A symbolic layer for autonomous component-based software agents

Razvan Dinu[1] and **Tiberiu Stratulat**[2] and **Jacques Ferber**[3]

Abstract. In order to handle complex situations, an autonomous agent needs multiple components ranging from simple input/output modules to sophisticated AI techniques. Integrating a high number of heterogeneous components is a non-trivial task and this paper discusses the use of a symbolic layer to address this issue. After an overview of existing techniques and their limitations this paper proposes a new approach through a generalized hyper-graph model in which the interaction of different components is modeled through a triggering mechanism based on patterns. Finally, the paper shows how a flexible symbolic middleware can be built and a few examples are presented.

1 Introduction

In order to keep up with the increasingly complex real-world problems, autonomous software agents need to integrate more and more components that range from simple input/output modules to sophisticated AI techniques. As the number of components increases the integration itself becomes an issue which unfortunately has been neglected until recent years. More and more researchers agree that *"the question about the inner workings of the pieces themselves holds equal importance to the question about the nature of the various dynamic glues that hold the pieces together"* [15].

When integrating multiple components, two levels of integration can be distinguished: *generic* and *specific*. The generic level is concerned with general mechanisms such as *how* components communicate with each other and *how* they exchange data. The specific level is concerned with the details of integrating components X_1, X_2, ..., X_n, of specific types, such as *when* component X_i calls a function of component X_j, *what* data should X_i provide, *when* should X_i send the data to X_j, etc.

Usually, in a running software agent, the generic level takes the form of a middleware and provides primitives for data and control flow to different specific levels. Such a middleware has to provide solutions to three main challenges: *communication*, *data sharing* and *global control*. The communication challenge is concerned with how different components can reach each other and how can they use each other's functionalities. The data sharing is concerned with how components can provide data (content) to other components. Global control is concerned with how all the interactions between components are handled and how a coherent global behaviour of the agent can be achieved.

One traditional technique for generic integration of multiple components is the *blackboard system* in which a set of experts, also called knowledge sources (KS), are constantly monitoring a blackboard searching for an opportunity to apply their expertize. Whenever they find sufficient information on the blackboard they apply their contribution and the process continues [6]. Unlike other techniques that implement formal models, the blackboard approach was designed to deal with ill-defined complex interactions. One of the first applications of the blackboard system was the speech understanding HEARSAY-II system [7] in which multiple components used a shared blackboard to create the required data structures.

Another generic integration technique is based on message passing and usually uses a publish-subscribe mechanism in which components subscribe to different types of messages and whenever a message arrives it is forwarded to corresponding modules. A message-based communication protocol for AI that has been gaining in popularity in recent years is the OpenAIR protocol managed by mindmakers.org [2].

CORBA is a well known standard by OMG [5], according to which components written in multiple computer languages and running on multiple computers are exposed as objects and their interaction is performed by method invocation. CORBA is very used as system integration in humanoid robotics, see for instance the simulator OpenHRP [10].

All of the above techniques provide more or less solutions to the three challenges mentioned earlier. Blackboards clearly provide means for data sharing, enables communication between components indirectly, but the control component is usually a simple scheduler and it does not help much in assuring a coherent global behaviour of the agent. On the other hand, message-passing focuses on communication and object-oriented techniques on communication and somewhat data sharing. Both leave global control entirely up to the interacting components.

Both improvements and hybrid solutions have been proposed for the above techniques. For example, whiteboards [16] consist of a blackboard with (i) a general-purpose message type, (ii) ontologically defined message and data stream types and (iii) specification for routing between system components. They also add an explicit temporal model thus providing more specialized solutions for communication and data sharing challenges. Also, the GECA Framework (Generic Embodied Conversational Agent) uses a hybrid solution in which multiple blackboards are used to perform message-passing based on message types [11].

Our opinion is that components integration would be much easier if we had an integration technique based on a more expressive data model and which provided better support for different *patterns of global control*.

