ERESYE: an Erlang Expert System Engine

Antonella Di Stefano[2], Francesca Gangemi[1], Corrado Santoro[2]

[1]Erlang Training and Consulting
416 Fruit & Wool Exchange
Brushfield Street, London E1 6EL, England
francesca@erlang-consulting.com

[2]University of Catania
Dept. of Computer and Telecomm. Engineering
Viale A. Doria, 6
95125 - Catania, Italy
{ad,csanto}@diit.unict.it

Abstract

This paper describes ERESYE, a tool for the realization of intelligent systems (*expert systems*) using the Erlang language. ERESYE is a rule production system that allows rules to be written as Erlang function clauses, providing support for their execution. ERESYE is also able to support object-oriented concepts and ontologies thanks to a suitable *ontology handling tool*, providing means to translate object-based concepts into an Erlang form. The architecture of ERESYE and its basic working scheme are described in the paper. A comparison with CLIPS, one of the most known tools for expert system programming, is also made. The description of some examples of ERESYE usage are provided to show the effectiveness and the validity of the proposed solution, which opens new and interesting application scenario for Erlang.

1 Introduction

To date, computer systems are programmed using traditional and well-known languages like C, C++ or Java. When a system has to include some "intelligent" capabilities, as in the design of agent-based systems [27, 23], the integration of external tools is mandatory. As an example, when an expert system able to process inference rules is needed, the programmer is forced to use a tool such as CLIPS [2] or JESS [1] to implement the "intelligent" part and integrate it with the language employed in the computational part. This is due to the fact that assertive languages are not able to represent rules and knowledge through native constructs with an adequate degree of efficiency and flexibility. On the other hand, current rule processing tools concentrate on expert system support and do not provide functions and/or libraries for general-purpose programming.

Mixing programming approaches leads to peculiar problems since it forces the programmer to deal with two different languages and programming philosophies, and it introduces inefficiencies arising from the need to provide a suitable interface to convert data transferred between the two runtime environments.

Languages such as Prolog or LISP are able to offer a solution to the problem above: LISP can be considered general-purpose while offering many features for expert system programming [20, 18]; Prolog, even if it was initially designed as a (specialized) logic language, has then been enriched with many characteristics making it general-purpose.

In this scenario, the Erlang language emerges as an interesting alternative approach: Not only is Erlang general-purpose and easy to use, but it presents some features that make it suitable to write rule-based systems. Moreover, the characteristics of the runtime environment, such as concurrency model, "behaviour"-based programming, fault-tolerance support, whose effectiveness has already been demonstrated, are able to provide a general-purpose programming and execution platform much more interesting than that of Prolog and LISP. As for rule-based programming, the syntax and semantics of Erlang present three interesting characteristics:

1. Symbols and primitive types (i.e. atoms and tuples) are well suited to represent *facts* of a knowledge base.
2. Function clauses, written as *predicates* on parameters, which if matched activate the clause, fit well in the representation of the *precondition* part of a rule; at the same time, the function body can represent the *action* part.
3. The use of pattern matching facilitates the implementation of rule-handling algorithms, also improving processing performances.

Following these intuitions, the authors started a research project aimed at verifying the ability of Erlang to realize rule-processing systems, in order to offer a more complete runtime environment for Erlang programs and expanding the application fields of this language with new areas. As a result, a software library called ERESYE (*ERlang Expert SYstem Engine*) has been written.

ERESYE gives a programmer the ability to write *rule engines*, each featuring its own knowledge base and a set of rules. These rules are written as Erlang functions, using the standard Erlang syntax.

ERESYE is enriched with the possibility of handling ontologies [17], by allowing the definition of the "universe of discourse" by means of class hierarchies; such classes can be included in rules through classical object-based abstractions.

In addition, ERESYE supports fault-tolerance since an engine is an instance of the `gen_server` behaviour and can thus be included in a supervisor tree.

ERESYE is part of the agent platform eXAT [3, 24, 16, 15, 14], but its characteristics make it suitable for both agent-based applications and general-purpose contexts. An API is made available, allowing any Erlang application to interact with an ERESYE engine. Moreover, an ERESYE engine, thanks to its structure and API functions, can also serve as a Linda-like *coordination infrastructure* [10], in order to support coordination in Erlang programs.

The paper is structured as follows. Section 2 describes the main features of the tools currently available to write rule processing systems, focusing in particular on CLIPS [2]. It then provides an overview of RETE [19], the matching algorithm currently used in all of these tools. Section 3 describes ERESYE, showing its architecture and the basic functionality, and highlighting the way in which an ontology can be written and handled. Section 4 presents some examples. Section 5 compares CLIPS with ERESYE. Section 6 concludes the paper.

2 Overview of CLIPS and RETE

Like human experts, artificial expert systems provide solutions based on *experience* and *knowledge*. These system are often implemented by means of *production rules* manipulating a knowledge expressed as a set of*facts*. Rules and facts are then "crunched" by an *inference engine* that executes rules and change the knowledge base in order to solve the specific problem. Rules are expressed in the `if-then` form, where the right-hand side (RHS) represents the action to execute when asserted facts satisfy the conditions expressed in the left-hand side (LHS).

To develop an expert system, a model of knowledge and an inference engine are necessary. To this aim, several tools are today available [18, 5, 4, 2, 1]. These are very similar to one another, as they use the RETE algorithm [19], a fast many-to-many pattern matching solution that, even if proposed about 20 years ago, is still "the reference solution" for inference engines.

Given such a similarity of expert system programming tools, we provide, in the following, a brief description of the most widely used tool, CLIPS [2] (C-Language Integrated Production System). Then the basic working scheme of the RETE algorithm will be presented.

```
 1 ;; This rule is activated when the fact
 2 ;; ‘(temperature ?x F)’, with ?x any, is asserted.
 3 (defrule convert
 4   ?temp <- (temperature ?x F)
 5   =>
 6   (retract ?temp)
 7   (assert (temperature (+ (- ?x 32) (/ 5 9) C ))))
 8
 9 ;; This rule is activated when the facts
10 ;; ‘(on-table ?object)’ and ‘(color ?object red)’,
11 ;; or ‘(on-table ?object)’ and ‘(color ?object blue)’
12 ;; are asserted.
13 (defrule move
14   (declare salience 99)
15   (on-table ?object)
16   ((color ?object red) or (color ?object blue))
17   =>
18   (assert (move-object ?object))
19   (retract (on-table ?object)) )
```

Figure 1: A CLIPS Sample Program

2.1 CLIPS

CLIPS is a tool that offers a complete environment for developing rule-based expert systems. It uses a LISP-like programming language that is able not only to represent rules and facts, but also to model the knowledge using *objects*, organized according to object-oriented concepts (i.e. classes with inheritance). Rules can match objects/classes as well as facts, which consist of one or more fields enclosed in parentheses and delimited by one or more spaces, e.g. `(temperature 0.556 C)`. Rules, defined by using the construct `defrule`, are expressed with a LHS providing the conditions that must be satisfied to activate the rule, and a RHS, after the "=>" sign, expressing the action. Rule actions change the knowledge by means of `assert` or `retract` operations, which respectively insert new fact or remove an existing fact from the knowledge base. Lines 1–7 of Figure 1 report the definition of a simple rule called `convert`.

