Jargons: Experimenting Composable Domain-Specific Languages

Frédéric Peschanski*
Laboratoire d'Informatique de Paris 6 (LIP6)
Frederic.Peschanski@lip6.fr

ABSTRACT

We present in this paper an exploratory research work about domain-specific languages (DSL). Bringing together structured documentation and programming language concepts, we provide a framework for the design and implementation of jargons, our terminology for DSLs. In our approach, we put the focus on the composition issue. First, jargon definitions are composable at the structural level. Moreover, composition points can be put in finer-grained construct descriptions. We also unify the operational representation through prototypes, providing many interesting properties regarding composition. The architecture of the resulting system is highly reflective - following a two-level meta-tower pattern - and oriented toward evolvability.

1. INTRODUCTION

It is of common use to consider *programming* as an esoteric task handled by weird people (so-called *programmers*) using strange languages to explain computers what they should do. On the other hand, "normal" people (so-called *end users*) use software tools (generally written by programmers) to instruct computers. Besides this common aim, it is considered as reasonable to keep these two worlds as isolated as possible. However, the situation is slowly but inexorably evolving [5]. We see the concept of *domain-specific language* (DSL) as an interesting mean to fill this gap since, in our opinion, it is adequately located between programmers and end-users. The authors of [16] propose the following definition:

> A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

*Thanks to Christian Queinnec, Jean-Pierre Briot, Fabrice Kocik and the anonymous reviewers for their useful comments on these work and paper.

The idea of our experimental work is to elaborate a framework for the design and implementation of composable domain-specific languages. In our terminology, a DSL is called a *jargon*, hence the name of our system: Jargons. There is no restriction on the purpose of a given jargon, besides the fact that it must be created within the framework of our system. Currently, the available jargons (listed in table 1) covers programming styles (functional, logical, object-oriented, etc.), programming features (threads, modules, exceptions, etc.) and expert languages (for structured documentation or jargon definition).

Through this experiment, we put the focus on the *composition* of jargons. The first motivation is that new jargons could be defined using other (less) specialized languages so that the wheel is not to be reinvented every time. Figure 1 shows various compositional relations between some available jargons (see section 4 for a discussion on this issue). We also think that a given problem might benefit from the availability of multiple specialized jargons that are composable with each other. The examples described in this paper provide, in our opinion, a good illustration of this idea.

As in [3], *embedment* in a general-purpose and "open-minded" mother language represents the fundamental property of our system. The programming language Scheme [4] is ubiquitous in our approach : it is used at the same time to implement the basic architecture of our system and to develop interpreters for some fundamental jargons (like OO dialects). Most jargons eventually produce Scheme code at interpretation time. Moreover, we will see that Scheme can be used as a composable jargon from other jargons constructs.

We decompose the problem of designing, implementing and composing Jargons in three major subproblems: definition of jargons constructs, interpretation and operationalization concerns. In order to define jargons constructs, our system introduces a concept similar to the DTD of structured documents in SGML or XML. Interpretation corresponds, in our case, to the analysis of the jargons uses (considered as special forms), followed by the execution of Scheme code which itself optionally produces standard Scheme code. It is then a mix of pure interpretation and runtime (without bootstrap) or preprocessor-time (with bootstrap) code generation. At the operational level, we introduce *prototypes* to address the issue of mixing jargons "effects". Roughly, prototypes can be defined as closures with message-based communications. This forms a good basis, in our opinion, to discuss the semantical concerns of jargons composition at the operational level while this is not the main concern of our pragmatic approach.

In order to support the process of jargons implementation, our system is architectured following a two-level meta-tower, conceptually very close to the architecture of the Maciste system [8]. This relatively complex architecture provides some interesting features and insights concerning automation and evolution of the jargons design process. We mainly focus on *bootstrap* and *monitoring* properties at this level.

In section 2 of the paper, we present, through a complete example in the domain of Multi-Agent Systems (MAS) [1] and robotics, the methodology supported by Jargons regarding the design of domain-specific languages. The implementation support is then described in section 3. The composition issue is addressed in section 4. First, a more complex MAS Jargon is proposed as an extension of the jargon developed in section 2. Next, we present a more generic way to compose definitions. Finally, we see how prototypes help the compositional needs at the operational level. We then describe in section 5 the basic architecture of the system. A panorama of related work (section 6) and a conclusion (section 7) end the paper.

2. DESIGNING NEW JARGONS

When considering the construction of a new jargon, our framework encourages a pragmatic point of view as illustrated by the case study described in this section. Our objective here is to develop an agent-based jargon called SRA (Simple Reactive Agents) and use it to describe simple robotics simulations.

While it could be argued than our examples reflect some object-orientation background with mainly syntactic differences, we would object that this syntactical specialization *makes the difference* so that the entities we manipulate are called *agents* and not *objects* anymore. It is then important to approach our examples from a *user's point of view* (someone interested in the description of agents), rather than from an *implementor's point of view* (someone interested in the operationalization of agents, probably through objects). As a matter of fact, the agents we consider in section 4.3 of this paper do not really resemble objects, in our opinion.

2.1 Jargon purpose

Identifying a precise domain and knowing *a priori* what will be the purpose of a given jargon is, as expected, the first step of the process towards its elaboration. Here is a non-exhaustive list of jargon categories:

- **Programming styles**: we see the possibility of mixing styles as a very important feature and objective of the jargons system. Jargons falling in this category concern object-orientation (with or without inheritance, active objects, etc.), component-based programming, logic-based programming and so on.
- **Programming features**: this concerns jargons that support programming features through adequate linguistic constructs , such as concurrency and distribution management, assertions and invariant checking, exceptions management, and so on.
- **Expert jargons**: all the jargons that address problems in domains not directly related to programming are concerned here. In this paper, we present a few prototype jargons aimed at simple collective robotics interactions, we see them at an intermediate between programming styles and expert jargons. We also worked on some structured documentation jargons such as SCHEMEDOC, a jargon for Scheme source code documentation.

