not included in the POAD process nor is the goal of the development to uncover new domain-specific patterns.

Finally, Jacobsen et al. present a pattern-oriented approach specific for framework development [19]. They highlight the distinction between regular design patterns and *meta patterns*, and show that patterns are useful in different ways in all phases of the framework development process—analysis, design, and implementation. During these phases, not only new patterns are created, but others are evolved by either transformation or replacement. The authors capture the importance of both using and generating patterns during the process in an incremental manner, but they do not show how to formalize this into a pattern language nor mention any connection to metamodeling activities.

4 A Case Study in the Multimedia Domain

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 or user interfaces. Although probably less known, the audio and music fields also have a long tradition of similar development tools. It is

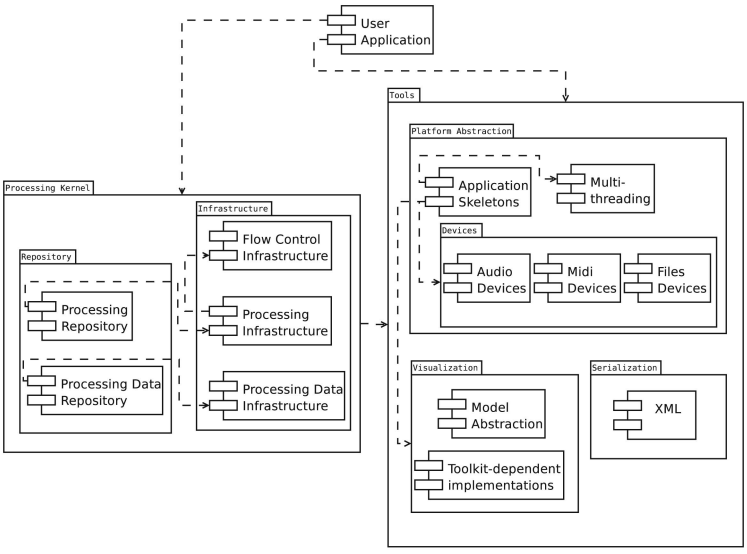

Fig. 2. CLAM components. The CLAM framework is made up of a Processing Kernel and some tools. The Processing Kernel includes an *infrastructure* that is responsible for the framework white-box behavior and *repositories* that offer the black-boxes. Tools are usually wrappers around preexisting third party libraries. A user application can make use of any or all of these components.

in this context where we find our award-winning CLAM framework [3].[4]

During the CLAM development process, several parallel activities have taken place. While some sought the goal of having a more usable framework, others dealt with the appropriate abstractions and reusable constructs in the domain. The latter gave place to the definition of a complete metamodel for the multimedia domain, and a pattern language for dataflow-oriented systems. Most of these ideas, although a result of the CLAM process itself, are validated by their presence in many other multimedia frameworks and environments.

Our experience in this development process originated the general approach we presented in Section 2. As such, CLAM touches upon the three workflows and phases described therein. In this section, we will not detail the activities and practices used in the process since they are already documented in the general-purpose approach. Instead, we will describe the output of each of the workflows. In Section 4.1, we explain the main components and features of the framework, which are the result of the framework workflow. In Section 4.2, we explain the metamodel, and in Section 4.3, we show how this metamodel can be accessed through a number of associated DSLs. Finally, Section 4.4 briefly explains the pattern language that was produced in the patterns workflow.

4.1 CLAM: A Multimedia Processing Framework

CLAM (originally from C++ Library for Audio and Music) is a full-fledged software framework for application development. Although it was initially tailored for audio and music, it has also proven its applicability to the broader Multimedia domain. CLAM has been used for applications that range from on-the-fly analysis of video soundtracks [33] to 3D audio spacialization and integration with 3D visual scenes [10]. It offers a conceptual domain-specific metamodel, algorithms for analyzing, synthesizing, and transforming audio signals, tools for handling audio and music streams and creating cross-platform applications, and ready-to-use applications.

We will now highlight the main features in CLAM. For further information, please refer to our comprehensive overview [5] or to any of the more focused publications cited therein. CLAM, as well as all other included applications mentioned in this paper, is available for download in the project webpage.[5]

CLAM offers a *processing kernel* that includes an *infrastructure* and processing and data *repositories* (see Fig. 2). CLAM is both a *black-box* and a *white-box* framework [28]. It is black box because built-in components already included in the repositories can be connected with minimal or no programmer effort in order to build new applications. It is *white box* because the abstract classes that make up the infrastructure can be derived to extend the framework functionality with new processes or data classes.