The LHS of a CLIPS rule is made up of a series of *conditional elements* that must be satisfied to activate the rule. Generally, each conditional element is an expression matching a pattern against either a fact or an object (i.e. an instance of a user-defined class), which has to be present in the knowledge base. Matching is specified using actual values or variable bindings; as an example, the pattern in line 4 of Figure 1 matches all facts (`temperature` X `F`), with X any, binding this second element to the variable[1] `?x`. Logical operations, such as `and`, `or`, etc., can be also used as connectives for conditional elements, as rule `move` in Figure 1 shows[2].

When all patterns of a rule match facts, the rule is activated and put on the *agenda*, a collection of activations whose actions have not yet been executed. There could be multiple activations on the *agenda*, and CLIPS sorts them according to their *salience*, a priority on execution established when the rule is defined (see line 14 of Figure 1). A CLIPS program terminates executing when no activations remain in the *agenda*.

As additional features, CLIPS allows user functions to be defined and provides an API that allows external programs to be integrated with a CLIPS system, as illustrated in the next Section.

2.2 Integrating CLIPS with External Programs

The CLIPS tool is entirely written using the C language and provided as a collection of source files subject to a BSD-like license[3].

CLIPS has no support for dynamically linkable object modules, so integration of external functions or programs can be done by manually patching CLIPS source code and recompiling everything. C-native user-defined functions can be added by statically linking the function source code

[1] Variables in CLIPS are expressed by using the `?` prefix.

[2] The connective `and` is implicit and can be thus removed.

[3] See `http://www.ghg.net/clips/CLIPS-FAQ`.

with CLIPS sources, provided that some hooks have to be inserted in the main CLIPS source to signal the presence of such new functions. In a similar way, the integration of CLIPS with an existing program (given that it is written in C or C++) is performed by including CLIPS sources in the program's compilation process and using CLIPS API to access datatypes, rules, etc. Also in this case, the output is a single "all-in-one" executable.

If the program is instead written in language different than C/C++, the native interface provided by the language itself (e.g. JNI, if the language is Java) must be used, while the native functions interfacing CLIPS with the program must be, once again, compiled and linked with CLIPS sources.

2.3 RETE Algorithm

RETE is a match algorithm developed by Forgy in 1982 [19]. Since its first proposal, there have been few improvements and, even today, RETE is the most used algorithm for the implementation of rule-based expert systems. The goal of the algorithm is to establish whether rule conditions match the current environment of asserted facts: If all conditions of a rule are satisfied the rule is *fired* and the corresponding action is executed.

RETE is an efficient algorithm because it maintains match status from previous computations and does not iterate over all the facts and rules. This is based on the empirical observation, called *temporal redundancy*, based on the principle that when a fact is asserted (or retracted) it affects only a small number of rules. The algorithm further reduces matching time by sharing test structures of the rules that feature identical conditions. It stems from the observation of a *structural similarity*: The same pattern often appears in more than one rule.

In the RETE algorithm, LHSs of rules are transformed into a rooted direct acyclic graph (see Figure 2). In this special kind of network, nodes represent patterns, while paths from the root to leaves represent the LHS of rules. The network consists of *1-input* nodes, also called *alpha-nodes*, and *2-input* nodes, called *join-nodes*. The alpha-nodes perform tests on individual facts, while join-nodes test for consistent variable bindings between conditional elements, e.g. `(color ?x red)` and `(is-on ?x table)`, where the same variable binding appears in two different pattern matching expressions. A memory is associated with each node, storing the result of matching operations. An *alpha-memory* holds the current set of facts that satisfy the condition of the associated alpha-node. *Beta-memories*, associated to join-nodes, store partial instantiations of productions, i.e. sequences of facts that match some but not all the conditions of a rule. A sequence of facts is called *token*.

When a fact is asserted[4], it is passed to the top node of the network. Each alpha-node compares the value of the incoming fact with its condition pattern. If the test succeeds, the fact is stored in the corresponding alpha-memory and it is passed down to its successor nodes. Therefore, this fact arrives at a join-node and is compared with the tokens in the beta-memory of the parent node. If there is a match, a new token consisting of paired tokens with consistent variable bindings is created, stored in the beta-memory and propagated down to further successors. When a token reaches the bottom of the network, all the tests on the left-hand side of a particular rule succeeded, so the rule is fired.

Figure 2 reports the structure of the network of the RETE algorithm for the rule described in the upper-left side of the same Figure; this rule is activated following conditions C_1, C_2 and C_3 on three facts with two join variable bindings (variables X and Y). As Figure 2 shows, the network is made of three alpha-nodes and three join-nodes with associated memories; alpha-memories contain all facts matching conditions C_1, C_2 and C_3, respectively, while beta-memories (β_1, β_2 and β_3) contain tokens matching join conditions according to variable bindings.