2.2 Jargon uses

An issue of prime importance after having roughly delimitated the domain is to anticipate on the expected uses of the jargon. The major objective here is to elaborate the corresponding syntactical constructs or *jargon uses.* Since we are aiming at a transparent and integral embeddement in the Scheme environment, jargon uses must be regular *s-expressions*, following the pattern:

```
(jargon:construct contents)
```

In our robotic study, each robot will be described as a very simple agent. They will perform perception (through sensors) and actions. One of the action, namely behavior, describes the general activity of the robot.

When designing the jargon constructs, it is also important to know where will be the *inner composition points.* In our example, the action bodies represent ideal composition points: we would like any Scheme expression to be usable here. Composition is addressed more thoroughly in section 4.

To summarize, a typical use of the SRA jargon would be:

```
(sra:agent SillyRobot
  (sensor s1 Touch)

  (behavior
    (if s1
      (begin (reverse) (turn-left 90))
      (move)))

  (action reverse () ...)
  (action turn-left (angle) ...)
  (action move () ...))
```

The above statement defines a robot type SillyRobot, which provides a simple touch sensor and a minimalistic behavior. The robot always move forward except when it touches something with its sensor. In this case, the robot reverses its engine and turns on its left by 90 degrees.

2.3 Jargon effects

The next step is to define what will be the impact of the jargon constructs on the underlying Scheme interpreter. There are several possible ways to interpret a jargon use, we can distinguish the following categories:

- **functional effect**: the construct is interpreted as a pure function, without any side effect.
- **side-effect**: the interpretation leads to the modification of a shared resource such as the global environment, a file on a disk, a network resource and so on.
- **code generation**: as a special case of the previous category, an interpretation can result in the generation of some support code, modifying the interpreter environment.
- **entity-orientation**: in many situations, the generated code will itself lead to the generation of some

new code at execution time. This is the case of all jargons that describe software entities like objects, components, agents, finite automata and so on, all represented by prototypes at execution-time.

Our agent-oriented jargon is mainly concerned by the last category. The declaration of the robot type SillyRobot leads to the generation of a constructor function: sra:make-SillyRobot whose purpose is to create new simulated robots:

```
(define r2d2 (sra:make-SillyRobot))
```

2.4 Prototypes as entities

As stated above, entities are represented at the operational level by *prototypes*, providing the following properties:

- **internal state**: this is a set of variables that can change during the prototype's life-cycle.
- **functionalities**: a set of functions can be performed by the prototype, for example by reading and writing data from and to its internal state.
- **message-based communication**: prototype's state and functionalities cannot be directly accessed from the outside world. To allow this, a message-based communication model is provided.

So, one of the fundamental question here is to define precisely the state, functionalities and messages exposed by the prototypes generated for a given jargon. The SRA prototypes contain the sensor states, functionalities are actions and behavior, messages are mainly generic (plus sensor fetching).

To send a message to a prototype the syntax will always be:

```
(prototype 'message arg1 arg2 ...)
```

Let's see if our r2d2 knows of which family it is a member of:

```
(r2d2 'get-type)
==> SillyRobot
```

Then, we can test the touch sensor s1 like this:

```
(r2d2 'get-s1)
==> #f
```

This indicates that the robot does not sense anything. Finally, we can simulate the activity of the robot through the step functionality:

```
(r2d2 'step)
(r2d2 'get-s1)
==> #t
```

Here, the sensor informs the user that the robot has touched something.

2.5 Summary

Through our case study in the domain of collective robotics, we asked the basic questions related to jargon design. Let us summarize the important questions to ask (preferably in order) and the answers corresponding to our case study :

1. *What is the purpose of the jargon ?* collective robotics
2. *What are typical jargon uses ?* agent constructs (see section 2.2), environment constructs (not shown).
3. *What are the resulting jargon effects ?* generation of prototype generators.
4. *What are the properties of the generated prototypes ?* internal state contains sensors informations, functionalities are actions and behavior, message are system messages and sensor fetchers.

In table 1, we show the current list of implemented jargon and the corresponding categories (programming styles, features or expert jargons). The last two jargons are of primary importance. First, scheme is at the same time the underlying language of the system and also one of the available jargons. To achieve a meta-circular description of the system, the jargon used to describe the other jargons - namely, METAJARGON - is expressed in itself, as explained in section 5.2.

3. IMPLEMENTATION SUPPORT

With in one hand the design of a new jargon and in the other hand a computer running a Scheme interpreter or compiler and the Jargons framework, we can now start the implementation phase. We describe in this section three aspects of the framework : jargons definitions, evaluation model and interpretation.

3.1 Jargon definitions

In order to define a new jargon, we propose the METAJARGON meta-language that is close to the DTD language of XML. However, unlike XML where a DTD is not itself an XML document, a METAJARGON definition is also a regular Jargon definition. Provided with a correct jargon definition, the system is able to generate automatically a specialized validating parser and also to bind the specific jargon interpreter (see section 3.3). Let's define the SRA Jargon:

```
(JARGON sra "agents/sra.sld"
 (DESCRIPTION "Simple reactive agent model")
 (AUTHOR "Fred Peschanski")
 (VERSION "0.2"
 (ROOT agent (TERMINAL kind SYMBOL)
  (* (NODE sensor (TERMINAL name SYMBOL)
       (TERMINAL type
         (ENUM Touch Temperature Color Rotation))))
 (NODE behavior (SCHEMEDEF))
  (* (NODE action (TERMINAL name SYMBOL)
      (TERMINAL params DOTTED-LIST-OF-SYMBOL)
      (SCHEMEDEF))))))
```

This definition first gives some *meta-data* about the jargon: a short description, its author and version. Then, the ROOT keyword introduces the only *root construct* of the SRA jargon[1]: sra:agent. We define it to contain a *terminal* (TERMINAL keyword) giving its kind (as a symbol) followed by a possibly empty succession (* operator) of sensors. A sensor is a non-terminal (NODE keyword) that contains a name and a category (from a finite set of possibilities). Then, a behavior node must be indicated, containing a very important sub-node: (SCHEMEDEF). This states that any correct Scheme expression can be used to describe the behavior's body. This is one of the major composition feature

[1]Jargons with multiple root constructs can of course be elaborated.

