A Logic-Based Framework for Digital Signal Processing

David Pastor, Yves Raimond, Samer Abdallah, and Mark Sandler

Centre for Digital Music, Queen Mary, University of London,
{david.pastor,yves.raimond,samer.abdallah,mark.sandler}@elec.qmul.ac.uk

Abstract. In this paper we describe a logic-based framework for Digital Signal Processing (DSP). This framework is implemented as a set of SWI-Prolog modules developing interfaces to heterogeneous signal processing libraries. Prolog terms representing common datatypes and a standard format for binary data are used to mediate between the different components of the system. The system provides the user with a unified and flexible front-end to define signal processing workflows. Using this framework, we develop a signal processing-enabled reasoner capable of performing the primary tasks through the following interfaces: decoding (specific libraries for each format), feature extraction (Vamp plugins), audio processing and effects (LV2/LADSPA plugins) and classification (WEKA machine learning API).

Key words: Digital Signal Processing, SWI-Prolog modules, signal processing workflows, processing-enabled reasoner, interfaced open source engines

1 Introduction

Digital signal processing (DSP) is a broad field and an important object of engineering research. Within this research community, particular investigators are discovering and developing new algorithms for signal processing and analysis. Many projects aim at developing formal APIs providing a framework for algorithms implementation. However, these APIs are normally designed as a closed environment which hardly interacts with other related APIs and applications. Although several APIs are wrapped and exported as "plugin" libraries which can be embedded in different applications, they still require the implementation of specific and non-interactive hosts.

This situation leads to many isolated applications, environments and sources of data working independently from each other. Any effort to combine them to obtain integrated results for analysis, classification and comparison requires tedious studies or "glue code" implementations.

In this paper we address this problematic situation by proposing and developing SWI-DSP, a logic-based framework for *Audio Processing* (as subset of DSP) built on a predicate calculus implementation: SWI-Prolog [15]. We rely

on a logic programming language with foreign language modules providing audio analysis capabilities to get rid of the application-dependency paradigm and offering a single logic-based front-end. Logic resources are also used to achieve a more capable DSP engine: a signal processing-enabled reasoner. The main and primary motivation for a logic-based implementation of such a system is to turn our conception of traditional procedural applications for computation of data towards declarative semantic frameworks to manage knowledge involving specific computations.

In §2 we identify the most common audio analysis tasks and summarise several relevant open source APIs aimed at performing such tasks. We evaluate the advantages and limitations of these APIs in order to find the requirements for SWI-DSP. In §3 we start viewing the system from the top and how it defines an interesting workflow-based architecture that characterises input/output relationships between processing tasks. These workflows are possible by means of common datatypes and a binary data standard format. §4 goes into the implementation of the modules that develop a communication layer with dedicated audio processing libraries.

Thus, SWI-DSP develops predicates triggering external computation performed by dedicated APIs. The datatypes allow and constrain the predicates communication and instantiation. These predicates are organised into modules with different levels of abstraction. However, this architecture lacks of an uniform interface to create programs suitable for the concrete needs. We tackle this problem in §5 by defining an unified and flexible front-end, composed by DSP "scenarios", which allows the user to define workflows with understandable semantics.

Data management is an important paradigm involved in many domains and also in the multimedia research domain. Computations can not often be retrieved from session to session or are stored in local filesystems. Local data repositories are unlikely directed to extend any shareable source of knowledge because they are not expressed in a standard format with common semantics.

Although SWI-DSP does not develop a data management system itself, we discuss this problem in §6 and present related works which contribute to achieve such a multimedia knowledge reasoner/manager. We conclude in §7 by assessing the value and scope of this work and introducing future advances.

2 Audio processing tasks and solutions

A particular application of DSP is the study of audio signals: one-dimensional, time domain signals, however, they also have representations in other domains such as frequency, time-frequency, etc. Audio signals evolve in time and have acoustic properties produced by the signal structure and shape. Music and speech are the main fields involving audio signals.