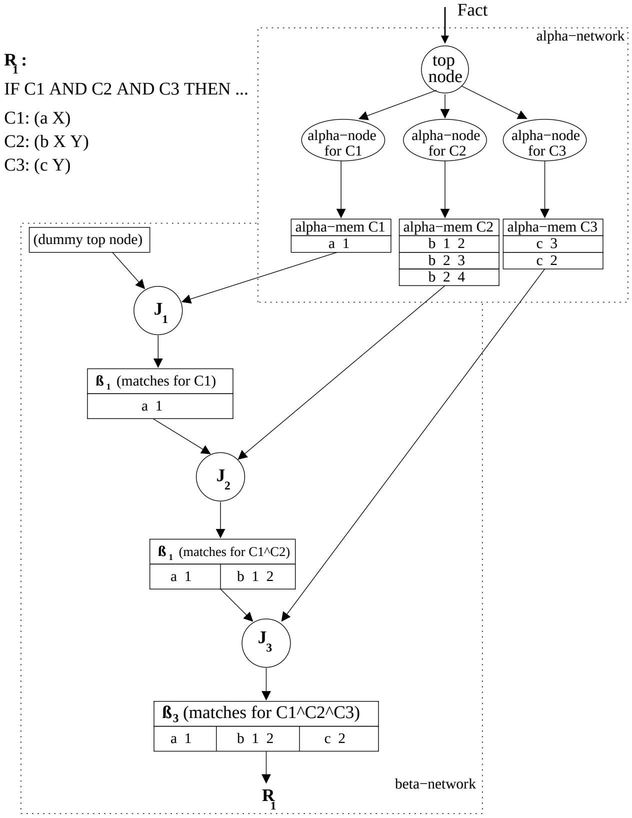

Figure 2: The network used by RETE for a sample rule

```
-module (sample).
-compile ([export_all]).

convert (Engine, {temperature, X, ’F’} = Temp) ->
  eresye:retract (Engine, Temp),
  eresye:assert (Engine,
                 {temperature, (X - 32) * 5/9, ’C’}).

check (Engine,
       {temperature, X, ’C’}, {alarm, off} = F)
    when X > 30 ->
  eresye:retract (Engine, F),
  eresye:assert (Engine, {alarm, on});

check (Engine,
       {temperature, X, ’C’}, {alarm, on} = F)
    when X < 25 ->
  eresye:retract (Engine, F),
  eresye:assert (Engine, {alarm, off}).

fan_coil (Engine,
          {alarm, on}, {fan_coil, off})
  when not ["{power, fail}"];true ->
  eresye:retract (Engine, {fan_coil, off}),
  eresye:assert (Engine, {fan_coil, on}).


start () ->
  eresye:start (sample_engine),
  eresye:add_rule (sample_engine, {sample, convert}),
  eresye:add_rule (sample_engine, {sample, check}),
  eresye:add_rule (sample_engine, {sample, fan_coil}).
```

Figure 3: Examples of ERESYE Usage

3 ERESYE

ERESYE is an Erlang library designed to allow the creation, management and execution of *rule-processing engines*, and is thus suited for the realization of expert systems.

Each engine has a *name* and *knowledge base (KB)*; the KB is made of a *fact base (FB)*, storing the set of facts representing the current knowledge, and a *rule base (RB)*, storing the set of inference rules representing the reasoning capability of the engine. Each fact is written in the form of an Erlang tuple, e.g. {`temperature, 50, 'F'`}, {`alarm,on`}, {`speed, 30, 'km/h'`}. Inference rules, which represent the actions to be executed by the engine when one or more particular facts are asserted in the FB, are instead written using standard Erlang functions. In particular, an ERESYE rule is implemented with an Erlang function clause where the first parameter represents the engine name in which the rule is executed and the other parameters are tuples representing the templates of the facts that must be asserted in the KB for the rule to be activated. For example, rule `convert` (lines 4–7 of Figure 3) is fired when the fact represented by the tuple {`temperature, X, 'F'`} is asserted; while the rule represented by the first clause of function `check` (line 9) is fired when both facts {`temperature, X, 'C'`}, with `X` greater than 30, and {`alarm, on`} are asserted. When a rule is fired, the first parameter of the relevant function (e.g. `Engine` in the examples in Figure 3) is bound to the engine name in which the rule itself is executed; this allows the KB manipulation code, present in the function body, to refer to the proper engine.

Guards can also be specified in rule/function declarations (as shown in the examples), creating additional conditions to be met in order for the rules to be fired. Guards can be also used to specify a *negated condition* on a fact assertion, i.e. checking that facts matching specific templates *are not* present in the FB. Rule `fan_coil` in Figure 3 is an example of such a use of negated conditions: it says that the action is taken only if {`alarm, on`} and {`fan_coil, off`} are asserted, and {`power, fail`} is **not** asserted. As it can be seen from the example, since Erlang does not offer a suitable "not syntax", a workaround is used to specify such negated conditions: Facts or fact templates must be specified in a guard as a *list of strings* prefixed by "`not`" and or'ed with the `true` constant. Indeed, this guard is useless from the Erlang point of view (the `true` constant triggers the clause in any case), but is instead correctly interpreted by ERESYE[5].

The body of a rule implements the action to be executed when the rule is fired; it can contain any Erlang expression, as well as calls to functions for KB manipulation. To this aim, a suitable set of functions of the ERESYE API allows Erlang programs to interact with an ERESYE engine in order to assert a fact, retract a fact, wait for the presence of a fact with a given pattern, add a new rule, change rule priority, delete a rule, etc. The example in Figure 3 shows the use of the ERESYE API to manipulate the KB: Functions `assert` and `retract` insert a new fact and remove an existing fact respectively; functions `start` and `add_rule` are instead provided to create a new engine and to add rules to such an engine.

According to Erlang semantics, rules can be expressed using different clauses of the same function. This can be seen in the `check_alarm` function in Figure 3, which consists of two clauses. In these cases, each clause is considered as a different rule[6]. Since rules are referred by the ERESYE API using the function name, the possibility of expressing different clauses implies to *group* such clauses in a *rule set*, in order to be manipulated altogether. When a function name representing a rule set is given to one of the API functions for inserting the rule, deleting a rule, changing the priority of a rule, etc., the API operation is applied to all the rules of the set.

3.1 ERESYE Architecture

From the architectural and runtime point of view, each ERESYE engine is composed of three Erlang processes, as Figure 4 depicts: the *Processor*, the *Rule Scheduler* and the *Executor*[7].

[4] Similar actions are performed by the algorithm also when a fact is retracted.

[5] We agree that this "not syntax" is not so elegant and we plan to improve it in future releases of ERESYE.

[6] An exception to this mechanism is the one used in ontology handling. There each function is considered as a *single rule*, no matter its clauses. This different functionality, which is not the default behaviour, will be described in Section 3.3.

[7] In particular, the Processor and the Rule Scheduler are implemented as instances of "gen_server" OTP behaviour.