Name	Category	Purpose	Effect
MOON & al.	Style	Object models (with or w/o inheritance...)	prototypes
SCOPE & al.	Style	Component-based jargon (GUI framework,...)	prototypes
SRA & al.	Style	Agent-oriented jargons (reactive, cognitive...)	prototypes
SCHELOG	Style	Prolog-like embeddement	functional
MODULE	Feature	Module management	side-effects
THREAD	Feature	Thread management (semaphores, condition variables, monitors)	side-effects + prototypes
STRUCTDOC	Expert	Structured documentation	side-effects
SCHEMEDOC	Expert	Scheme code documentation	side-effects
SCHEME	Style	Scheme in jargons	see R5RS
METAJARGON	Expert	Jargon of jargons	code generation

Table 1: List of currently implemented jargons

Category	Purpose	Contents
ROOT	Root constructs, top-level jargon uses	Name and non-empty sequence of subnodes
NODE	Non-terminal node	Name and non-empty sequence of subnodes
TERMINAL	Terminal node	Name, type (basic, user, enum, exact match, exclusion list)
EVAL	Pre-evaluated terminal	Same as above
*	Zero or many operator	Sequence of subnodes to match
+	One or many operator	Sequence of subnodes
?	Zero or One operator	Sequence of subnodes
|	Alternative operator	Sequence of alternative subnodes
SCHEMEDEF	Scheme definition recognizer	Sequence of expressions
SCHEMEONEDEF	Scheme recognizer	Only one expression

Table 2: Node categories in METAJARGON

of the framework. Actions also accept a parameters list defined as a dotted list of symbols. In table 2, we define all the node categories available in METAJARGON.

It is finally important to notice the type constructs in the node definitions (such as SYMBOL, ANY, and so on). Since the framework is open regarding type definitions and interpretations, *type checking* can be introduced and specialized very early in the development process.

3.2 Evaluation model

Jargon uses are interpreted in Scheme as *special forms*, their evaluation model is specific. Unlike standard forms, the arguments for jargon constructs are not evaluated by default. The idea here is that jargon constructs should be more *declarative* than regular scheme constructs[2]. However, there exists a specific METAJARGON construct, (-> defs), which inverts the evaluation model. Suppose for example that we would like the kind of agent to be generated by program. We expect to write something like this:

```
(sra:agent (generate-agent-kind)
  ...)
```

This statement will be rejected by the Jargon parser since (generate-agent-kind) is not a correct symbol (and the definition is expecting a symbol value).

To evaluate the argument before analyzing the resulting expression, we can use:

```
(sra:agent (-> (generate-agent-kind))
  ...)
```

[2]It is in our beliefs that some jargon users will not even *care* about evaluation.

If the result of the function call is a correct symbol, then we will obtain the expected effect, the kind of the agent being programatically generated.

In some situation, it can be interesting to express at the jargon level that a particular terminal should be pre-evaluated. Suppose for example that we want to enforce the first case (generated agent kind). To that effect, we can use the EVAL keyword to replace TERMINAL in the jargon definitions:

```
(JARGON sra "sra.sld"
  ...
  (ROOT agent (EVAL kind SYMBOL)
  ...))
```

From now on, if we want to use a symbol directly as in our SillyRobot agent, we will have to write:

```
(sra:agent 'SillyRobot
  ...)
```

3.3 Jargon Interpreters

Concerning the interpretation phase, regular scheme code is used to describe the semantics of jargon constructs.

The interpreter function for the SRA jargon analyzes the agent definitions by selecting and filtering its internal components : list of sensors, states and actions. From these informations, the interpreter then produces a macro called sra:make-X (where X is the agent kind) whose purpose is to generate on demand a prototype closure. The general form of a prototype is:

```
(letrec ((self
  (let (<internal state>
    (letrec (<functionalities>)
      (lambda (message . args)
```

```
    (case message
      ((<message-pattern>) <message-action>)
        ...
      (else (error:raise "Message not handled")))))))))
  self)
```

This lambda form encloses a self reference, the definition of the internal state as a succession of let variables, a set of functionalities in a letrec block followed by the lambda-definition and the message selector block (in a case form).

In the case of the SillyRobot agent kind, the generated prototypes will have the following form :

```
(letrec ((self
  (let ((s1 (sensor:make-sensor 'Touch))
    (letrec ((behavior (if s1 ...))
             (reverse (lambda () ...))
             (turn-left (lambda (angle) ...))
             (move (lambda () ...)))
      (lambda (message . args)
        (case message
          ((get-type) 'SillyRobot)
          ((get-s1) (s1 'fetch-sensor))
          ((reverse) (apply reverse args))
          ((turn-left) (apply turn-left args))
          ((move) (apply move args))
          (else (error:raise "Message not handled")))))))))
  self)
```

While the SRA agents do not embed conceptually any internal state, their representation prototypes are stateful in order to manage the agent's sensor informations. It is important not to mix up conceptual forms (e.g. agents) with their operational representations (prototypes).

It is of course impossible to anticipate all the needs in term of jargon interpretation. However, the Jargons framework proposes as described in section 5.2 several generic functionalities to support the implementation of interpreter functions.

4. COMPOSITIONAL ISSUES

The key interest in the Jargons approach concerns the issues related to composition. There are different things that we would like to compose in a DSL framework: definitions, uses and prototypes.