The properties of an analog and continuous signal (acoustic signals) can be studied by analysing their digital (sampled and quantised) versions. A one-dimension digital signal is therefore a mapping from a discrete time domain

(usually finite) to a discrete value domain and is often represented as a sequence of values associated with successive time points. Digital signals can be processed to make new derived signals or extract some of their characteristics.

We consider the following digital signal processing tasks.

- Effects;
- Feature Extraction;
- Classification.

2.1 Effects

A first step in many signal processing is a normalisation, conditioning or re-representation of the signal, which may involve several techniques such as filtering, amplification, Fourier transforming, etc.

It is very common in musical systems to provide tools that modify acoustic properties of the signal by introducing specific effects. Reverberation, delay or distortion are examples of effects that can be digitally obtained reproducing natural or instrumental alterations of the signal.

In practise, we characterise this group of tasks by its input/output relationships. The operation outputs a signal (which may be time, frequency or multidimensional domain) when it is fed with an input signal. Oscillators and synthesis systems outputs signals without any inputs.

One important dedicated API is the Linux Audio Developer's Simple Plugin API (LADSPA) [10] which has been successfully used in many systems and applications and is a widely spread implementation for algorithms development. We host LADSPA in SWI-DSP but there are other interesting libraries like DSSI plugins for effects and LV2 (new generation of LADSPA) which are targets for future extensions.

2.2 Feature Extraction

Acoustic properties which are the substrate of the psychological perspective of audio signals are conveyed in the signal and therefore in their digital representations. The so-called feature extraction paradigm aims at extracting different levels of audio features to characterise, classify and compare the audio signals (and any signal in general). The domain of the feature represents a qualitative facet of the signal (tempo, chromatics, timbre...), so features do not convey all the signal information. This is interesting as distinct signals may have some very close features and differ significantly in others.

The Vamp Plugins API [11] offers a framework to develop feature extraction plugins that can be hosted in any application able to control the "plugin life-cycle". We host Vamp plugins as basis for feature extraction in SWI-DSP. In general, plugin libraries are characterised for being suitable for development but application-dependent.

2.3 Integrative solutions

With the same motivation of this work there are some integrative APIs which combines different tasks to perform low and high-level operations in a closed implementation. CLAM [5], jMIR [7] and Marsyas [6] present a complex framework collecting tools to perform different DSP tasks. The most interesting feature of these frameworks is the possibility to create DSP networks combining specific algorithms and modules thanks to a workflow-based architecture (Marsyas and CLAM) or with a descriptive data format (jMIR).

CLAM also offers a metamodel of DSP processes which provides semantics to design the networks properly presenting a higher and more manageable level of abstraction. However, these APIs still present two important limitations that our work is aimed at solving:

- They do not offer a common format to exchange data or an easy way to interact with other applications;
- They are based on a procedural system unable to collect and integrate knowledge from different sources to obtain new knowledge.

2.4 Classification

Classification is one the most interesting and complex paradigms in any specific domain exploiting the "intelligence" of computational machines. WEKA (Waikato Environment for Knowledge Analysis) [12] collects different machine-learning algorithms and provides several user interfaces to manage the analysis. It also provides a formal interface (in Java) for development and interaction with other systems.

However, this generic API does not provide semantics to create a dataset which is the most intriguing and complex issue for classifying data properly. Marsyas and jMIR integrate WEKA orienting it for music classification purposes by offering a code interface with other modules (Marsyas) or by extending the WEKA file format ARFF (jMIR). Both solutions still seem to lack of effective semantics and a interactive front-end to perform classification of audio data.

3 SWI-DSP architecture

From the previous comparison and review we consider the following requirements:

- Allowing workflow design;
- Providing an interface to interact with other related systems and applications;
- Featuring an unified, flexible, interactive and understandable front-end;
- Providing an open-development platform;
- Management of shareable data semantically labelled;

– Enhancing traditional procedural implementations by building a logic-based framework of declarative nature allowing reasoning and inferring.