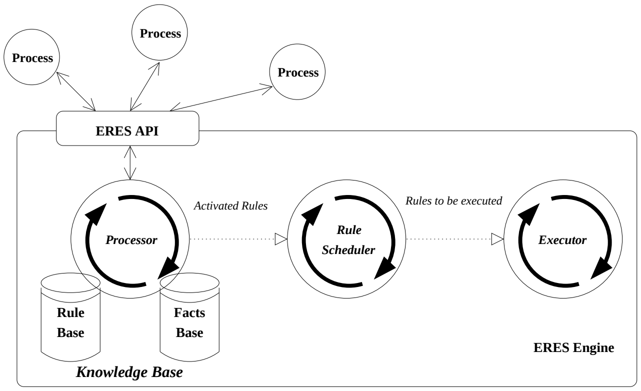

Figure 4: Architecture of an ERESYE Engine

The *Processor* has the responsibility of managing the *fact base* and the *rule base*. To allow the interaction of other Erlang processes with this engine, the *Processor* offers an API based on message passing, a mechanism commonly used in Erlang to perform interaction among processes.

The main task of the *Processor* is the identification of the rules to be executed. Each time a fact is asserted or retracted the Processor runs a modified version of the RETE algorithm to identify all the rules to be activated: They are the rules containing, in the function clause, a pattern matching the asserted/retracted fact. These rules (i.e. the identifiers of the associated function clauses that have been fired) are sent to the *Rule Scheduler* for execution.

Since each rule may have an assigned priority, which reflects the importance with respect to other rules[8], the process *Rule Scheduler* is charged with the task of selecting rules to be executed on the basis of their priority and according to the engine scheduling policy, which can be selected among the following choices[9]:

- **Depth**. A rule with priority p is scheduled before each other rule with a priority less than p. If there exists other rules with priority p, the new rule is scheduled *before* the others.
- **Breadth**. A rule with priority p is scheduled before each other rule with a priority less than p. If there exists other rules with priority p, the new rule is scheduled *after* the others.
- **Fifo**. Rules are scheduled in the same order of their activation, no matter of their priorities.

Each time a rule is scheduled, it is sent to the process *Executor*, which really performs the execution of the rule body; when the latter is completed, another rule can be accepted for execution from the *Rule Scheduler*. Rule execution is performed in a sequential fashion, according to priorities and the scheduling policy. If needed, however, a rule execution can be *detached* and started in a separated process. This detaching ability, to be specified when the rule is added to the engine (by means of the `add_rule` API function), can be used when concurrent computations have to be implemented.

3.2 ERESYE and RETE

ERESYE inference engines run a modified version of the RETE algorithm to determine which rule has to be activated following a change in the knowledge base. The implementation takes advantage of Erlang native matching constructs to build the conditions for alpha- and beta-memories. This working scheme is described in the following paragraphs.

When a rule is added to the system, the first step is to retrieve LHS conditions by obtaining the definition of the function clause that specifies the new rule. This operation is performed by

[8] The greater the number the higher the priority.

[9] The execution policy is specified by the user process when it creates the engine.

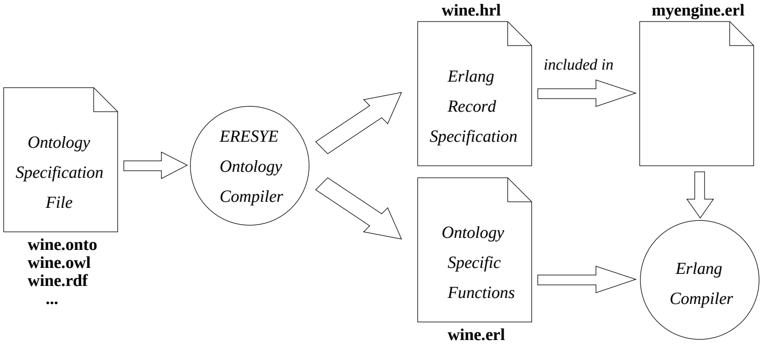

Figure 5: ERESYE Ontology Handling

parsing the source code (using the `epp` module) and building a string representation of the fact patterns activating the rule. As an example, for the rule

```
sample_rule(Engine, {a, X}, {b, X, Y}, {c, Y}) ->
   %% do something.
```

the source code analysis returns the list `["{a,X}", "{b,X,Y}", "{c,Y}"]`[10].

According to the RETE algorithm, for each condition, an alpha-node and an alpha-memory are created. In order to test if an asserted (or retracted) fact matches the condition associated to an alpha-node, ERESYE's RETE implementation takes advantage of Erlang's pattern matching mechanism: A specific `fun` is created at run-time[11] and it is called following fact assertion, to establish whether it satisfies the condition associated to the alpha-node. For example, given the `sample_rule` written above, the `funs` that check conditions for alpha nodes are the following:

Node	Fun
α_1	`fun ({a, X}) -> true end.`
α_2	`fun ({b, X, Y}) -> true end.`
α_3	`fun ({c, Y}) -> true end.`

As for join-nodes, as reported in Section 2.3, they are charged with the task of checking for consistent variable bindings between two or more condition patterns, e.g. the variable X, which appears in both the first and the second condition of `sample_rule`, must be bound to the same value in order to the activate rule. Erlang allows the implementation of this test in a simple way, thanks (once again) to its pattern-matching feature. Like alpha-nodes, a `fun` is created for each join-node; this fun has two arguments, each relevant to one of the two inputs of a join-node. For the `sample_rule` above (which, in turn, generates the RETE network depicted in Figure 2), `funs` associated to join-nodes are:

Node	Fun
J_1	`fun ([], {a, X}) -> true end.`
J_2	`fun ([{a, X}], {b, X, Y}) -> true end.`
J_3	`fun ([{a, X},{b, X, Y}], {c, Y}) -> true end.`

Note that the first argument matches a *token*, i.e. a list of facts coming from the beta-memory of the parent join-node. The second argument instead matches a fact that arrives from an alpha-memory. If pattern variables are consistently bound, the execution of the `fun` succeeded and a new token can be created and passed down to its successor. Only tokens that reach the bottom of the network cause the activation of a rule.

[10] Each condition is represented as a string because pattern variables are unbound.

[11] The `fun` is created *on-the-fly* with use of `erl_scan`, `erl_parse` and `erl_eval` modules.