Figure 1 shows an example of the interest in composing jargon definitions and implementations. The OO dialects, such as microon or moon are implemented in pure Scheme while the latter reuse important parts of the former through definitional composition. Agent-based dialects, as described in this paper, are also intrinsically related to the OO dialects since we use them to build the prototype generators. It is however notable that an agent category denotes a set of classes, there is no direct correspondance between agents and objects.

4.1 Composing jargon definitions

In order to illustrate the composition of jargons definitions, we will take the example of the RAIS[3] jargon whose purpose is to implement a stateful agent model. This jargon introduces a few new constructs: state to provide the internal state definition and initialize to provide the internal state contents at instantiation time. The description of the sensors are also delegated to a specific jargon. We can

[3]Reactive Agents with Internal State.

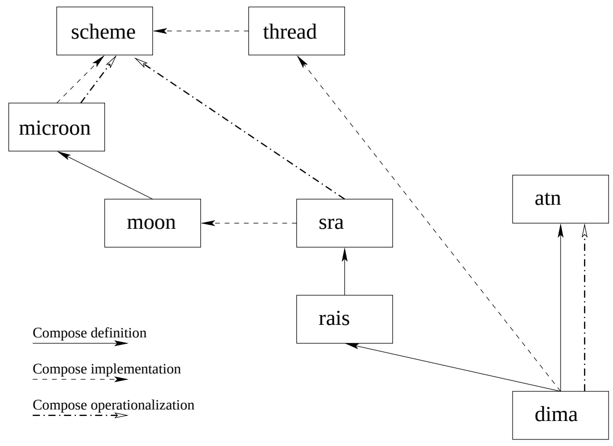

Figure 1: Jargon compositions

for example define an agent which is more and more happy when it encounters something blue:

```
(rais:agent ArtisticRobot
  (sensor s1 Touch)
  (sensor s2 Color (Discrete BLUE BLACK YELLOW))

  (state happiness 0)
  (state name)

  (initialize (agent-name)
    (set! name agent-name))

  (behavior
    (if s1
      (begin  (if (eq? s2 'BLUE)
                  (+ happiness 1))
              (turn-left 90))
      (move)))

  (action reverse () ...)
  (action turn-left (angle) ...)
  (action move () ...))
```

The definition of the RAIS jargon can then be seen as a composition of the SRA and SENSOR jargons :

```
(JARGON rais "agents/rais.sld"
  ... meta-data ...
  (USES sra "agents/sra.sld")
  (USES sensor "agents/sensor.sld")
  (ROOT agent (TERMINAL kind SYMBOL)
    (*  (NODE sensor (TERMINAL name SYMBOL)
          (UREF sensor:sensor-description)))
    (*  (NODE state (TERMINAL name SYMBOL)
          (? (EVAL init ANY))))
    (?  (NODE initialize
          (TERMINAL params DOTTED-LIST-OF-SYMBOL)
          (SCHEMEDEF)))
    (UREF sra:agent-behaviour)
    (*  (UREF sra:agent-action))))
```

After the usual meta-data, a new keyword, USES, is employed to explain that the current definition reuses some constructs from external definitions, here the SRA and SENSOR jargons. The UREF keyword can be used to make references to constructs in these external definitions (we call them *use references*).

The path specification follows the pattern:

```
(UREF jargon:root-node1-node2 ... )
```

For example, the behavior specification of a RAIS agent is exactly similar to the one of the sra:agent-behavior so we can indicate (**UREF** sra:agent-behavior) instead of having to give again the full definition.

More than just simplifying the definition of new jargons, this composition scheme reduces greatly the implementation of the corresponding interpretation function. The idea is that in general, the interpretation function is decomposed following the structural pattern so specific functions are in charge of specific node types. It is then possible to reuse directly these specific interpretation functions in order to build the resulting interpreter. For example, the interpreter functions for sensors or actions do not have to be redefined in the RAIS jargon. The more composition is exploited, the more interpreter implementation is simplified.

4.2 Composing jargon uses

As we pointed out in section 3.1, the (SCHEMEDEF) construct represents a very important part of jargon definitions regarding composition. The example we develop in this section is a more complex RAIS agent, a GarbageRepartitor robot. What we would like to do is to associate a knowledge database to this robot and use it to define its behavior. The purpose of the ResourceRepartitor robot is to find resources in the environment and try to give them to ResourceCollector robots. Resources and robots are distinguished by size and color, thanks to the corresponding sensors. To develop the knowledge database, we provide in Jargons a prolog-like jargon named SCHELOG[4]. Thanks to the (SCHEMEDEF) construct, we can mix functional and logic code:

```
(rais:agent ResourceRepartitor
  (sensor fetch-size Size (Discrete NOTHING SMALL LARGE))
  (sensor fetch-color Color (Discrete GREEN YELLOW))

  (state resource #f)

  (state sighted-object
    (schelog:relation ()
      (rule (relate 'resource) if (fetch-size 'SMALL)
                                and (fetch-color 'YELLOW))
      (rule (relate 'robot) if (fetch-size 'LARGE)
                             and (fetch-color 'GREEN))
      (rule (relate 'nothing) if (fetch-size 'NOTHING))
      (fact (relate 'unknown))))

  (state behavior-selection
    (schelog:relation ()
      (rule (relate give) if (sighted-object 'robot)
                          and (%is #t resource))
      (rule (relate search-robot) if (sighted-object 'nothing)
                                  and not (%is #t resource))
      (rule (relate search-resource) if (sighted-object 'nothing))
      (rule (relate take) if (sighted-object 'resource)
                          and (%is #t resource))
      (fact (relate escape))))

  (behavior
    (apply (schelog:which (X) (behavior-selection X)))))

  (action give () ...)
  (action search-agent () ...)
  (action search-resource () ...)
  (action take () ...)
  (action escape () ...))
```

[4] prolog-like jargon is a port of the Schelog system by Dorai Sitaram [13].