The CLAM *infrastructure* is the result of an in-depth and iterative domain analysis. It encompasses a number of abstract classes that are responsible for the white box or extensible behavior in the framework. In order to build a particular CLAM system, the user has to instantiate the concrete-derived classes or implement a derived class that might add a new specific processing capability. The *infrastructure* component also includes the application logic, such as dataflow graph management and nodes execution.

CLAM contains a *processing repository* and a *data repository*. The processing repository contains a large set of ready-to-use processing algorithms. On the other hand, the data repository contains all the classes that act as data containers or encapsulated versions of the most commonly used data in the domain. These classes make use of the data

4. It received the 2006 ACM Best Open-Source Multimedia Software Award.

5. http://www.clam-project.org.

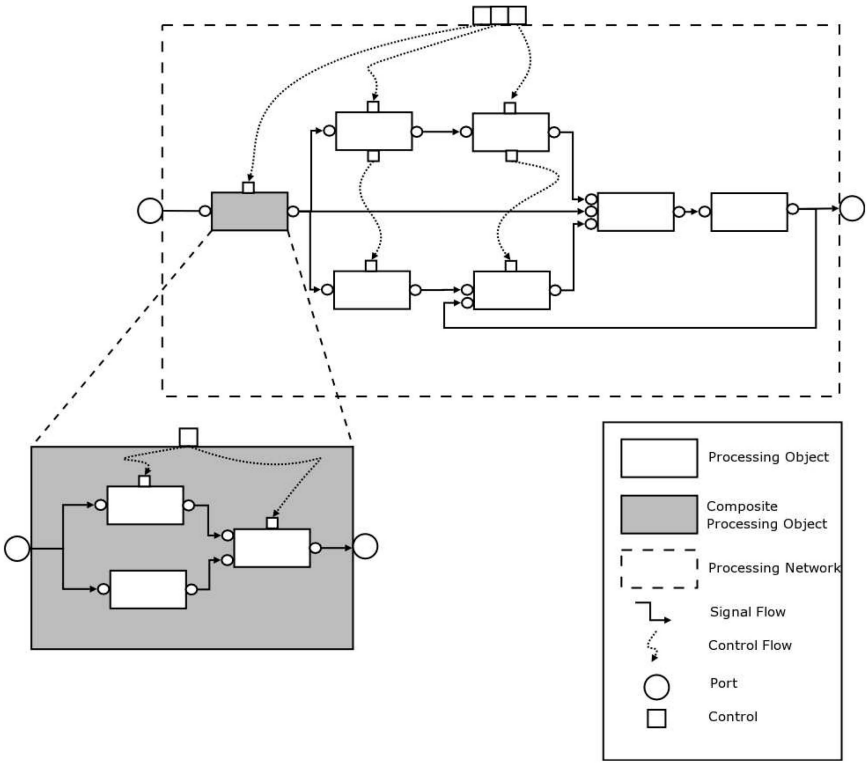

Fig. 3. Graphical model of a 4MPS processing network. Processing objects are connected through ports and controls. Horizontal left-to-right connections represent the synchronous signal flow, while vertical top-to-bottom connections represent asynchronous control connections.

infrastructure and are therefore able to offer metaobject services such as a homogeneous interface or built-in automatic XML persistence.

CLAM also includes a number of tools for services, such as input/output or XML serialization. These tools aim at being a swiss-army knife of services that might be needed in the domain. All of these tools are possible because of the integration of third-party open libraries into the framework. In this sense, one of the benefits of using CLAM is that it acts as a common point for already existing heterogeneous services [3].

The framework has been tested on—but its development has also 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 metamodel, the use of the design patterns, and the benefits of the framework.

4.2 4MPS: A Multimedia Domain-Specific Metamodel

The result of the CLAM development process in its metamodel workflow was the Metamodel for Multimedia Processing Systems (4MPS for short) [4], 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.

For this reason, we proposed 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 actor-oriented graphical models of computation used in system engineering. The metamodel is not only an abstraction of many ideas found in the CLAM framework, but also the result of an extensive review of similar frameworks. It is therefore expected that domain experts are familiar with most of its concepts and constructs.

The metamodel 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 meta-object-like facilities, such as reflection and serialization.

Although the metamodel clearly distinguishes between these two different kinds of objects, the managing of Data constructs can be almost transparent for the user. We can therefore describe a 4MPS system as a set of Processing objects connected in graphs called *Networks* (see Fig. 3).

The metamodel can also be expressed in the language of graphical models of computation as a particular case of *Dataflow Networks* [23]. Different properties of the systems, such as their optimal schedule or minimal latency [8], can be derived in this way.