***(a)* wine.onto**

```
class(wine_grape) ->
{ name = [string, mandatory, nodefault] };

class(wine) ->
{ name = [string, mandatory, nodefault],
  color = [string, mandatory, nodefault],
  flavor = [string, mandatory, nodefault],
  grape = [set_of(wine_grape), mandatory, nodefault],
  sugar = [string, mandatory, nodefault]};

class('red-wine') -> is_a(wine),
{ color = [string, mandatory, default(red)] };

class('white-wine') -> is_a(wine),
{ color = [string, mandatory, default(white)] };

class('Chianti') -> is_a('red-wine'),
{ sugar = [string, mandatory, default(dry)] }.
```

***(b)* wine.hrl**

```
-record('wine_grape',{
  'name'}).

-record('wine',{
  'name',
  'color',
  'flavor',
  'grape',
  'sugar'}).

-record('red-wine',{
  'name',
  'color' = 'red',
  'flavor',
  'grape',
  'sugar'}).

-record('white-wine',{
  'name',
  'color' = 'white',
  'flavor',
  'grape',
  'sugar'}).

-record('Chianti',{
  'name',
  'color' = 'red',
  'flavor',
  'grape',
  'sugar' = 'dry'}).
```

***(c)* test_wine.erl**

```
-module (test_wine).
-include ("wine.hrl").

my_rule (Engine, #wine{} = W) -> % Executed following #wine assertion.
  .... ;  % Do processing.
my_rule (Engine, W) -> % Executed following #'red-wine', #'white-wine'
                       % and #'Chianti' assertion.
  my_rule (Engine, wine:wine (W)). % Typecast to #wine.

start () -> eresye:start (myengine, wine),
            eresye:add_rule (myengine, {test_wine, my_rule, 1}).
```

Figure 6: The “wine” ontology, the generated include file and an example of its usage

3.3 Modeling and Using Ontologies

ERESYE expresses each fact by means of an Erlang tuple allowing programmers to establish meaningful structures, according to the real-world facts they want to represent. In writing intelligent systems, however, it is usual to model the “universe of discourse” by using *ontologies* [17], providing an easy way not only to express the concepts but also to organize them with useful relationships, such as *is-a*, *part-of*, etc. Ontology support is also becoming important thanks to the increasing research on semantic web [8] and the development of standards for ontology specification, such as RDF, OWL or OWL-S [26, 25].

In order to comply with this current trend in intelligent system engineering, ERESYE is able to support ontologies—i.e. *classes* with hierarchies and *objects*—and use them in engines. But since Erlang is not object-oriented, ERESYE comes with a tool able to transform an object-based ontology specification into an Erlang-readable (non-object-based) form, maintaining semantics in ERESYE engines. As Figure 5 shows, the *ERESYE Ontology Compiler* tool is an Erlang program that parses ontology specification files written in a standard notation and produces two Erlang source files: (*i*) a **.hrl** file containing class definitions as Erlang *records*, and (*ii*) a **.erl** file containing a set of functions for the manipulation of the specific ontology. The **.hrl** file is generated by translating each class into an Erlang record, provided that the hierarchy is “flattened” by incorporating each attribute of a class-record in all the relevant child class-records (in the following, we will adopt the term *class-record* to indicate an Erlang record specifying a class of an ontology); therefore, creating a fact referring to an object of class `'T'` implies to create and Erlang record of type `'T'`[12]. Information on class hierarchy lost in the include file is instead encoded in the **.erl** file by means of suitable `is_a` and `childof` functions. The source (**.erl**) file also contains some generated functions to perform class typecasting.

[12] Using records in facts and rules is not in contrast with the basic functioning of ERESYE that maps facts into tuples, because, from the runtime point of view, an Erlang record is transformed into a tuple by the Erlang compiler.

As an example, Figure 6 shows a snippet of the definition of a “wine” ontology in file `wine.onto`[13]. In the current version of ERESYE, ontologies can be written using an ad-hoc Erlang-like syntax. The ability to translate files written in standard notations, such as OWL, will be available in the next releases of ERESYE[14]. In the same Figure 6, `wine.hrl` is the file generated by the ERESYE Ontology Compiler and reports the translation in Erlang records of the classes of the “wine” ontology. Note the flattening applied to resolve class hierarchy.

Once generated, the two ontology-related files can be employed to realize ontology-aware rule-based systems: the **.hrl** file has be included in the source file specifying the rules of the engine, while functions in the **.erl** file can be, when needed, directly invoked. This ontology awareness, while supported by ERESYE, needs a particular care in rule definition: A rule like

```
my_rule(Engine, #wine  = W) -> %% do something.
```

has a different meaning if the ontological or Erlang point of views are considered. In the ontological point of view, such a rule has to be fired when a class-record of one of the types `'wine'`, `'red-wine'`, `'white-wine'` and `'Chianti'` is asserted (i.e. the base class-record `'wine'` and all of its children). But in the Erlang point of view, where object-orientation is not supported, the function head will match *only* if the second parameter will be a `'wine'` record. In ERESYE this mismatch in point of views is solved with a two-fold mechanism: (*i*) by delegating to the algorithm RETE the task of ontological matching, and (*ii*) by using a different way of declaring rules on the Erlang side, as depicted in the `test_wine.erl` sample source in Figure 6-c and described in the following.

The engine is made ontology-aware by specifying the ontology name as the second parameter of the creation function (line 10 of the listing in Figure 6-c); this allows the engine to properly understand that rules matching of class-records have to be processed according to object-based concepts. Such awareness is supported by the use of the include and source files generated from ontology specification. As an example, the function clause in line 4 of Figure 6-c will refer to a match to class-record `'wine'` and its children. A direct consequence of this ontology awareness is that, in building the structure for the RETE network, the conditions for alpha- and join-nodes will be generated by considering the class hierarchy, and thus, for the example above, by matching records `'wine'`, `'red-wine'`, `'white-wine'` and `'Chianti'`.

On the Erlang side, since clause in line 4 of Figure 6-c, used to build conditions for RETE, will match only records `'wine'`, another clause must be added to match all the other cases: In the example `test_wine.erl`, this clause casts the parameter to a `'wine'` class-record by using a function provided by the generated ontology module and listed in Figure 7, and then calls the first clause to perform the processing. In such a case, this second clause will not to be considered as *another* rule[15], but only as a (less restricting) clause of the first rule to be executed following the standard Erlang clause-matching mechanism, not subject to RETE activation. To this aim, a programmer should be able to specify, when adding a rule, the function clause that has to be considered for building conditions for RETE; this can be performed by using another form of the `eresye:add_rule` API function, which accepts also the clause number (in order of appearance in the source file). Line 11 of Figure 6-c shows an example of what has been explained.

3.4 ERESYE Engines as Coordination Media

Even if ERESYE has been designed to include rule-based reasoning processes in Erlang programs, its engines can also serve as Linda-like *coordination media*. Linda [10] is a well-known coordination language offering a simple abstraction to perform coordination among parallel processes. It is based on a shared data structure, the *tuple space*, hosting data in the form of *Linda tuples*, i.e. sequences of typed fields represented with a comma-separated list of values enclosed in parentheses, e.g. `("data", 10, 3.1415)`. Coordination is made possible by means of the three basic

[13] This is an excerpt from the “wine” ontology reported in the W3C-RDF web site.