By pattern of global control we understand the most abstract model that can be used to explain the behaviour of the software agent.

[1] University of Montpellier 2, LIRMM
[2] University of Montpellier 2, LIRMM
[3] University of Montpellier 2, LIRMM

A classical example of such a pattern, especially used in robotics, is the Brooks subsumption architecture [4]. In this approach components are structured into layers and those situated at higher levels are capable of altering the input and inhibiting the output of components at lower levels.

Another very widely used pattern of global control, especially in multi-agent systems, is BDI (Beliefs Desires Intentions) [14]. The software agent maintains a set of beliefs based on which desires are created. A desire which the agent has decided to pursue becomes an intention and a plan is chosen to achieve the desired goal.

More sophisticated patterns of global control come from the agent architectures domain. For example the INTERRAP agent architecture [13] uses three control layers: Behaviour-based layer, Local Planning layer and Cooperative Planning layer. Each layer has its own world model and includes subcomponents for situation recognition, planning and scheduling.

When we say that the generic integration middleware should support patterns of global control such as the ones mentioned above we are not saying that the patterns should be entirely implemented inside the middleware. But rather, the middleware should contain only part of the pattern and should smoothly integrate with components implementing key aspects of the control pattern (for instance a planning engine).

This paper focuses on the generic level of integration and proposes a middleware model that enables easier and more straightforward integration of different AI and non-AI components of an autonomous software agent. The next section introduces our approach and sections 3, 4 and 5 introduce our new symbolic model for generic integration and also perform a preliminary evaluation of its performance. Section 6 presents an implementation for smart phones based on the Android platform and finally, section 6 and 7 present our comments and conclusions.

2 Approach

As it has been outlined in the previous section, the main shortcomings for current approaches concern the *data sharing* and *global control* challenges. Our approach is an extension of the blackboard model which addresses exactly these two challenges.

2.1 Data sharing challenge

Firstly, we propose that the blackboard uses a more expressive symbolic data model rather than just isolated bits of typed data. The chosen symbolic structure is inspired by the generalized hyper-graph model proposed by [3]. Hyper-graphs generalize normal graphs by allowing an edge to contain more than two nodes and a directed hyper-graph considers edges as ordered sets (tuples). We are interested in a generalization of directed hyper-graphs in which an edge can contain both nodes and other edges. This represents the generalized hyper-graph model we're using and it will be described in more details in the next section. However, we will be using a different terminology that makes more sense in the context of symbolic representations: *symbols* instead of nodes and *links* instead of hyper-edges.

Secondly, we extend the generalized hyper-graph structure with a *map* which associates each symbol of the hyper-graph with another symbol. Finally, we allow each symbol to have some attached information, which can be typed or not.

A generalized hyper-graph, the information associated with the symbols and the map of symbols form a SLiM structure (Symbol Link Map). From now on, we will use the capital version (SLiM) to refer to the model and the lower letter version (*slim*) as a shorthand for "SLiM structure" which refers to a concrete structure.

One related work which uses a hyper-graph model close to ours is [12]. They use a directed hyper-graph and integrate a typing system in which a node has a handle, a type, a value and a target set. The main differences in our model are the lack of the typing system and the addition of the symbolic map which, as it will be shown in future sections, can be used to create a typing system. However, they show how such a hyper-graph structure can be efficiently implemented and used as central database especially in AI applications. The OpenCog project [9] is also an illustrative example of hyper-graphs usage in AI projects. These works show the increasing interest of using the flexible hyper-graphs structures in AI.

2.2 Global control challenge

In order to address the issue of *global control* we inspired ourselves from the patternist philosophy of mind whose main premise is "the mind is made of patterns". In this perspective a mind is a collection of patterns associated with persistent dynamical processes capable of achieving different goals in an environment. For a quick overview of the patternist philosophy of mind and also a different way of applying it in the context of AI we recommend [8].