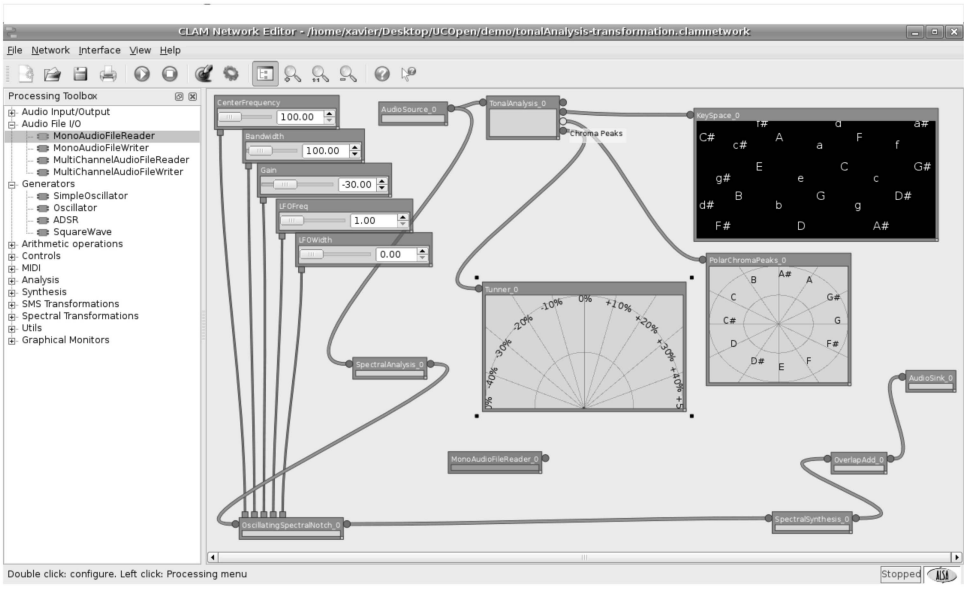

Fig. 4. NetworkEditor is the visual DSL for the CLAM framework. It can be used not only as an interactive multimedia dataflow application, but also to build networks that can be run as stand-alone applications.

4.3 Accessing the Metamodel through a Domain-Specific Language

The 4MPS metamodel offers a domain-specific ontology that helps software developers understand the domain and helps domain experts understand the framework. However, we are still lacking the concrete tools that give users easy access to all these services and put them together in a coherent way: We lack the concrete syntax given by the notation in which to express it, we need a domain-specific language.

One immediate way to access all of these services and interact with the metamodel layer is to use the framework itself and code new applications by using the black boxes that are provided and extending the white boxes. In some sense, the code—together with the metamodel and the patterns—provides a low-level DSL (see CLAM classes and correspondence to metamodel concepts in Fig. 5). There is, however, an important issue with using this approach: The DSL syntax becomes coupled to the low-level programming language syntax. This makes it hard for users to focus on the metamodel level and it becomes a barrier to its understanding.

For this reason, we decided to offer an alternative and more accessible DSL syntax in the form of a visual language. The 4MPS is itself a graphical metamodel, so offering access to this level becomes immediate and simply a matter of implementing the appropriate tool. In CLAM, this tool is known as the NetworkEditor—because of its relation to the Network class in the metamodel. The Network Editor allows users to interact directly with a graphical representation of the metamodel (see Fig. 4), which in turn maps directly to framework classes and constructs.

The Network Editor is a standalone application developed by adding a presentation layer to the framework classes. The user can directly access the repository of black-box components and interact with it by configuring objects and defining 4MPS Networks. The tool can not only be used to build fast prototypes, but it can in fact generate final applications, either standalone or audio plugins, with efficiently compiled code.

Both the metamodel and the associated DSL were evaluated using the approach described in Section 2.2.3. See the publication, where the metamodel is presented for more details on the results [4].

4.3.1 *Optimized CLAM DSL Syntaxes for Specific Problems*

Although the graphical representation of the metamodel is usually preferred, it is sometimes practical to have a direct one-to-one mapping to a textual format. In the case of CLAM, XML was chosen as the basis for our textual-based DSL syntax (see the example in Listing 1). Because these textual files contain a complete definition of a 4MPS Network, they have a direct visual representation. This allows users to interact graphically with the metamodel without directly accessing the CLAM framework. As a matter of fact, the Network Editor itself bases its persistency format on this XML schema.

```
<?xml version="1.0" encoding="UTF-8"
  standalone="no" ?>
<network id="ExampleNetwork">
  <processing id="file_reader"
    type="AudioFileReader">
    <SourceFile>
      <URI>/home/xavier/0001.wav</URI>
    </SourceFile>
    <Loop>1</Loop>
  </processing>
  <processing id="sink" type="AudioOut">
  </processing>
  <processing id="control_sender"
    type="OutControlSender">
    <Min>0</Min>
    <Default>0</Default>
    <Max>1</Max>
    <Step>0.01</Step>
```