[14] In CLIPS, automatic generation of ontologies is made possible through a suitable plug-in for Protégé, a visual ontology tool able to process OWL files. To allow automatic generation also for ERESYE ontologies, the relevant Protégé plug-in is currently under development.

[15] We remind that the default working scheme of ERESYE considers each clause as a different rule.

```
'wine' (X = #'redwine'{}) ->
  #'wine'{
    'name' = X#'redwine'.'name',
    'body' = X#'redwine'.'body',
    'color' = X#'redwine'.'color',
    'flavor' = X#'redwine'.'flavor',
    'grape' = X#'redwine'.'grape',
    'sugar' = X#'redwine'.'sugar'};

'wine' (X = #'whitewine'{}) ->
  #'wine'{
    'name' = X#'whitewine'.'name',
    'body' = X#'whitewine'.'body',
    'color' = X#'whitewine'.'color',
    'flavor' = X#'whitewine'.'flavor',
    'grape' = X#'whitewine'.'grape',
    'sugar' = X#'whitewine'.'sugar'};

'wine' (X = #'Chianti'{}) ->
  #'wine'{
    'name' = X#'Chianti'.'name',
    'body' = X#'Chianti'.'body',
    'color' = X#'Chianti'.'color',
    'flavor' = X#'Chianti'.'flavor',
    'grape' = X#'Chianti'.'grape',
    'sugar' = X#'Chianti'.'sugar'}.

'redwine' (X = #'Chianti'{}) ->
  #'redwine'{
    'name' = X#'Chianti'.'name',
    'body' = X#'Chianti'.'body',
    'color' = X#'Chianti'.'color',
    'flavor' = X#'Chianti'.'flavor',
    'grape' = X#'Chianti'.'grape',
    'sugar' = X#'Chianti'.'sugar'}.
```

Figure 7: The cast functions generated from the "wine" ontology

operations, `out`, `in` and `rd`, that can be performed on the tuple space. `out` allows a new tuple to be written in the tuple space. `in` extracts a tuple, from the tuple space, given its *template*, i.e. a specification of how the tuple to extract has to be formed. `rd` behaves like `in` but reads the tuple without removing it from the tuple space. A template specification for `in` and `rd` is a tuple whose fields can be constants used to match actual values, or type names used to specify a match with a given type. For example, the template `(?String, 10, ?Float)` matches all tuples whose first field is a string, the second is the constant 10, and the third is a floating-point number. Operations `in` and `rd` are *blocking*, i.e. if no tuple matching the template is present, they blocks until another process performs an `out` operation and writes a matching tuple.

Linda has been employed in several contexts requiring coordination, either embedded in other programming languages or provided as an external library [11, 12, 13, 28]. As instances, C-Linda [10] introduces in the C language the possibility of directly using Linda primitives, while T-Spaces [22] and JavaSpaces [21] are examples of Java implementation of Linda tuple spaces.

By performing a mapping of Linda concepts into ERESYE, it can be easily noted that an ERESYE engine without rules is able to store Erlang tuples offering an API to handle such a tuple repository (the fact base); in particular, functions `assert`, `wait` and `wait_and_retract` behave exactly like Linda's `out`, `rd` and `in`, respectively. Such functions can be used by several concurrent processes to access the engine, as Figure 4 highlights, thus making coordination activities possible. While function `assert` has already been described, API function `wait` allows a calling process to suspend itself until a fact with a given template/pattern is asserted in a given engine. The template is specified by means of a tuple where each element can be *any constant* used to indicate an actual parameter; the special constant `'_'`, to indicate any value; or a *fun*, to specify a more complex matching expression (e.g. a predicate on the type, like `is_number`, `is_list`, `is_atom`, etc.). When such a fact is asserted, the control is given back to the calling process, provided that the function returns the matching fact. Function `wait_and_retract` behaves similarly but also

```
-module (phil).
-compile ([export_all]).
-define (N_PHIL, 5).

start () ->
  eresye:start (restaurant),
  phil_spawn (0).

phil_spawn (?N_PHIL) -> ok;
phil_spawn (N) ->
  eresye:assert (restaurant, {fork, N}),
  spawn (phil, philosopher, [N]),
  if
    N < (?N_PHIL - 1) ->
      eresye:assert (restaurant, {room_ticket, N});
    true ->
      ok
  end,
  phil_spawn (N + 1).

philosopher (N) ->
  think (),
  Ticket = eresye:wait_and_retract (restaurant,
                            {room_ticket, '_'}),
  eresye:wait_and_retract (restaurant, {fork, N}),
  eresye:wait_and_retract (restaurant,
                     {fork, (N + 1) rem ?N_PHIL}),
  eat (),
  eresye:assert (restaurant, {fork, N}),
  eresye:assert (restaurant,
                     {fork, (N + 1) rem ?N_PHIL}),
  eresye:assert (restaurant, Ticket),
  philosopher (N).
```

Figure 8: The Linda Dining Philosophers Solution implemented with ERESYE

performs retraction of the matching fact, thus emulating Linda in primitive. An example of the use of this functions in a Linda-like scenario is given in Figure 8, demonstrating the Erlang/ERESYE porting of the C-Linda dining philosophers solution provided in [10].

Such an ability to emulate Linda tuple spaces is another important contribution of ERESYE, since it is able to introduce coordination artifacts in Erlang programs while preserving the "share nothing" semantics of the Erlang concurrency model. In fact, interaction of processes with the tuple space (ERESYE fact base) is performed by means of message passing, while tuples (facts) are passed by-value in accordance with concurrency concepts of Erlang. Moreover, by combining Linda tuple space emulation with rule definition and processing, coordination functionality can be enhanced by allowing the realization of *reactive tuple spaces* [13, 28], i.e. tuple spaces where the assertion of specific tuples is able to trigger user-defined computations.

4 Examples

In order to better understand the characteristics and capabilities of ERESYE, this Section will show two examples.