We define a pattern as a particular type of slim and we show how a set of patterns can be efficiently matched using an automaton. Next, we propose an interaction mechanism between components based on patterns that uses a central SLiM structure which can by accessed and modified by any component. Each component can register two types of patterns: data patterns and capability patterns. Whenever a component modifies the central slim and a data pattern is found then the corresponding component is notified. Also, whenever a component requests the execution of something that matches a capability pattern then the corresponding component is notified.

As it will be detailed in the following section all these mechanisms provide a very flexible way of performing interaction between different components of a software agent and they can be packed into a symbolic middleware which can be used in conjunction with other agent frameworks.

3 The SLiM Model

This section formally introduces the SLiM model and also proposes a representation language to represent a slim.

3.1 Formal definition

Definition 1. Let S be a finite set of elements. T_S is the set of all tuples over S and it is inductively defined as:

- $T_0 = \{(0, \emptyset)\}$.
- $T_k = \{X \cup \{(k, s)\} | X \in T_{k-1}, s \in S\}$ for $k \geq 1$
- $T_S = \cup_{k=0}^{\infty} T_k$

The sole element of T_0 is called the empty tuple and will be denoted simply by $\emptyset$. An element $t \in T_k$ is called a tuple of length k. Instead of $t = \{(0, \emptyset), (1, s_1), ..., (k, s_k)\}$ we use the equivalent notation $t = (s_1, s_2, ..., s_k)$. We also use the notation $s \in t$ to mean $\exists j \geq 1$ such that $(j, s) \in t$.

Definition 2. Let the following:

i. S be a finite set of elements called *symbols*.

ii. $l\colon S \to T_S$ be a function called a *linking* function on S.
iii. $i\colon S \to I$ be a function called an *information* function on S, where I is a set of elements.
iv. $m\colon S' \to S$, where $S' \subset S$, be a partial function on S called a *map* on S.

Then the quadruple $< S, l, i, m >$ is called a SLiM structure or simply a *slim*. The elements of $s \in S$ for which $l(s) \neq \emptyset$ are also called *links* and if $x, y \in S$ and $m(x) = y$ we say that x is mapped to y.

Below are a few terminological definitions associated with the SLiM model.

Definition 3. A symbol $d \in S$ is *reachable* from a symbol $s \in S$ if and only if there exist $s_1, s_2, ..., s_n \in S$ such that $s_1 \in l(s), s_2 \in l(s_1), ..., s_n \in l(s_{n-1})$ and $d \in l(s_n)$.

Definition 4. A symbol $d \in S$ is *mappable* from a symbol $s \in S$ if and only if d is equal to s or there exist $s_1, s_2, ..., s_n \in S$ such that $m(s) = s_1, m(s_1) = s_2, ..., m(s_{n-1}) = s_n$ and $m(s_n) = d$.

Definition 5. A tuple $(s_1, s_2, ..., s_n) \in T_S$ is called an *implied link* if and only if there exist $x_1, x_2, ..., x_n, y \in S$ such that x_1 is mappable from s_1, ..., x_n is mappable from s_n and $l(y) = (x_1, x_2, ..., x_n)$. If $x_i = s_i$ for $i = 1, n$ then the link is called *explicit*.

Definition 6. A slim $< S, l, i, m >$ is called *acyclic* if and only if no symbol can be reached from itself.

3.2 Representation language

Before going any further we will introduce an abstract syntax for a representation language, called the *SLiM language*, that can be used to describe a slim. Given S and I the sets of symbols and information elements, the language is given by the following EBNF:

`slim`	$\to$	`symbol+`	(1)
`symbol`	$\to$	`[id|link][:[info|symbol]]?`	(2)
`link`	$\to$	`[id=]?{symbol+}`	(3)
`id`	$\to$	$s \in S$	(4)
`info`	$\to$	$x \in I$	(5)

(the curly brackets are part of the terminal alphabet of the SLiM language)

Before giving a few examples we will provide the semantics of the production rules. In order to do that we consider that the non-terminal nodes of the grammar `symbol`, `link` and `id` have a synthesized attribute "s" which holds the corresponding symbol $s \in S$:

Production rule			**Attribute rule**
`symbol`	$\to$	`id[...]?`	`symbol.s = id.s`
`symbol`	$\to$	`link[...]?`	`symbol.s = link.s`
`link`	$\to$	`id={symbol+}`	`link.s = id.s`
`link`	$\to$	`{symbol+}`	`link.s =` *use/new*
`id`	$\to$	$s \in S$	`id.s =` s

The *use/new* keyword means that if there is already a link corresponding to the sequence of symbols on the right side then the attribute `link.s` uses the same id, otherwise it gets a random id from S not used by any other production rule. Below we will give the semantics of each of the right sides of production rules 2 and 3.

Right side	**Semantics**
`{`s_1 `...` s_n`}`	$l(\textit{use/new}) = (\mathrm{s}_1.\mathrm{s}, ..., \mathrm{s}_n.\mathrm{s})$
`id={`s_1 `...` s_n`}`	$l($`id.s`$) = (\mathrm{s}_1.\mathrm{s}, ..., \mathrm{s}_n.\mathrm{s})$
`[id|link]:symbol`	$m($`[id|link].s`$) =$`symbol.s`
`[id|link]:info`	$i($`[id|link].s`$) =$`info`

Here's an example of a slim described using the SLiM language:

```
here={my location}          (1)
here:{city Lyon}            (2)
{user said msg:"Hello"}     (3)
```

Let $< S, l, i, m >$ be the slim described in the above example. The symbols set is $S = \{$ `my`, `location`, `here`, `city`, `Lyon`, `user`, `said`, `msg`, `rand1`, `rand2` $\}$ and the information set is $I = \{$ `null`, `"Hello"` $\}$. The first line creates a link between the symbols `my` and `location` and assigns the id `here`, which means $l($ `here`$) = ($`my`,`location` $)$. The second line creates a link between `city` and `Lyon` whose id is not important (and we can consider it to be `rand1` $\in S$) and maps the symbol `here` to it. This means $l($ `rand1`$) = ($`city`,`Lyon` $)$ and that $m($ `here`$) =$`rand1`. The third line creates a link between other three symbols and assigns some information to the last one, $i($ `msg`$) =$`"Hello"`. All other symbols $s \in S$ have $i(s) =$`null`.

The meaning of the first production rule is the union of all the symbols, information and mappings defined by the `symbol` production rules. We say that a SLiM representation (a string in the SLiM language) is *valid* if and only if there are no contradictions (i.e. a symbol being assigned two different information, a symbol being mapped to different symbols, multiple definitions of a link, etc.). From now on, we will use the SLiM language rather than the formal definition to describe SLiM structures.

One last observation concerns the use of curly brackets to create links. The language does not use parentheses or square brackets in order to avoid confusion with languages such as LISP or REBOL.

3.3 Modeling with SLiM

The SLiM model does not impose any particular semantics on the data being represented in a slim. It is a semi-structured, general purpose model based on a generalized hyper-graph structure. The actual schema that will be used in a slim will evolve dynamically and data integrity constraints can be enforced by different components using the slim. This type of flexible models is especially suitable for online environments.

4 Patterns

As it was discussed in section 2, the pattern concept is central to our approach. Below we will explain what a pattern is, how can multiple patterns be matched, and we perform a simple performance evaluation of the proposed matching mechanism.

4.1 Definition

Definition 7. `?` and `@` are two special symbols called the *any* and the *root* symbols.

Definition 8. A *pattern* is an acyclic slim $< S, l, i, m >$ with the following properties:

i. `?` $\in S$ and `@` $\in S$;
ii. `@` is mapped to a symbol, which is called the *root of the pattern*, and all other mappings, if any, are to `?`;

iii. the symbols mapped to `?` are called *generic* symbols and they are all reachable from the root of the pattern;
iv. it contains no information ($I = \{\emptyset\}$).