As initial step, we build a framework able to develop digital signal processing workflows based on a predicate logic implementation (SWI-Prolog) extending its vocabulary by wrapping computations under logic predicates. We develop task-oriented modules and define functors capturing the common datatypes allowing the communication between modules and hiding the internal implementation. In addition to this, we define a standard format for binary data exchanged among processing modules. We define these utils in the **swilib** module which provides a C++ library and a SWI-Prolog module for their management.

Processing libraries and analysis applications are normally implemented in fast and efficient code using an imperative language (e.g. C/C++) which is machine-operation oriented instead of focused on task knowledge. Functional programming offers a different paradigm. Functional languages try to capture the meaning of computational tasks in a more declarative way, using functional relationships to describe "what" the computation should be rather than "how" to do it. Modern functional languages like Haskell [14] provide rich and powerful type systems describing and constraining the set of legal values a function can take as input or return as output. This seems to be a better approach to define high-level audio processing functions, but C/C++ is still the language for the most important open source resources.

Logic programming languages have also a high-level and declarative nature. They combine knowledge and rules to reason about the knowledge and infer new insights. This is an important feature to consider a logic framework as underlying mechanism to relate the hierarchy of data involved in multimedia domains. However, they are not as powerful as a computational machine even when it is possible to declare procedures with an execution sequence. There are several works that develop a typed-functional language on a logic framework like lambda-Prolog or Mercury [13]. Mercury embeds computational capabilities in a purely declarative language within a strong typed and moded logic framework. Mercury types and predicate modes provide the compiler with the information to generate efficient code but as environment for declaring and interactive prototype, Mercury is "less forgiving" than Prolog. SWI-Prolog in particular has a very comfortable development environment and hence we use SWI-Prolog.

Our approach results to be more suitable to develop an open platform for research and development. External engines based on open source APIs provide faster computations than any functional or logic/strong-typed language and a suitable developers interface.

3.1 Datatypes controlling workflows

In section 2 we have seen how tasks are strongly characterised by their input/output relationship. The system requires datatypes that guarantee the correct input and output matching. We identify these datatypes by analysing the tasks classification. We also want to look at the Music Ontology specification

[8] as a formal description of music-related concepts including audio analysis terminology. We distinguish the following datatypes to any signal processing system:

- signal: this type represents an audio signal in a physical timeline;
- feature: this type represents some facet of the audio signal properties;
- parameter: this type represents the different variables of the computation context (computations are not only a one-to-one function);
- timestamp: this type represent the reference of a data object to a timeline.

We implement these datatypes in SWI-Prolog as functors (name/arity) that bind together the attributes of a concept (e.g. signal(Channels, SampleRate, Length, [DataIdPerChannel])). These datatypes (with the disadvantage of being a non-flexible model) can be interpreted by any module guaranteeing the correct instantiation pattern of the predicates and ensuring a valid execution.

3.2 Large Binary Objects

Large binary data is present in any multimedia system and needs to be treated in the most efficient and manageable way. The so-called BLOBs are objects wrapping large binary data. The representation of data in SWI-Prolog can be done through lists which can not handle such amount of data (e.g. signals of 1000000 samples) or through SWI-Prolog BLOB terms which are cumbersome as datatype.

We have implemented a mechanism of unique *data identifiers* that are stored in a structure updated at runtime. Each identifier holds the binary representation of an arbitrary large sequence of floating point format data. This identifiers-based system is useful in the following ways:

- It hides the raw binary representations or inefficient SWI-Prolog lists by handy representations of data as atoms: e.g. **__data_435**;
- It implements a look-up database whose status can be easily updated and checked;
- It provides multiple ways to export/import data from the structure through the identifiers. Data can be wrapped into or extracted from a blob term and displayed/read as Prolog list (length restrictions). It allows dumping data into or loading it from a binary file (using the identifiers to keep track of the data);
- It unifies the "raw" data representation allowing the communication at the lowest level between modules;
- One identifier can be used to retrieve data from a previous session as long as we make the data object persist under its identifier and reserve the identifier in the new session.