The first one, which is listed in Figure 9, is an ERESYE version of the sieve of Eratosthenes, implemented using the "multiset rewriting" principle of the GAMMA coordination model [7]. The basic working scheme is the following: gather all the numbers from 2 to N (100 in our example) and remove all multiples by comparing all possible pairs. It is not an efficient solution, but we chose such an example to show the ability of ERESYE to support coordination abstractions based on multiset rewriting [7, 9], indeed the same example is used to show how to program in GAMMA. As the listing shows, rule remove_multiple is triggered for each X, Y pairs and, as the action, it retracts X if it is a multiple of Y. Rule final_rule is instead executed if the fact {is, finished} is not present in the fact base; since this second rule has a priority less than the first rule (see lines 22 and 23 of the listing), it will be executed only when there will be no more instances of rule

```
-module (sieve).
-compile ([export_all]).

remove_multiple (Engine, {X}, {Y}) when ((X rem Y) == 0) and (X =/= Y)->
  eresye:retract (Engine, {X}).

final_rule (Engine, {is, started} = X)
    when not ["{is, finished}"];true ->
  eresye:retract (Engine, X),
  eresye:assert (Engine, {is, finished}).

start () ->
  eresye:start(sieve),
  eresye:assert(sieve, [{X} || X <- lists:seq(2, 100)]),
  eresye:assert(sieve, {is, started}),
  eresye:add_rule(sieve, {sieve, remove_multiple}, 2),
  eresye:add_rule(sieve, {sieve, final_rule}, 1),
  eresye:wait_and_retract(sieve, {is, finished}),
  io:format("~p~n", [eresye:get_kb(sieve)]),
  eresye:stop(sieve),
  ok.
```

Figure 9: The ERESYE version of the Sieve of Eratosthenes

`remove_multiple`. Therefore, `final_rule`, by asserting the fact {`is, finished`}, will signal that there are no more multiples: Thus the fact base will contain only prime numbers.

The second example is a very simple expert system to guide authors in choosing the journal for the submission of a paper. In this example, we wrote a `journal` ontology, shown in Figure 10, to allow a classification of journals on the basis of the publisher and the circulation ("national" or "international"). The working scheme of the expert system is based on making a series of questions to the user regarding the topics of the paper to submit, a possible choice of the publisher, a possible choice of the circulation, and a minimum impact factor. The source code (too long to be depicted here) uses 13 rules. A sample run of such an expert system is shown in Figure 11.

5 Comparing ERESYE and CLIPS

In this Section, a brief comparison between CLIPS and ERESYE will be provided, aiming at proving that both the tools exhibit the same features for the realization of expert systems. In performing such a comparison, we will take into account the capabilities and flexibility of both tools in *representing the knowledge* and *expressing production rules*.

As for the first item—*knowledge representation*—as reported in Section 2, CLIPS represents facts using an (ordered) list of symbols enclosed within parentheses. It also supports non-ordered facts (i.e. facts whose fields are named) and objects, which can be organized in a class hierarchy. It is easy to check that Erlang provides similar structures for ERESYE facts, i.e. *tuples* for ordered facts and *records* for non-ordered ones. Objects and classes are instead supported in ERESYE thanks to ontology definition and handling as reported in Section 3.3.

As for the specification of production rules, the basic definition scheme, which implies a series of templates where *all* must match facts in order to activate the rule, is supported in ERESYE. In addition to this basic scheme, CLIPS provides other logic connectives to improve flexibility in the rule head; in fact, matching expressions in conditional elements can be made more expressive by using `or` and `not` connectives. Such additional features can also be provided by ERESYE: `or` connectives in a rule head (see lines 1–8 in Figure 12a) can be supported in ERESYE thanks of the use of multiple clauses for the same rule, as lines 1–8 in Figure 12b shows. Even if the syntax is not exactly the same of CLIPS, the semantics is maintained.

Other connectives provided by CLIPS, called *connective constraints*, are used to implement more complex tests for *single fact elements*; they are the symbols `&`, `|` and `~`, which represent "and", "or" and "not" logical operation respectively, as the examples in lines 8–18 of Figure 12a shows (the

```
class (journal) ->
 { title = [string, mandator, nodefault],
   main_topic = [string, mandator, nodefault],
   publisher = [string, mandatory, nodefault],
   type = [string, mandatory, nodefault],
   impact_factor = [number, mandatory, nodefault] };

class (’international-journal’) ->
  is_a (journal),
 { type = [string, mandatory,
           default(international)] };

class (’national-journal’) ->
  is_a (journal),
 { type = [string, mandatory, default(national)] };

class (’IEEE-journal’) ->
  is_a (’international-journal’),
 { publisher = [string, mandatory, default(’IEEE’)] };

class (’ACM-journal’) ->
  is_a (’international-journal’),
 { publisher = [string, mandatory, default(’ACM’)] };

class (’italian-journal’) ->
  is_a (’national-journal’).
```

Figure 10: The “journal” ontology

comments also give a brief explanation of the example). Such connectives can be supported in ERESYE by combining the “not” syntax with multiple rule clauses, as shown in the ERESYE translation of the examples depicted in lines 8–18 of Figure 12b.

Connective constraints are also enriched with the possibility of invoking predicate functions (or mathematical/logical predicate expression), which, if they evaluate to true, trigger rule firing: Significant examples of such a feature are shown in lines 20–30 of Figure 12a. We can easily note that such connective constraints correspond to *guards* and can thus be supported in ERESYE thanks to the `when` Erlang keyword, as lines 20–30 of Figure 12b show. The only limitation is with user-defined functions, which can be employed in CLIPS connective constraints, but are instead illegal in Erlang guards, as only BIFs can be used. This is a characteristic of the language that (as widely known in the Erlang community) is tied to side-effect avoidance in guard execution. However, this is not a so tight limitation, since, in general, user-defined functions employed in expert systems are related to simple operations, such as type checking, more or less complex value comparisons, etc., which can be as well supported by Erlang BIFs.

A final remark is the integration of an expert system engine to other (external) programs. We already discussed the drawbacks of CLIPS in Section 2.2, which force programmers to statically embed CLIPS source code in their applications. In ERESYE instead, the situation is quite different and more flexible: Applications interact with ERESYE engines through the Erlang standard message passing mechanism, so Erlang programs can use the API functions of the `eresye` module, while C/C++ or Java programs can use the libraries provided to interact with Erlang processes, such as `jinterface` or `EPI` [6].