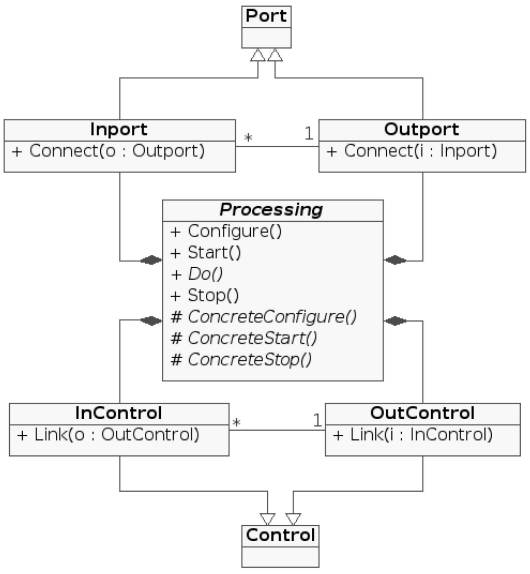

Fig. 5. The main participant classes in the CLAM implementation of the 4MPS metamodel.

```
    <ControlRepresentation>Vertical
      Slider</ControlRepresentation>
  </processing>
  <port_connection>
    <out>file_reader.Samples</out>
    <in>sink.Audio Input</in>
  </port_connection>
  <control_connection>
    <out>control_sender.out</out>
    <in>file_reader.Offset</in>
  </control_connection>
  <flowcontrol type="Push"/>
</network>
```

Listing 1: Simplified example of a 4MPS Network definition using CLAM's XML Networks DSL syntax.

```
network = ClamNetwork(file(fullSourcePath))
network.setConfig(source, "NSources",
  numPorts))
for i in range(numPorts) :
  newProcessing = "%s_%s"%(delay,i)
  network.duplicateProcessing(delay,
    newProcessing, 10*i, 50*i)
  network.addConnection
    ('control_connection',"
    BackgroundDelay",
    "0", newProcessing, "Delay in Samples")
```

Listing 2: Example of a complex CLAM network defined as a modification of a previous network (maybe designed with the Visual Networks syntax) using the Scripted Networks DSL syntax.

In this way, we see that the metamodel is not coupled to a particular visual language, but can in fact be instantiated by different syntaxes, such as an XML dialect or a scripting language. As a matter of fact, once we have the metamodel and the framework in place, we can think about extending the available syntaxes with other syntaxes that are optimized for some particular uses.

The following list summarizes the five syntaxes that have been developed within CLAM with their advantages and disadvantages:

- **Black-Box C++** is code in C++ using the black-box framework style (i.e., instantiating Processing objects via factories, connecting their data and control ports, configuring their parameters, and setting them into running state.

  Pros: It gives total flexibility and it is appropriate for some specific uses, such as ones requiring a complex application logic.

  Cons: Exposes the C++ syntax and low-level (non-domain-related) details to domain-experts. It may also be unsafe since the user can introduce operations that compromise the execution requirements, such as real-time constraints.
- **Black-Box Scripting** is based on Python code in the black-box framework style.

  Pros: It has a simplified syntax when compared to the black-box C++, while still retaining much of its flexibility. It allows for interactive signal-processing scripting in Python. It eases the process to graphically represent data using drawing packages.

  Cons: It does not allow running in real time with low-latency, and it does not separate the network configuration state from the network running state.
- **Scripted Networks** Python code using a module for creating and modifying previously created networks (see Listing 2) This syntax is used to scale up simpler networks created with the Visual Networks DSL syntax. Therefore, the two concrete syntaxes work synergistically in the same workflow.

  Pros: It is the concrete DSL syntax in CLAM that allows for better management of complexity because of its high-level interface for creating and modifying networks. The runtime of this concrete DSL syntax is CLAM's Prototyper, which offers both real time and offline operation.

  Cons: Though it comes close, it is not as flexible as the black-box framework approaches.
- **XML Networks** is the textual XML definition of a network (see Listing 1).

  Pros: It gives access to all of the details the network. Can be written programatically and executed efficiently with CLAM's Prototyper. It allows for repetitive operations by using textual functions such as "copy and paste" or "replace all." It is the serialization format of the Visual Networks.

  Cons: It is less intuitive, difficult to modify, and easily causes errors. Better manipulated through intermediate tools.
- **Visual Networks** is the NetworkEditor's visual building language (see Fig. 4).

  Pros: It is an intuitive and didactical way to describe multimedia processing algorithms allowing both signal processing and control flows. It executes

compiled code efficiently, while allowing users to interact with control parameters and to receive visual feedback of data flowing through the network.