The following example shows how a decoded audio file can be easily represented by the data identifiers that make easy to work with such a big sequence of data. A data identifier can be used in the same way as the name of a person to identify a data object which is related to other entities within a predicate.

```
?-decode('../myfile.*', Signal).
Signal = signal(2, 44100, 1200000, ['__data_0', '__data_1']).

%we can relieve memory by dumping the data into files
%and loading it later on. Files will have the corresponding id
?-data_out('__data_0').
?-data_out('__data_1').

?-data_in('__data_0'), data_in('__data_1'),
get_frame(signal(2, 44100, 1200000, ['__data_0', '__data_1']), 200, 5,
frame(Ch, Sr, L, [Ch1D, Ch2D])),
data_to_list(Ch1D, List), length(List, Llen).
Ch1D = '__data_2'
List = [6.45, 4.56, 3.23, 9.78, 3.26]
Llen = 5
```

3.3 Processing workflows

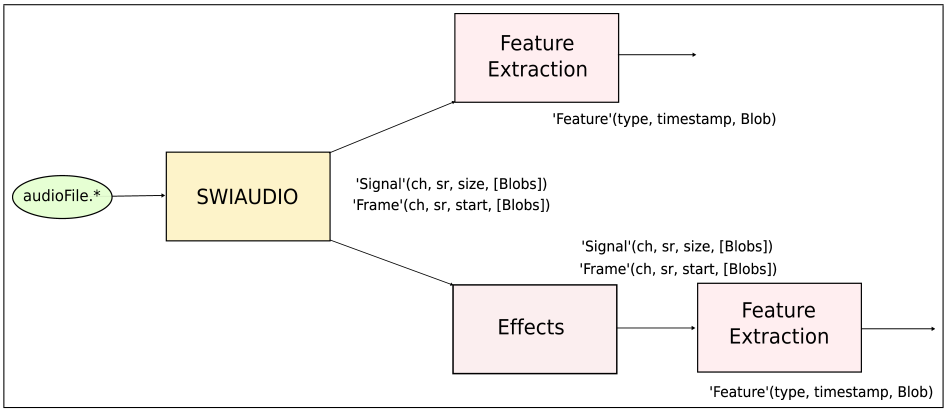

Fig. 1. Example of how to create a workflow: extract the tempo of a signal and a reverberated version of it

Figure 1 shows a possible workflow that is designed by using the functional framework formed of datatypes, a standard format for data and the modules of SWI-Prolog predicates which are described in 4.

4 SWI-DSP modules

Each task module (functional predicates) is composed by lower level modules which offer communication with an external API or library in charge of the

computations. By hosting "plugin libraries" we can extend easily the SWI-DSP vocabulary (new plugins can be added without code modifications). These low level modules are not constrained by datatypes (every API deals with its own concepts) so they require a precise understanding of the API and the interface.

4.1 swiaudio

This module is in charge of providing audio data to the system. This module is splitted in two sub-modules: **swiaudiosource** and **swiaudiodata**. The former is a module for decoding which interacts with C decoding libraries (mad, soundfile, fishsound, oggz and faad) to extract a *Signal* term given and audio input file supported by the libraries. This process is triggered by a simple call to the predicate *aspl_decode(+InputFile, -Signal)* which recognises the file extension and calls a specific library allowing a fast decoding process and even more important an independent term representing the encoded data in a standard way for the supported formats: mp3, ogg, mp4, aac and wav.

The latter is a module which defines utilities typically required in DSP. The most important set of predicates is dedicated to framing. The rest of predicates are mainly oriented to extract information from a Signal or a Frame term.