6 Conclusions

This paper has presented ERESYE, an Erlang tool for the development of rule-based expert systems, allowing Erlang programs to include reasoning capabilities. Such a tool opens new and interesting application areas to this language. Indeed, the main contribution of ERESYE is the possibility of directly employing Erlang constructs (i.e. functions and data types) to represent knowledge and rules, allowing the design of inference systems that can be tightly integrated with an Erlang application. A similar characteristic is not exhibited by other programming languages,

```
Erlang (BEAM) emulator version 5.4.3 [source] [hipe]

Eshell V5.4.3  (abort with ^G)
1> journal_sample:start().
Enter topic of your paper ("q" to end): internet
Enter topic of your paper ("q" to end): AI
Enter topic of your paper ("q" to end): q
Select the publisher (ieee/acm/any): any
Select the type (i)nternational/(n)ational): i
What is your minimum impact factor: 0.3
The journal is {'international-journal',
                   'Trans. on Internet Computing',
                   internet,
                   'ACM',
                   international,
                   ...}
The journal is {'international-journal',
                   'Internet Computing',
                   internet,
                   'IEEE',
                   international,
                   ...}
The journal is {'international-journal',
                   'Trans. on Pattern Analysis ....',
                   'AI',
                   'IEEE',
                   international,
                   ...}
ok
2>
```

Figure 11: An Execution of the Journal Selection Expert System

such as C, C++ or Java, that, for the same purpose, need the integration of external tools featuring a different programming approaches to write inference rules and facts. All of these concepts and results have been illustrated through the whole paper. As a proof of concept, a comparison with CLIPS, the expert system tool often used in conjunction with C programs, is provided, highlighting that even if CLIPS and ERESYE feature syntactical differences, by using proper rewriting guidelines any CLIPS concept (fact, object or rule) can be also expressed in ERESYE, thus proving the equivalence of the tools, as well as the effectiveness of the ERESYE solution.

References

[1] http://herzberg.ca.sandia.gov/jess/. JESS Web Site, 2003.

[2] http://www.ghg.net/clips/CLIPS.html. CLIPS Web Site, 2003.

[3] http://www.diit.unict.it/users/csanto/exat/. eXAT Web Site, 2004.

[4] http://www.drools.org. Drools Home Page, 2004.

[5] JSR-000094 Java^{TM} Rule Engine API, http://www.jcp.org/aboutJava/communityprocess/review/jsr094/, 2004.

[6] Erlang Plus Interface, http://epi.sourceforge.net, WWW, 2005.

[7] J. Banatre and D. LeMetayer. The Gamma Model and its Discipline of Programming. *Science of Computer Programming*, 15:55–77, 1990.

[8] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, May 2001.

***(a)* CLIPS**

```
(defrule systemFlow
  (error confirmed)
  ( or ( and (temp high) (valve closed))
       ( and (temp low) (valve open)))
  =>
  action)

(defrule example1
  % match all “(data X)” but “(data blue)”
  (data ~blue) => ... )

(defrule example2
  % match all “(data X)” but “(data red)” and “(data green)”
  (data ~red & ~green) => ... )

(defrule example3
  % match “(data red)” or “(data green)”
  (data green | red) => ... )

(defrule example4
  % match “(data X)”, given that X is a number
  (data ?x & :(numberp ?x )) => ... )

(defrule example5
  % match “(data X Y)”, given that Y is twice X
  (data ?x ?y=(* 2 ?x )) =>... )

(defrule example6
  (data ?x) (value ?y)
  (test (>= ( abs (- ?x ?y)) 3)) => ... )
```

***(b)* ERESYE**

```
systemFlow (Engine, {error, confirmed},
            {temp, high}, {valve, closed}) ->
  action;
systemFlow (Engine, {error, confirmed},
            {temp, low}, {valve, open}) ->
  action.

example1 (Engine, {data, X})
  when not ["{data, blue}"]; true -> ...


example2 (Engine, {data, X})
  when not ["{data, red }",
            "{data, green }"]; true -> ...

example3 (Engine, {data, red}) -> ...;
example3 (Engine, {data, green}) -> ...


example4 (Engine, {data , X})
  when is_number (X) -> ...


example5 (Engine, {data, X, Y})
  when X==2 * Y -> ...


example6 (Engine, {data, X}, {value, Y})
  when abs(X - Y) >= 3 -> ...
%
```

Figure 12: CLIPS/ERESYE Comparison

[9] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[10] N. Carriero and D. Gelernter. Linda in Context. *Comm. ACM*, 32(4), April 1989.

[11] P. Ciancarini. Coordination Models and Languages as Software Integrators. *ACM Computer Surveys*, 28(2), 1996.

[12] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating Multiagent Applications on the WWW: A Reference Architecture. *IEEE Transaction on Software Engineering*, 24(5), 1998.

[13] M. Cremonini, A. Omicini, and F. Zambonelli. Coordination and access control in open distributed agent systems: The TuCSoN approach. In A. Porto and G.-C. Roman, editors, *Coordination Languages and Models*, volume 1906 of *LNCS*, pages 99–114. Springer-Verlag, 2000.

[14] A. Di Stefano and C. Santoro. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. In *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2003)*, Villasimius, CA, Italy, 10–11 Sept. 2003.

[15] A. Di Stefano and C. Santoro. eXAT: A Platform to Develop Erlang Agents. In *Agent Exhibition Workshop at Net.ObjectDays 2004*, Erfurt, Germany, 27–30 Sept. 2004.

[16] A. Di Stefano and C. Santoro. Designing Collaborative Agents with eXAT. In *ACEC 2004 Workshop at WETICE 2004*, Modena, Italy, 14–16 June 2004.

[17] D. Fensel. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer, 2004.

[18] C. Forgy. OPS5 Users Manual. Technical Report CMU-CS-81-135, Dept. of Computer Science, Carnegie-Mellon Univ., 1981.

[19] C. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, pages 17–37, 1982.

[20] C. Forgy. The OPS Languages: An Historical Overview. *PC AI*, Sept. 1995.

[21] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces(TM) Principles, Patterns, and Practice*. Sun Microsystems, 1999.

[22] IBM. *TSpace Intelligent Connectionware*. WWW, `http://www.almaden.ibm.com/cs/TSpaces/`, 2005.

[23] J. Bradshaw et al., editor. *Software Agents*. AAAI Press, Cambrigde, Mass., 1997.

[24] C. Santoro. *eXAT: an Experimental Tool to Develop Multi-Agent Systems in Erlang - A Reference Manual*. Available at `http://www.diit.unict.it/users/csanto/exat/`, 2004.

[25] W3C. *OWL Web Ontology Language Overview*. WWW, `http://www.w3.org/TR/owl-features/`, 2004.

[26] W3C. Resource Description Framework (RDF): Concepts and Abstract Syntax, 10 Feb. 2004.

[27] M. J. Wooldridge. *Reasoning About Rational Agents*. The MIT Press, July 2000.

[28] F. Zambonelli, N. Jennings, A. Omicini, and M. Wooldridge. Agent-oriented software engineering for Internet applications. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, chapter 13, pages 326–346. Springer-Verlag, Mar. 2001.