Cons: It is difficult to manage big or repetitive networks, for example, when the number of subnetworks is a configurable parameter. It is also difficult to reconfigure parameters consistently across many processing objects in the network (or many networks).

4.4 A Pattern Language for Multimedia Processing Systems

While 4MPS offers a valid high-level metamodel for the domain, it is sometimes more useful to present a lower level architecture in the language of design patterns, where recurring and nonobvious design solutions can be shared. Such a pattern language bridges the gap between an abstract metamodel such as 4MPS and the concrete implementation given a set of constraints. It also provides an efficient way to document the framework itself [20].

In the following paragraphs, we offer a brief summary of a complete pattern language for dataflow real-time multimedia processing formalized by the authors [9].

All of the patterns in our catalog fit within the generic architectural pattern defined by Manolescu as the *Data Flow Architecture* [24]. However, this architectural pattern does not address relevant problems related to real-time multimedia processing, such as message passing protocols, scheduling of processing-objects executions, or data management. Our pattern language is organized within three main categories:

- **General Dataflow Patterns** address problems of how to organize high-level aspects of the dataflow architecture by having different types of module 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 those patterns.
- **Network Usability Patterns** address how humans can interact with dataflow networks.

Some of the patterns in the catalog are very high-level, while others are more focused on implementation issues. Although the catalog is not domain complete, it can 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 can be used to further refine the solution. These relations form a hierarchical structure depicted in Fig. 6. The arcs between patterns represent "enables" relations: Introducing a pattern in the system enables other patterns to be used.

The catalog shows how to approach the development of a complete dataflow system in an evolutionary fashion without the need to do *big up-front design*. On each decision which will introduce more features and complexity, a recurrent problem is faced and addressed by one pattern in the language.

Two of these patterns, *Typed Connections* and *Port Monitor*, are central to CLAM because they enable two key features of the framework which are: One, the ports are

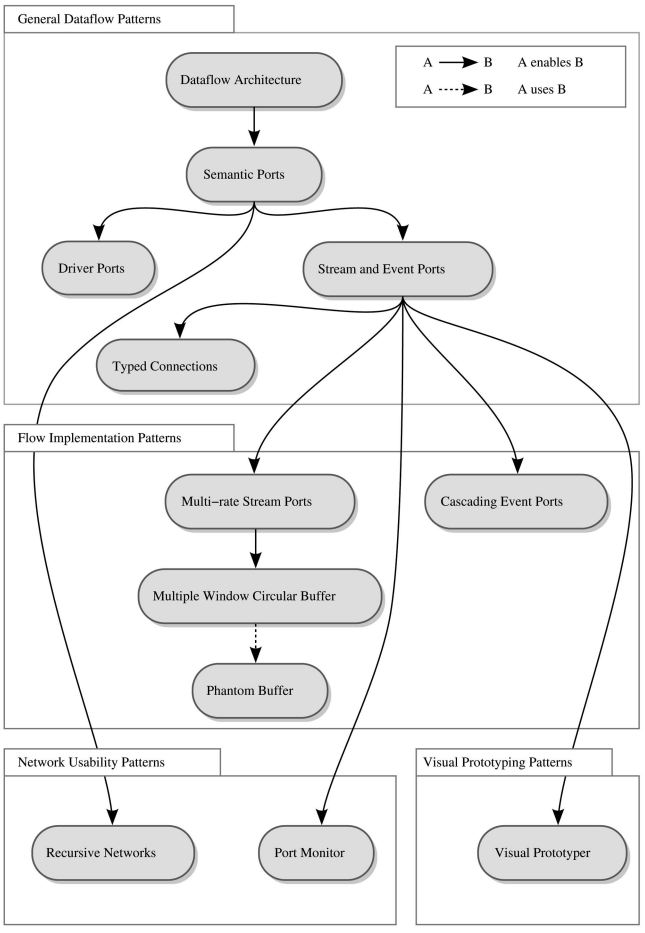

Fig. 6. The multimedia dataflow pattern language. High-level patterns are on the top and the arrows represent the order in which design problems are being addressed by developers.

typed but not restricted to a number of types, and two, the processed data can be visualized in real time while keeping up with the lock-free constraints of the processing thread.

We shall now provide here a summarized version as an example of the kinds of patterns available in the catalog. Apart from their importance in the context of CLAM, we have chosen these two patterns for another reason: They both have broad applicability beyond the specific context of our framework. Complete versions of these and the rest of the patterns can be found in the original catalog [9].