4.2 feature extraction

This module defines a set of predicates and rules (specific API routines for the computations) for feature extraction. As we have seen in §2, the module is characterised for a clear input/output relationship. So we can declare a simple predicate to extract the tempo of an audio signal: tempo(+Input, -Tempo) being Input an instantiation of the Signal datatype and Tempo a list of instantiations of the Feature datatype. This predicate, that returns a tempo-related list of features given an input signal, is indeed triggering a specific computation tool, so we could select the specific tool by extending the predicate: *tempo(+Input, +Tool, -Tempo)*.

We rely on the Vamp Plugin API as it offers an efficient plugin and host implementation. Certainly, there is a limitation in the sense that any feature extraction algorithm we may want to use has to be implemented as Vamp Plugin. As a counterpart, it is easier to write an algorithm as Vamp Plugin than to interface many APIs or particular algorithm implementations.

The sub-module **swivamp** is a communication layer based on the SWI-Prolog foreign interface allowing us to query the plugin information, set up the plugin context for the transform, execute the plugin given a Signal term and retrieve the output features for the plugin according to the lifecycle specification. Plugins are wrapped in a similar fashion than the audio data so we can use a plugin instance representation (a plugin id pointing to a plugin instance existing in memory), so that we can have concurrent plugins working at the same time. The previous tempo/3 predicate is declared as this set of rules (defined with a procedural meaning):

```
tempo(Input, Tool, [feature(Type, Timestamp, Feature)|...]):-
     Input = 'Signal'(Ch, Sr, L, PCMs),
     Tool = 'Vamp',
     vamp_feature_of(tempo, Input, Tempo).

vamp_feature_of(Type, Signal, WholeFeature):-
     select_plugin(Type, PluginKey, Output),
     vamp_outputs_for(Signal, PluginKey, Output, Features),
     flatten(Features, WholeFeature).

vamp_outputs_for(Signal, PluginKey, Output, Feature):-
     vmpl_load_plugin_for(PluginKey, Signal, Plugin),
     set_blockSize(Plugin, Block),
     set_stepSize(Plugin, Step),
     get_channels(Signal, Channels),
     vmpl_initialize_plugin(Plugin, Channels, StepSize, BlockSize),
     vamp_compute_feature(Signal, Step, Block, Outputs, Plugin, WholeFeature).

vamp_compute_feature(Signal, Step, Block, Outputs, Plugin, Features):-
     get_samples_per_channel(Signal, L),
     findall(F, vamp_process_signal(Signal, L, Step, Block, Output, Plugin, F), FSet),
     get_sample_rate(Signal, SampleRate),
     vmpl_remaining_features(Plugin, L, SampleRate, Outputs, Remaining),
     append(FSet, Remaining, RawFeatures),
     delete(RawFeatures, [], Features).

vamp_process_signal(Signal, L, StepSize, Block, Out, Plugin, Features):-
     set_limit_framing(L, StepSize, Limit),
     set_framing(StepSize, L, Limit, Start),
     vmpl_process_block_framing(Plugin, Signal, Start, Block, Out, Features).
```

The first rule selects the tool for the queried feature. The following one identifies automatically the specific plugin within the Vamp API in charge of extraction the "tempo" (and passes a term to unify the returned feature). The third predicate sets up the plugin for the processing (in this case with default parameters). The predicate vamp_compute_features acts as host retrieving and organising the extracted features which are indeed computed by the last predicate. This last predicate configures the framing and calls the swivamp interface that rapidly communicates and executes the plugin (the efficiency of the extraction is preserved). The features are returned as a list of terms represented using the feature datatype: feature(type, timestamp(start, duration), '__data_id').

4.3 audio processing

This module defines any computation which outputs a Signal term. It comprises any sort of effect or filtering over a signal. It reproduces a real time processing

by outputting processed frames as they are computed (non-deterministic call to a processing predicate) or stores the processed blocks to return and altered version of the input signal.