The sighted-object predicate infers an object type (resource, agent, unknown or nothing) from data fetched by the sensors. The behavior-selection predicate then relates a particular action to the sighted object information. For example, if the robot senses a resource, it can take it if the resource flag is true. When the robot senses another agent, then it may give the collected resource, and so on. The behavior then only consists in trying to unify behavior-selection with a variable X.

From a procedural description of the robot's behavior (describing how the robot must behave), we obtained a more declarative one (focusing on what the robot should do). We are convinced that such mix of programming styles is an important step towards the design and implementation of very high-level *expert languages*.

4.3 Composing prototypes

In order to illustrate the interest in unifying the operational form of the jargon entities as prototypes, we will use the example of another agent-oriented Jargon, named DIMA [2]. The agent model of DIMA enforces the separation between the definition of the agent and the definition of its behavior. To describe an agent's activity, developers must define an *augmented transition network* (ATN). We provide a jargon for the definition of such ATNs, like the one shown in Figure 2, representing a subset of the behavior of the ResourceRepartitor robot kind. This network can be described in a declarative way, using the ATN jargon:

```
(define RRbehavior
  (atn:network
    (state has-resource)
    (state no-resource)

    (transition from has-resource to no-resource
               if (sighted-object 'robot) action (give))

    (transition from no-resource to has-resource
               if (sighted-object 'resource) action (take))
```

Compared to the RAIS version, the definition of the ResourceRepartitor robot in the DIMA jargon is greatly simplified since the behavior is now separated (note also that the resource flag is not needed anymore). The resulting statement looks like this:

```
(dima:agent ResourceRepartitor
  (sensor fetch-size ...)
  (sensor fetch-color ...)
  (state sighted-object
    ...)
  (action give ...)
  ...)
```

Now, at the operational level, we have to select the behavior of the agent:

```
(define my-agent (dima:make-ResourceRepartitor))
(my-agent 'set-behavior RRbehavior)
```

And later, in the agent system, we will have to initialize and run the embedded ATN like this:

```
(myAtn 'initstate 'no-resource)
(myAtn 'step)
```

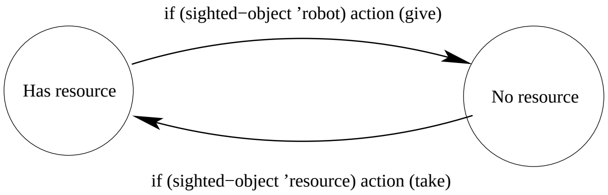

Figure 2: An augmented transition network for an agent behavior

In conclusion, while ATNs and agents are very different entities, they can be composed[5] thanks to the uniqueness of their operational representation.

5. THE JARGONS ARCHITECTURE

Thoughout this section, we adopt the point of view of the Jargons system manager(s). Using and even designing Jargons do not necessitate a deep understanding of the system internals. The architecture of the Jargons system is somewhat complex, since it is highly reflective. However, as in Maciste [8], this reflective structure is mostly driven by a pragmatic effort, much more than by a theoretical one. Figure 3 describes the overall architecture of the system, as explained in the following sections.

5.1 Base level: managing jargon uses

As we already stated, a Scheme interpreter underlies the whole system[6]. On top of this interpreter, developers can use the available jargons to develop programs. The base-level framework interprets the jargon uses and translate them (if necessary) into correct Scheme code. This layer can be seen as a Scheme library but while a library generally does not change during a session, this particular library is highly generative.

For example, in the presence of the SRA jargon, the base level framework contains a predicate:

```
(sra:agent? thing)
```

This tests if thing is an agent or not. When we add the definition of a new agent, such as SillyRobot, then a more specialized predicate is added into the framework:

```
(agent:SillyRobot? thing)
```

Of course, the definition of a new agent can be done at runtime, hence the generative trait of the base-level framework.

[5]Here, the ATN implicitly references the agent through the use of perceptions and actions in the network transitions and the agent reciprocally references the network as its behaviour.

[6]We use almost all the features standardised in the R5RS document. Without bootstrap, the system needs important extensions like stateful macros (*à la Common-LISP*) or eval in the current environment. Some jargons also needs implementation-specific extensions: GUI frameworks, threads and so on. For our experiments, our platform of choice is MzScheme [10] but we also experimented Bigloo [12] (for compilation purposes).

5.2 Meta level: managing jargon definitions

The *meta-level* of the system is concerned with the design and implementation of new jargons. From this point of view, the descriptions of jargons form the inputs of the *incremental compiler*. This compiler will generate the specific parsers for the jargons as well as the calls to the corresponding transformers and interpreters.

As already stated, the particular jargon used to describe other jargons has the interesting property to be itself a jargon. Given this *meta-circular* definition, it is possible to bootstrap the system by providing the definition of METAJARGON. This way we can for example obtain a generic jargon parser and automatize the generation of several helper functions. This bootstrapping role is one of the functionalities of the *meta-level framework*.

To help at the construction of the interpreter part of the jargon, this layer also proposes several convenience functionalities such as:

- Node transformers: before being interpreted, particular node constructs can be transformed (context-sensitive parsing).
- Node selectors and filters: selection of nodes from generic patterns can help at the interpretation of jargon constructs. It is also possible to filter the constructs in a context-sensitive manner during the selection phase.
- Type managers: the type system is open to the addition of user types.
- Structure recorders: some jargons will record some informations (like an internal agent description in sra), generic functions exist to build these definitions.
- Code generators: the generation of code is supported by a set of functions, essentially by a prototype generation framework.

Of course, this layer is intensively developed at the moment and is enriched as new jargons are added in the system.