4.4.1 Pattern: Typed Connections

Context: Multimedia dataflow systems might need to manage different kinds of tokens. In the audio domain, we might need to deal with audio buffers, spectra, spectral peaks, MFCC's, etc. 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 should have to deal with this complexity, and this is not desirable. It is better to directly provide them with 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.

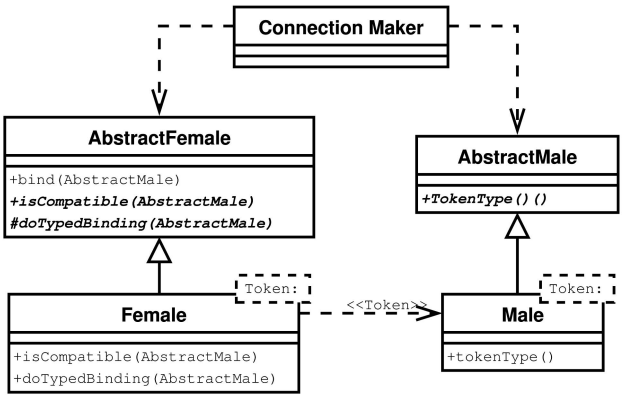

Fig. 7. Class diagram of a canonical solution of typed connections.

Using typed connections may imply that the entity that handles the connections should deal with all of the possible types. This could at least imply that the connection entity would have a maintainability problem.

Problem: Connectible entities communicate typed tokens but token types are not limited. Thus, how can a connection maker do typed connections without knowing the types?

Forces:

- The processing thread is cost sensitive and should avoid dynamic type checking and handling.
- Connections are done at runtime by the user, so mismatches in the token type should be handled.
- Dynamic type handling is a complex and error-prone programming task, thus placing it on the connection infrastructure is preferable to placing it on concrete modules implementation.
- The collection of token types evolves and grows and this should not affect the infrastructure.

Solution: 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. Let the connection maker set the connections thorough the generic interface, while the connected entities use the token-type coupled interface to communicate with each other. Access typed tokens from the concrete module implementations using the typed interface. Fig. 7 shows the class diagram for this solution.

Use runtime type checks when modules get connected (*binding time*) to make sure that connected ports types are compatible, and, once they are correctly connected (*processing time*), rely only on compile-time type checks. To do this, the generic connection method on the abstract interface (`bind`) delegates the dynamic type checking to abstract methods (`isCompatible` and `typeId`) implemented on token-type-coupled subclasses.

Consequences: The solution implies that the connection maker is not coupled to token types. Only concrete modules are coupled to the token types they use.

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

Because both sides of the connection know the token type, buffering structures can deal with tokens in a wiser way when doing allocations, initializations, copies, etc. Concrete modules only have access to the static typed tokens. So, no dynamic type handling is needed.

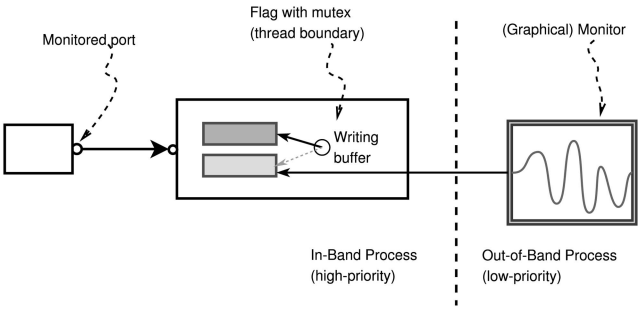

Fig. 8. A port monitor with its switching two buffers.

4.4.2 Pattern: Port Monitors

Context: Some multimedia applications need to show a graphical representation of tokens that are being produced by some module out-port. While the visualization only has soft requirements related to its smoothness, the process has real-time requirements. This normally requires splitting visualization and processing into different threads, where the processing thread is scheduled as a high-priority thread. But, because the non-real-time monitoring must have access to the processing thread tokens, some concurrency handling is needed and this often implies locking in the two threads.

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 (i.e., The process result must be calculated in a given time slot).
- Visualizations must be fluid; that means that it should visualize on time and often but it may skip tokens.
- The processing is not filling all of 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* offer the visualization thread a special interface to access tokens in a thread-safe way. Internally, they have a lock-free data structure, which can be simpler than a lock-free circular buffer since the visualization can skip tokens.

To manage concurrency and avoid process stalling, the *Port monitor* uses two alternated buffers to copy tokens (see Fig. 8). 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 blocked, it is just writing on the same buffer while the visualization holds the lock.

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, as it only lasts a single flag switching.

Unfortunately, the visualization thread may suffer starvation risk, not because the visualization thread will be blocked, but because it may always be reading from the same buffer. This may happen if, every time the processing thread tries to switch the buffers, the visualization is blocking. Experience tells us that this effect is not critical and can be avoided by minimizing the time the visualization thread is accessing tokens, for example, by copying and releasing them.