The module swiladspa offers communication with LADSPA which is the most generic and spread plugin API for general audio processing. The communication layer allows us to query the plugin description, control the set up of the plugin (control and audio ports) and retrieve the processed data. Through predefined routines we can offer a more handy access to the plugin or even declared predicates just defined by an input/output relationship.

4.4 swiweka

The WEKA API is interfaced through the package JPL available for SWI-Prolog defining a complex Java-Prolog interface to read/write ARFF files, load ARFF files, instantiate classifiers, execute classification and retrieve the classification scores of the dataset. This communication layer offers by itself a really cumbersome and API-dependent module. It requires semantics and interpretation rules to become into a useful logic-framework for audio data classification as described in the next section.

5 DSP front-end

So far we have described how SWI-DSP provides a vocabulary of logic predicates and datatypes for performing audio analysis, but this aproach is still inappropiate for some tasks. We have organised the modules in communication layers and task-oriented modules, but there is a trade-off between understandability and rich functionality. Use of the computational facilities provided by the various modules requires quite detailed knowledge of their capabilities as well as familiarity with Prolog. We do not expect this type of expertise from most of our potential users. Hence, we require a front-end that is closer to the application domain, providing an interactive "laboratory" at a level of abstraction closer to that expected by the signal processing practitioner.

We define some "DSP scenarios" that reflect the researching activity. This scenarios define the necessary patterns and datatypes (implemented as predicates and functors) to allow the user to actually perform high-level research preserving the necessary flexibility. The elementary scenarios are:

- Transform: This scenario provides a pattern to apply any computation over an audio signal. It specifies by a datatype the framing parameters, the running engine, the engine configuration and parameters, the sample rate (if variable), the type of transform and the domain specifications. These factors are captured with a functor used to link input/output (with full flexibility) in transform(+Input, +Transform, -Output)
- Classification: This scenario is provided for a proper setup of classification. This scenario specifies the classification domain, the categories (names, types

and ranges), the classification schema and the training set. It also provides a pattern for the input data set which is interpreted along with the classification environment to return the output distribution.

The patterns and functors are interpreted by hardcoded prior knowledge which starts up the subsequent steps to provide the queried output. More complex scenarios can be created in order to achieve higher or broader semantics for research activity.

This logic-based framework for DSP (Fig. 2) develops an interactive computational engine that exploits logic resources and their declarative nature (but also relies on procedural declarations). Such an engine can be embedded in other environments (by SWI-Prolog interfaces to other languages) developing graphical interfaces, visualisation applications or data management systems. Further knowledge can be achieved by combining different sources of data under a single declarative framework as explained in section 6.

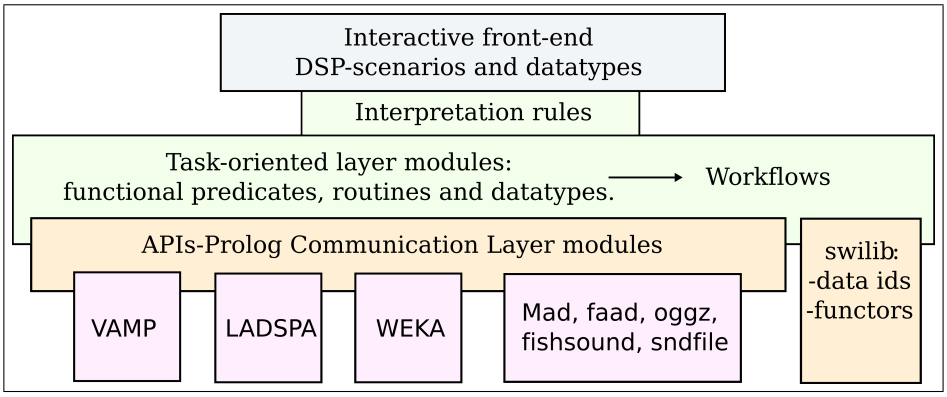

Fig. 2. Architecture of SWI-DSP

6 Data management and computation reasoning

We still need to address what is probably the most important issue in constructing the proposed "laboratory workbench". A primary aim of such a system is to manage complex relationships between many items of data, some of which may be large and temporally structured (e.g. audio signals). Some of the relationships are functional, in the sense that some items are computed from others using signal processing functions provided through the foreign language interfaces described above. Some relationships are not functional, eg, several human experts might label a set of signals by musical genre, resulting in many possible labellings for each signal. This type of knowledge management requires mechanisms for persistent storage of data objects and the procedures that were used to