5.3 Meta-meta level: bootstrap and monitoring

During the development process, we achieved at a certain time *after* the start of the project to regenerate a part of the system (from a set of preliminary Jargons to METAJARGON). This operation, called *bootstrap*, is working on the whole system. Following the classification of [8], the bootstrap program is a *meta-meta-level* concept, related to control. While it is a little bit soon to develop on the usefulness of a meta-meta-level, we want to stress the point that the system would not have even been thought about without this two-level meta-tower in mind.

We envisage, as in Maciste, to use the meta-meta-level concept of *monitoring* to substantially augment the automatization features of the system (like controlling the bootstrap from observations at the base and meta-levels). For the moment, it is already possible to pre-generate, from a simple order, a specialized version of the Jargons system for a given configuration. The resulting code can then be passed to an optimizing Scheme compiler (eval forms are not needed in bootstrapped mode).

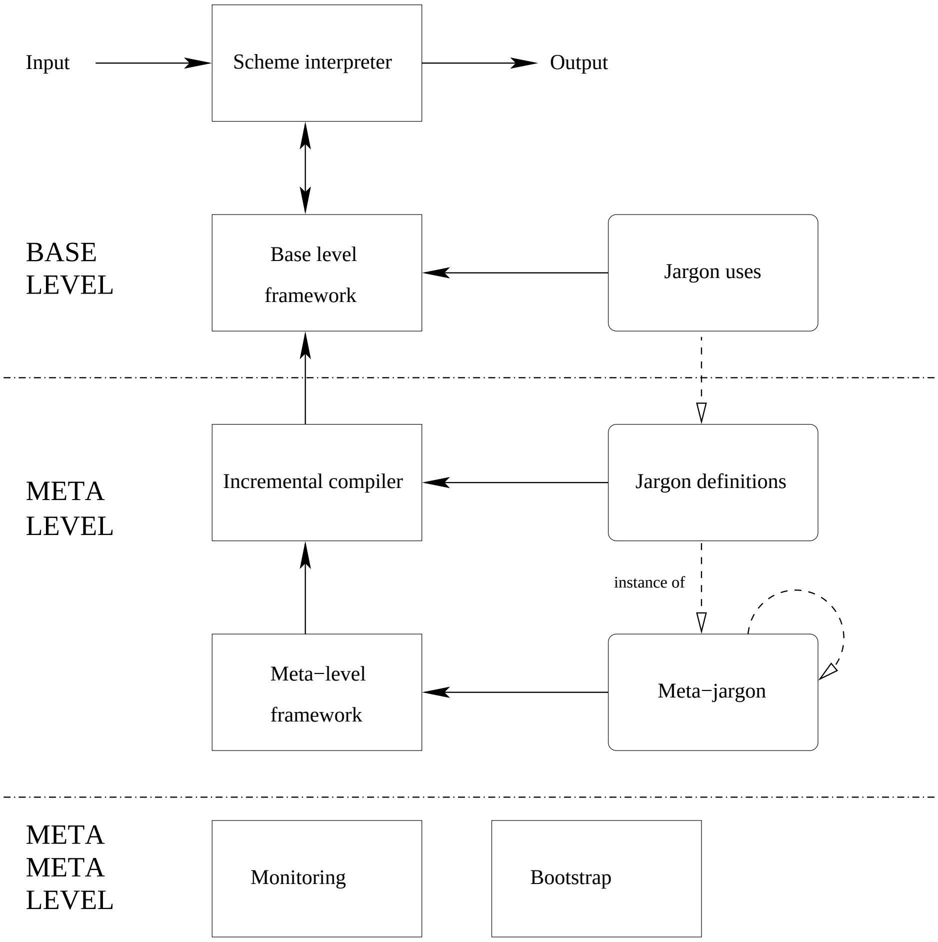

Figure 3: Jargons system architecture

6. RELATED WORK

Many research work on domain-specific languages have been presented in the USENIX DSL conferences [15]. However, few of these work concern the elaboration of generic DSL frameworks, they are more aimed at presenting specialized languages for specialized domains. That is, *top-down* visions (from domains to languages) are generally priviledged over *bottom-up* approaches (from languages to domains) like ours.

The work of Paul Hudak on modular interpreters for Domain-Specific Embeded Languages (DSEL) [3] is *in essence* very close to our Jargons system. Our motivations are quite comparable. However, while we focus on a methodology to implement and compose pragmatically DSELs, Hudak focuses on the interpreter construction. We do not agree that syntax is less important than semantics because we see the latter as the denotation of the former. Syntax (or structure, as we call it in this paper, following the XML terminology) captures the domain intention while semantics is more concerned by operationalization. Strong commitments to a given general-purpose (and quite exotic) programming language is another common point of the two approaches. However, embedment in Scheme and Haskell is a very different notion. Imperative traits seems important in both approaches, while Haskell's monads provide probably a better mathematical basis than Scheme's side effects. But our focus in on experimentation, though our bias for practicalness over mathematical soundness.

LAML [6] is another interesting language-oriented application embedded in Scheme. LAML styles resemble jargons while their domain is related to World Wide Web automation. It is however difficult to categorize both approach since Jargons could support Web-oriented DSLs while LAML may be extended to other domains. However, our discussion is more oriented toward the internals of our system and the methodology it supports than on domain-specific concerns such as WWW scripting. Papers on LAML do not, at least to our knowledge, describe the way to develop new styles.

The *compose* group [14] proposes also some comparable work. But while their framework is based on partial evaluation and so oriented toward speed efficiency, we do not put the focus on the performance issue. However, it is important to state that the Jargons approach does not pose any problem regarding theoretical performances. But in practice, our system is before all meant for prototyping purposes. However, many applications do not reclaim hyper-sophisticated optimizations and in that case, we propose in our opinion a viable approach.