5 Conclusions: Frameworks Generate Domain-Specific Languages

We have presented a framework-development process that aims at generating a domain-specific metamodel with associated domain-specific languages and a pattern language. In most cases, it is unrealistic to deploy a DSL by designing a full-fledged domain framework as there are easier and more direct ways of obtaining the benefits of DSLs. However, our proposal is the complement: When designing a framework you should aim at producing a DSL.

When building an application framework, we are generalizing across a set of systems that belong to a particular domain. We aim at offering the tools and the conceptual infrastructure needed to implement all of those systems. A well-designed framework is not just about reuse of code, but also about conceptual reuse: It should present a precise model of computation and a conceptual framework or domain metamodel. The white-box components (i.e., base classes) of the framework are mainly responsible for it.

Our process model is iterative but promotes the separation of concerns in three different workflows—framework, metamodel, and patterns—and activities in three different phases—inception, construction, and formalization. In order to derive our particular domain metamodel, we need to perform some initial analysis to identify basic requirements, understand different viewpoints, and choose driving applications in the inception phase. But, we cannot aim at understanding and modeling the whole domain from the start. The framework-development process is, as most software development, iterative by nature. Thus, just as the framework is iteratively constructed, so should the metamodel and the pattern language be refined in each iteration.

And once we have a stable domain metamodel, it is fairly straightforward to provide associated domain-specific languages. We have the concepts and constructs; all we are missing is an appropriate notation. The code, together with the white-box and black-box components it allows us to access, can already be considered an initial—and low-level—DSL. Adding textual or visual concrete syntax to this well-defined metamodel is only an implementation detail.

Our main conclusion is that any well-conducted framework design process will produce a DSL. Therefore, just as *patterns generate architectures* [11], **frameworks generate domain-specific languages**.

Acknowledgments

The authors would like to thank all of the developers of the CLAM framework who have participated in the process described in this paper throughout the years. In particular, they mention the continuous contribution of David Garcia. Work in this paper has been partially funded by an ICREA grant from the Catalan Government, the Universitat Pompeu Fabra, and Barcelona Media.

References

[1] M. Aksit, F. Marcelloni, and B. Tekinerdogan, "Developing Object-Oriented Frameworks Using Domain Models," *ACM Computing Surveys,* p. 11, 2000.

[2] C. Alexander, *The Timeless Way of Building.* Oxford Univ. Press, 1979.

[3] X. Amatriain, "Clam: A Framework for Audio and Music Application Development," *IEEE Software,* vol. 24, no. 1, pp. 82-85, Jan./Feb. 2007.

[4] X. Amatriain, "A Domain-Specific Metamodel for Multimedia Processing Systems," *IEEE Trans. Multimedia,* vol. 9, no. 6, pp. 1284-1298, Oct. 2007.

[5] X. Amatriain, P. Arumi, and G.D., "A Framework for Efficient and Rapid Development of Cross-Platform Audio Applications," *ACM Multimedia Systems,* vol. 14, no. 1, pp. 15-32, 2008.

[6] S.W. Ambler, "Agile Model Driven Development Is Good Enough," *IEEE Software,* vol. 20, no. 5, pp. 71-73, Sept./Oct. 2003.

[7] G. Arango, "A Brief Introduction to Domain Analysis," *Proc. ACM Symp. Applied Computing,* pp. 42-46, 1994.

[8] P. Arumi and X. Amatriain, "Time-Triggered Static Schedulable Dataflows for Multimedia Systems," *Proc. Multimedia Computing and Networking Conf.,* 2009.

[9] P. Arumi, D. Garcia, and X. Amatriain, "A Dataflow Pattern Language for Sound and Music Computing," *Proc. Pattern Languages of Programming Conf.,* 2006.

[10] P. Arumi, D. Garcia, T. Mateos, A. Garriga, and J. Durany, "Real-Time 3D Audio for Digital Cinema," *J. Acoustical Soc. Am.,* vol. 123, no. 5, p. 3937, 2008.

[11] K. Beck and R. Johnson, "Patterns Generate Architectures," *Proc. Eighth European Conf. Object-Oriented Programming,* 1994.

[12] D. Bonachea, K. Fisher, A. Rogers, and F. Smith, "Hancock: A Language for Processing Very Large-Scale Data," *Proc. Second Conf. Domain-Specific Languages,* pp. 163-176, 1999.

[13] J. Bosch, M. Molin, M. Mattson, and P. Bengtsson, "Object-Oriented Frameworks: Problems & Experiences," *Building Application Frameworks,* John Wiley and Sons, 1999.