compute them, as well as effective management of a local database, side-effects of computations, information from other sources such as manual annotations or external databases, and a high-level interface for specifying and scheduling potentially complex workflows involving many computations and combining data from different sources.

We do not develop such manager within SWI-DSP, but we are developing a Prolog-based system that does address most of these concerns [3]. Information pertaining to many types of object, including signals, computed results, musical scores, composers etc. is represented uniformly using a system of predicates, some of which are backed by a relational database. When time-consuming computations are involved in satisfying a particular predicate, the results can be persistently stored along with a record of the bindings for that predicate. If, at a later time, the same computation is requested using the same input parameters, the result is retrieved from the database instead of being recomputed. It is also possible to look up the input parameters that resulted in a given value being computed, and to list all previously recorded computations of a given predicate.

This is in some ways similar to the "tabling" mechanism provided by some logic languages, notably XSB [4]. The previous implementation of this knowledge management system included a module that allowed the use of MATLAB to implement computationally intensive functions. SWI-DSP can now provide a similar capability to use VAMP plugins, LADSPA plugins, and WEKA code. The data identifier mechanism is compatible with the overall system as the identifiers can be stored in a database and associated with files containing the actual binary data.

The other half of the problem, that of workflow management, is addressed using a implementation of *transaction logic* [1]. Transaction logic allows the specification, in a declarative way, of sequential or parallel compositions of actions, possibly involving database update, side-effects, and interaction with a user or other external device. The primitives offered by the transaction logic formalise the notion of calls with side-effects and provide an alternative to some of the more procedural styles of Prolog usage, such as failure driven loops, making it easier to specify complex workflows without sacrificing a clear declarative semantics. Transaction logic applied to complex data/computational environments has been successfully applied to develop "virtual laboratory workbenches" [2].

SWI-DSP combined with a data manager based on Concurrent Transaction Logic (CTR), which also controls concurrent processes, develops a multimedia knowledge system to drive research with semantics and a logic formalism exploiting reasoning and declarative definition of computations and providing an interactive source of knowledge.

Currently, we have another implementation of such a manager oriented to interact with the *Semantic Web*. This project, so called Henry[1], aggregates processing workflows and structured data available on the Semantic Web and publishes back the results it is able to derive. It can interpret queries and express data through a Notation3 [16] parser/entailment module. Notation3 is a syntax

[1] Source code project at `http://code.google.com/p/km-rdf/`.

for the Semantic Web format which provides a way to reproduce a deductive model based on triples (Subject, Predicate, Object) and rules. SWI-DSP provides Henry with signal processing services. A working instance can be found at `http://dbtune.org/henry/`.