Before discussing the different properties from the above definition we will provide two examples, described using the SLiM language, so that the reader can have a better grasp of what patterns look like. Each pattern has been enclosed inside an additional set of curly brackets:

```
{
    {sound enabled}
    @: { notify user Message:? }
}
{
    online
    @: {User:? wants {listen album Album:?} }
    {User allowed music}
}
```

Figure 1. "Examples of patterns"

First of all, we are only interested in patterns at the symbolic level, disregarding the information attached to symbols, hence property iv). Secondly, the SLiM model is intended to represent data mainly through symbols and links and that is why patterns will be used to describe only parts of the symbolic hyper-graph. As a consequence, a pattern contains only mappings that have special meaning to the matching mechanism.

In a few words, a pattern is a generic way of describing a set of symbols and links. A pattern can contain regular symbols or *generic* symbols (those mapped to `?`) which act as place holders for regular symbols. For convenience, we can omit the "`@:`" for simple patterns. Also, we can write directly {`listen album ?` } if the name of the generic symbol is not relevant when presenting a pattern.

Intuitively the first example in figure 1 can be matched for example by the following slim:

```
some-message: "Hello User!"
  {sound enabled}
  {notify user some-message}
```

The generic symbol `Message` is a place holder for the `some-message` symbol. The link {`sound enabled`} exists as it is.

In order to explain the role of the `@:` mapping we have to define exactly what is a match for a pattern.

4.2 Matching a pattern

Definition 9. A slim M is a match for a pattern P if and only if it can be obtained from P by:

1. replacing all mappings to `?` with mappings to other non-generic, possibly new, symbols;
2. replacing the occurrences of all generic symbols in links with the symbol they're mapped to;
3. removing the mapping from `@` and the symbols `?` and `@`.

Definition 10. Let X be a slim, P be a pattern and M be a match of P. We say that M is a match of P in X if and only:

i. every symbol in M which is not generic in P also exists in X;
ii. every link in M is implied in X.

For example, the following slim (M):

```
User: current-user
Album: s32
online
{current-user wants {listen album s32} }
{current-user allowed music}
```

is a match for the second example pattern in the following slim (X):

```
current-user: John
{s32 author Michael-Jackson}
{s32 title s41:"Bad"}
online
{current-user wants {listen album s32} }
{John allowed music}
```

We can easily see that the symbols such as `online`, `wants`, `music`, etc. exist directly in X and so do links such as {`listen album s32`} . On the other hand the link {`current-user allowed music`} is only implied in X (by definition 5) because `current-user` is mapped to `John` and we have the link {`John allowed music`}.

One important aspect of matching a pattern is the fact that the links described by the pattern must be implied in the slim we're searching, and not necessarily exist. This gives a lot of flexibility when modeling data we slim. We can choose to leave some links implicit and still be able to match them in patterns. A more in-depth discussion of implied links would be suitable but due to space limitation we will leave it to the reader to imagine how implied links can be used.

Searching a set of symbols and links that match a pattern in a slim can be a very time consuming task especially when generic symbols are used in more than one link. It is the equivalent of a join operation on a relational database which requires special indexing in order to be processed efficiently. That is why we divide a pattern in two, the root of the pattern and the rest, and we impose that the generic symbols are reachable from the root.

If the root is a link and some of the symbols inside the link are also links and so on, we have an ordered tree[4] of symbols determined by the root of the pattern. We will refer to this tree as *the tree of the pattern* (figure 2 shows the tree for the second example pattern).

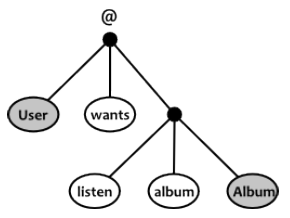

Figure 2. Example of "tree of a pattern"

Since all the generic symbols are reachable from the root, in order to find a match for a pattern we first have to find a match for the tree of the pattern. If one is found, all the other symbols and links can be easily checked since there are no other generic symbols. But even

[4] a tree in which the order of the sons is important.

finding a match for a tree in a slim can take a very long time so we will limit our approach to a particular case which will be used in our symbolic middleware presented in the next section.