Recently, the focus on DSLs has somewhat lost a part of its strength. It seems that from a top-down perspective, the *meta-modeling* techniques are more popular nowadays [9]. Talking about models instead of languages is, in our opinion, mainly a rhetoric trick since people have to express their models in one way or another. By focusing on computer problems in non-computer domains, meta-modeling work introduce an unavoidable gap between expert models and computational ones. This can be seen as a discontinuity between the conceptual and operational properties. We pose as our main objective to avoid this discontinuity, hence our bottom-up vision. The junction between the well-established results of meta-modeling techniques and the more and more powerful meta-programing systems such as Jargons seems to be a reachable objective.

Some research work, generally from the logic-based programming community, introduce the concept of *multi-paradigm programming*. Similarly to Mozart [11][7], our framework offers multiple ways - or styles or paradigms - to construct computer programs. However, the multi-paradigm programming environments generally offer a fixed set of paradigms while we propose a framework to define new paradigms. The drawback is one more time essentially related to practical speed performances.

As already indicated, the architecture of the jargons system is very close to (and in fact almost copied from) the Maciste system [8]. However, the two systems diverge in purpose: implementation of expert systems for Maciste and domain-specific languages for Jargons. From a more technical point of view, Maciste and Jargons do not provide the same underlying computation model. Maciste generate C code while Jargons is entirely based on the higher-level Scheme language. Once again, all the properties lost regarding practical performances are, in that case largely, balanced by the simplicity, scalability and maintainability of the jargons system.

7. CONCLUSION AND FUTURE WORK

The experiment described in this paper is mainly exploratory though the current results are yet encouraging. In a short period we were able to design and implement jargons for very different purposes (see table 1). For example, the agent-based jargons introduced in this paper are real and put in use in various examples related to Multi-Agent Systems.

Initially built to support a component-based programming environment [7], the Jargons system has been successfully extended to multiple programming styles and features. We see structured documents as very high-level and domain-specific but with weak semantics computer entities. Comparatively, programming languages are more low-level, more generic in purpose and (often) with stronger semantics. So, from our point of view, it seems like a good idea to try to bring closer both worlds.

Another goal we aimed throughout this research work was to understand and implement a true two-level meta-tower as explained in [8]. The three important direct consequences of this particular architecture are: meta-circular definitions, automatic validating parser generation and bootstrapping mode. In the future, we are confident that this architecture will offer more interesting properties: extended bootstrap, monitoring, and perhaps in the long term, autonomous activity (the system never stops).

From another perspective, we do not know yet any other usable[8] programming metalanguage[9] than Scheme that would have allowed the development of a so complex system in so little time and with so interesting results (from our biased point of view, of course). It is then also interesting to consider Jargons as an example of an application which takes advantage of the advanced features of the Scheme programming language.

This argument also leads to perhaps the most contradictory aspect of our approach : the comparision with more tra-

[7]The mozart language proposes a very interesting mix of constraint-based logic programming, object-orientation and concurrent computation.

[8]implementation.

[9]Initially focusins on ML dialects, we were as a matter quite disapointed by their lack of true meta-level features

ditional (and popular) approaches to development through classes frameworks and parametric modules. It is, as everybody knows, possible to answer domain-specific concerns using OO methodologies. However, from the user's point of view, there exist big differences between OO frameworks and DSL : learning curve, maintenance, provability, and so on. From the developer's point of view, the main difference concerns the fact that classes and modules can be used to implement jargons but also other paradigms like any-order functions, logic predicates, automata, components, agents and so on. It should finally be pointed out that at the operational level, we agree that operationalizations of objects (we call them prototypes and represent them by closures) provide a good basis for composition.

In the future, we intend to add new jargons to the system and produce applications that exploit this rich environment (mainly in AI and MAS). Jargons is before all a research platform for research experiments but we hope to share some results in the long-term.

8. REFERENCES

[1] J. Ferber. *Multi-Agent System: An Introduction to Distributed Artificial Intelligence*. Addison Wesley, February 1999.

[2] Z. Guessoum and J.-P. Briot. From active objects to autonomous agents. *IEEE Concurrency*, 7(3), 1999.

[3] P. Hudak. Modular domain specific languages and tools. In *Fifth International Conference on Software Reuse*, pages 134–142, Victoria, Canada, June 1998.

[4] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language scheme. Technical report, February 1998.

[5] B. A. Nardi. *A Small Matter of Programming*. MIT Press, 1993.

[6] K. Normark. Programming world wide web pages in scheme. *Sigplan Notices*, 34(12), 1999.

[7] F. Peschanski. A typeful composition model for dynamic software architectures. Technical report, University of Paris VI - Pierre et Marie Curie, 2001.

[8] J. Pitrat. Implementation of a reflective system. *Future Generation Computer Systems*, 12, 1996.

[9] N. Revault, H. Sahraoui, G. Blain, and J.-F. Perrot. A metamodeling technique: The métagen system. In *Tools 16: Tools Europe'95*, pages 127–139, Versailles, France, Mar. 1995. Prentice Hall.

[10] Rice University PLT group. *MzScheme v. 103*. `http://www.cs.rice.edu/CS/PLT/packages/mzscheme`.

[11] Saarlandes University Programming Systems Lab and al. *Mozart programming system v. 1.2.0*. `http://www.mozart-oz.org/`.

[12] M. Serrano. *Bigloo v.2.3a*. `http://kaolin.unice.fr/ serrano/bigloo/bigloo.html`.

[13] D. Sitaram. *Programming in Schelog*. `http://www.cs.rice.edu/CS/PLT/packages/schelog/`.

[14] S. Thibault. *Domain-Specific Languages: Conception, Implementation, and Application*. PhD thesis, University of Rennes 1, October 1998.

[15] USENIX. *2nd conference on Domain-Specific Language DSL'99*, http://www.usenix.org, 1999.