[14] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns.* John Wiley & Sons, 1996.

[15] J. Cleaveland, "Building Application Generators," *IEEE Software,* vol. 5, no. 4, pp. 25-33, July 1988.

[16] S. Cook, "Domain-Specific Modeling and Model-Driven Architecture," *J. Model-Driven Architecture,* pp. 2-10, Jan. 2004.

[17] T.R.G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *J. Visual Languages and Computing,* vol. 7, no. 2, pp. 131-174, 1996.

[18] P. Hudak, "Building Domain-Specific Embedded Languages," *ACM Computing Surveys,* vol. 28, 1996.

[19] E.E. Jacobsen, B.B. Kristensen, and P. Nowack, "Characterising Patterns in Framework Development," *Proc. 25th Int'l Conf. Technology of Object-Oriented Languages and Systems,* 1997.

[20] R.E. Johnson, "Documenting Frameworks with Patterns," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* 1992.

[21] R.E. Johnson and J. Foote, "Designing Reusable Classes," *J. Object Oriented Programming,* vol. 1, no. 2, pp. 22-35, June/July 1988.

[22] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process,* second ed. Prentice Hall PTR, July 2001.

[23] E. Lee and T. Parks, "Dataflow Process Networks," *Proc. IEEE,* vol. 83, pp. 773-799, May 1995.

[24] D.A. Manolescu, "A Dataflow Pattern Language," *Proc. Fourth Pattern Languages of Programming Conf.,* 1997.

[25] S.J. Meller, A.M. Clark, and T. Futagami, "Model Driven Development," *IEEE Software,* vol. 20, no. 5, pp. 14-18, Sept./Oct. 2003.

[26] M. Mernik, J. Heering, and A.M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys,* vol. 37, no. 4, pp. 316-344, 2005.
[27] W. Pree, *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.
[28] D. Roberts and R. Johnson, "Evolve Frameworks into Domain-Specific Languages," *Proc. Third Int'l Conf. Pattern Languages for Programming,* Sept. 1996.
[29] D. Roberts and R. Johnson, "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks," *Proc. Third Conf. Pattern Languages and Programming,* vol. 3, 1996.
[30] L. Sesera, "Hierarchical Patterns: A Way to Organize (Analysis) Patterns," *J. Systemics, Cybernetics, and Informatics,* vol. 3, no. 4 pp. 37-40, 2005.
[31] A. van Deursen, "Domain-Specific Languages versus Object-Oriented Frameworks: A Financial Engineering Case Study," *Proc. Conf. Smalltalk and Java in Industry and Academia,* pp. 35-39, 1997.
[32] A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM SIGPLAN Notices,* vol. 35, no. 6, pp. 26-36, 2000.
[33] J. Wang, X. Amatriain, and D. Garcia, "Multilevel Audio Description," *Proc. World Wide Web Conf.,* 2009.
[34] D. West, "Metaphor, Architecture and XP," *Proc. XP Conf.,* 2002.
[35] S. Yacoub and H. Ammar, *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems.* Addison-Wesley Longman Publishing Co., 2003.
[36] S. Zachariadis, C. Mascolo, and W. Emmerich, "The Satin Component System—A Metamodel for Engineering Adaptable Mobile Systems," *IEEE Trans. Software Eng.,* vol. 32, no. 11, pp. 910-927, Nov. 2006.
[37] M. Zelkowitz and D. Wallace, "Experimental Validation in Software Engineering," *Information and Software Technology,* vol. 39, no. 11, pp. 735-743, Nov. 1997.

Xavier Amatriain is a research scientist at Telefonica Research, where he currently leads research projects related to Web mining, social networks, and recommender systems. In the same company, he is involved in activities related to software architecture best practices and agile methods. At the same time, he is an associate professor at the Universitat Pompeu Fabra, where he teaches software engineering and information retrieval. Prior to this, he was a research director at the University of California, Santa Barbara, where he led research in immersive and virtual environments and 3D Audio. He has been coordinating the CLAM project for audio and multimedia processing since its inception in the course of his PhD thesis.

Pau Arumi received the PhD degree in computing science from the Universitat Pompeu Fabra in 2009, and the MSc degree in computer science from the Universitat Politècnica de Catalunya in 2002. He is currently a researcher at Barcelona Media Audio Group, leading the development team and working on real-time 3D audio systems. He has been involved in several industrial projects delivering systems in the areas of 3D audio authoring tools, cinema exhibition, and live events broadcasting. Since 2000, he has been one of the core developers of the CLAM open-source framework. He is a professor of software engineering in the Technology Department of the Universitat Pompeu Fabra. His research interests include dataflow-based real-time multimedia processing, and 3D audio technologies.