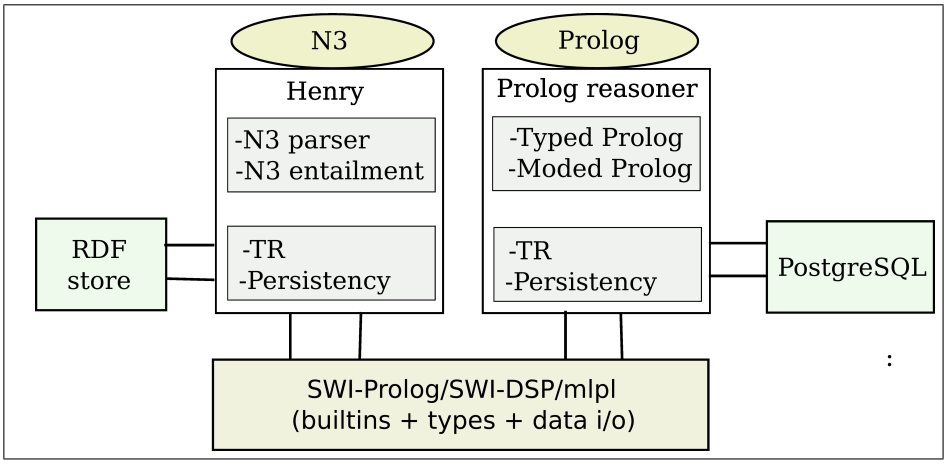

Fig. 3. Collaboration diagram between SWI-DSP, the Matlab interface and the high-level reasoning implementations

7 Conclusion and Future work

We have presented a logic-based engine that provides digital signal processing services under an interactive and declarative framework which is able to make data inferences. Such an engine solves many of the problems presented by current approaches to develop an integrated platform for signal processing and multimedia analysis in general. This system is aimed at providing services to high-level reasoners capable of integrating and reasoning about heterogeneous data available in different data systems with the same declarative nature.

We expect of SWI-DSP to be a useful workbench for DSP research and also a platform for development enriching the community resources. The proposed system and future extensions are expected to become a powerful workspace to integrate and analyse heterogeneous multimedia data providing enhanced services to a wide range of users and contributing to extend a common source of knowledge.

Near future work and development on SWI-DSP are oriented to extend its vocabulary and capabilities. First, by interfacing more open source APIs to cover the implementation of more DSP tasks (not necessarily restricted to the audio domain). Also, by supporting symbolic data (MIDI) input which is a really

important source of audio knowledge and providing analysis modules for such an input.

In general, any source of knowledge, processing tool or logic paradigm which allow us to integrate and infer new knowledge is a target in order to reach higher-levels of abstraction to understand the nature and effects of the signals.

References

1. Bonner, A.J., Kifer, M.: Concurrency and Communication in Transaction Logic. In: Joint International Conference and Symposium on Logic Programming, pp. 142–156. Bonn (1996)
2. Bonner, A.J., Shrufi, A., Rozen, S.: LabFlow-1: a Database Benchmark for High-Throughput Workflow Management. In: 5th International Conference on Extending Database Technology, pp. 463-478. Springer-Verlag, Lecture Notes in Computer Science, volume 1057. Avignon (1996).
3. Abdallah, S., Raimond, Y., Sandler, M.: An ontology-based approach to information management for music analysis systems. In: 120th Audio Engineering Society. Paris (2006).
4. Sagonas, K., Swift, T., Warren, D. S.: XSB as an efficient deductive database engine. In: ACM SIGMOD international conference on management of data, pp. 442-453. New York (1994).
5. CLAM, `http://www.clam.iua.upf.edu/`
6. Music Analysis, Retrieval and Synthesis for Audio Signals. `http://marsyas.sness.net/`
7. McKay, C.: jMIR. `http://jmir.sourceforge.net/`
8. Raimond, Y., Abdallah, S., Sandler, M., Frederick, G.: The Music Ontology. In: 8th International Conference of Music Information Retrieval. Vienna (2007)
9. Henry. `http://dbtune.org/henry/`
10. Linux Audio Developer Simple Plugin API. `http://www.ladspa.org/`
11. Cannam, C.: Vamp Plugins. `http://www.vamp-plugins.org/`
12. WEKA home site `http://www.cs.waikato.ac.nz/ml/weka/`
13. Mercury Home Project `http://www.cs.mu.oz.au/research/mercury/index.html`
14. Haskell `http://www.haskell.org/`
15. SWI-Prolog Home `http://www.swi-prolog.org/`
16. Notation3 `http://www.w3.org/DesignIssues/Notation3`