[16] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages, an annotated bibilography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

APPENDIX

A. METAJARGON : VERSION 0.3

In this appendix, we present the structure of the core MetaJargon language, expressed in itself. There is a richer version of the meta-language which covers use references, anticipated evaluation and unordered sequences but jargon definitions are preprocessed to match this core level language in order to speed-up the parsing and interpretation processes as well as to ease the further automation properties of the system[10]

```
(JARGON METAJARGON "metajargon.sld"
  (VERSION "0.3")
  (DESCRIPTION "Core meta-langage structure")
  (AUTHOR "Fred Peschanski")
  (ROOT JARGON (TERMINAL name SYMBOL) (TERMINAL url STRING)
     (? (TERMINAL VERSION STRING))
     (? (TERMINAL DESCRIPTION STRING))
     (? (TERMINAL AUTHOR STRING))
     (+ (NODE ROOT (TERMINAL name SYMBOL)
           (* (| (NODE NODE (TERMINAL name SYMBOL)
                    (+ (| (IREF JARGON-ROOT-NODE)
                          (IREF JARGON-ROOT-TERMINAL)
                          (IREF JARGON-ROOT-*)
                          (IREF JARGON-ROOT-|)
                          (IREF JARGON-ROOT-?)
                          (IREF JARGON-ROOT-+)
                          (IREF JARGON-ROOT-&)
                          (IREF JARGON-ROOT-IREF))))
                 (NODE TERMINAL (TERMINAL name SYMBOL)
                    (TERMINAL type SYMBOL)
                    (| (NODE ENUM (+ (TERMINAL value ANY)))
                       (NODE EXCLUDE (+ (TERMINAL value ANY)))
                       (& (IREF JARGON-ROOT-TERMINAL-ENUM)
                          (IREF JARGON-ROOT-TERMINAL-EXCLUDE))
                       (& (IREF JARGON-ROOT-TERMINAL-EXCLUDE)
                          (IREF JARGON-ROOT-TERMINAL-ENUM))
                      (NODE EXACT (TERMINAL value ANY))))
                 (NODE * (IREF JARGON-ROOT-NODE))
                 (NODE | (IREF JARGON-ROOT-NODE))
                 (NODE ? (IREF JARGON-ROOT-NODE))
                 (NODE + (IREF JARGON-ROOT-NODE))
                 (NODE & (IREF JARGON-ROOT-NODE))
                 (NODE SCHEMEDEF EMPTY)
                 (NODE SCHEMEONEDEF EMPTY)
                 (NODE IREF (TERMINAL path SYMBOL))))))))
```

B. MICROON JARGON IMPLEMENTATION

We present here a part of the implementation of a minimal object-oriented jargon named microon. It provides basic OO features such as encapsulation in classes, access modifiers and initialization protocol. The implementation of this jargon is less than 100 lines of Scheme code, all the structure accessors being automatically generated by the Jargons system, as well as the validating parser and support.

First, here is the jargon definition for microon :

```
(JARGON microon "microon.sld"
   (VERSION "0.1")
   (DESCRIPTION "Micro Object Model")
   (AUTHOR "Fred Peschanski")
   (ROOT object
     (TERMINAL name SYMBOL)
     (* (| (NODE slot (TERMINAL access SYMBOL (ENUM public private readonly))
                (TERMINAL name SYMBOL (EXCLUDE public private readonly)))
           (NODE method (TERMINAL access SYMBOL (ENUM public private))
                        (TERMINAL name SYMBOL (EXCLUDE public private))
```

[10] As a general rule, we think that the more the metalanguage is concise, the more it is possible to support it at the implementation level.

```
                         (TERMINAL params DOTTED-LIST-OF-SYMBOL)
                         (SCHEMEDEF))))
     (? (NODE initialize (TERMINAL params DOTTED-LIST-OF-SYMBOL)
           (SCHEMEDEF)))
     (* (! (IREF object-slot) (IREF object-method)))))
```

The interpreter consists in a few utility functions that generate the right s-expressions for the prototype generation function. We present below the implementation of the method management, consisting in two part: the method definition and the message selector. The main function calls the prototype generator function of the meta-level framework.

```
;; module jargon initializations
(module:module microon
  (require jargon "jargon-core.scm")
  (require error "error.scm")
  (extra-files "microon.sld"))

;; Cannot be automatically generated
(define (microon:_object-interpret def)
  (let* ((name (microon:_object-get-name def))
         (slots (microon:_select-slots def))
         (methods (microon:_select-methods def))
         (initializer (microon:_select-initializer def))
         (iparams (microon:_initializer-get-params initializer))
         (ibody (microon:_initializer-get-body initializer)))
    (microon:_make-object-creator name slots methods iparams ibody)))

;; create the method list
(define (microon:_make-method-list methods)
  (meta:prototype-methods (lambda (method)
                             (list (microon:_method-get-name method)
                                   (cons (microon:_method-get-params method)
                                         (microon:_method-get-body method)))) methods))

;; create the method messages
(define (microon:_make-method-messages methods)
  (cond ((null? methods) '())
        ((eq? (microon:_method-get-access (car methods)) 'public)
         (cons (list (list (microon:_method-get-name (car methods)))
                     (list 'apply (microon:_method-get-name (car methods)) 'args))
                     (microon:_make-method-messages (cdr methods))))
        (else (microon:_make-method-messages (cdr methods)))))

;; create object closure creator
(define (microon:_make-object-creator name slots methods iparams ibody)
  (meta:generate-default-prototype
    (symbolize 'microon:make- name) ;; generator name
    (list (cons 'class name)  ;; slots
          (cons 'initialized #f)
          (microon:_make-slot-list slots))
    (list (microon:_make-method-list methods) ;; methods
          (microon:_make-initializer iparams ibody))
    (list (cons 'self 'self) ;; messages
          (cons 'get-class 'class)
          (microon:_make-slot-accessors)
          (microon:_make-slot-mutators)
          (microon:_make-method-messages)
          (cons 'initialize '(apply initialize args)))))
```