The pattern matching problem we are addressing is the following: *given a symbol in a slim and a pattern tree, can it be matched starting at the given symbol (the root of the instance tree will correspond to the given symbol)*?

We will represent a pattern tree using the simplified notation (no `@` and no generic symbols names). For example, for the second example pattern we get {`? wants` {`listen album ?`} } . This will be called the normal string representation of a pattern tree. So, the question is, given such a representation and a symbol (which can be a link) in a slim, can the tree be matched with the given symbol as root? This can actually be done in liner time by reading the tree pattern representation from left to right and by performing a depth-first navigation of the symbol in parallel. The algorithm is left as an exercice to the reader.

4.3 Summary

In order to conclude this section we will summarize the main idea: we have a set of patterns and a symbol in a slim and we are interested to see if there's a pattern whose tree can be matched with its root at the given symbol; if so, then the rest of the pattern can be easily checked once we have the values of all the generic symbols in the pattern; if all other symbols and links exist then we have found a match for a pattern.

5 SLiM based integration middleware

Now we will show how the symbolic SLiM module introduced in the previous section can be used in what we call *symbolic integration middleware*.

5.1 Symbolic middleware

We consider a software agent as having two parts: a *crown* which contains many different components and a *trunk* which contains one or more middlewares. The only way components can interact is through a middleware situated in the trunk. The SLiM middleware, which is the proposed solution for components integration, is composed of:

- A *slim* which acts as shared blackboard. Every component is able to create new symbols, modify links, attach information to symbols, change mappings, etc.
- A *capabilities index* and a *triggers index*. They are two sets of patterns together with two automata capable of matching them.
- A *behaviour rules* set. A rule is an association between a trigger pattern and an *entry-slim* (a slim with a designated symbol called *entry point*).

Every component can register one or more capability or trigger patterns with the SLiM middleware. For example a text-to-speech component can register the capability pattern {`speak ? english`}. A natural language processing component can register the trigger pattern {`user said ?`}.

In order to illustrate the role of each type of pattern or rule we will explain the different modes in which the SLiM middleware can be used.

***tell* mode**. In this mode a component can access the shared slim and perform whatever changes it needs. For example in this mode the speech recognition module would add {`user said msg:'Hello world!'`} .

***do* mode**. In this mode a component asks the SLiM middleware to do something by providing an entry-slim. The middleware will try to match the given slim, starting from the entry point, by a capability pattern. If one is found then the component that registered the pattern will be notified and it will be provided the match. If no pattern is found and the entry point is a link then all the symbols in the link will be successively used as entry points, creating a sequence of *do*-s. For example a component can request {{`wait 5 seconds`} {`speak msg:'Hello you too!' english`}} to be done.

***trigger* mode**. Whenever a symbol is created or a link made, the triggers automaton will try to match it to an existing trigger pattern. If one is found and it is associated with a component then the component will be notified. However if it is associated with an entry-slim, through a behaviour rule, a *do* will be requested on the entry-slim. For example the rule {`user said ?`} $\rightarrow$ {`log to history ?`} logs what the user said to a history.

***ask* mode**. This mode is not discussed in this paper but it allows a module to query a slim structure in the same manner as [12].

Other details such as how symbols from a trigger pattern match are used in the entry-slim of a behavior rule, details of the *do* mode or the role of mappings, which some readers might consider important, are not discussed in this paper.

Let's take now the two example patterns provided in previous section (figure 1). The first one could be registered by a component capable of producing some sounds, perhaps depending on the type of the message. However, the pattern also contains the link {`sound enabled`} . We can imagine a convention like {`sound enabled`} when components can produce sounds and {`sound disabled`} when they can't (the volume is set to 0). So, if a component wants to send a notification to the user it will request a *do* on the following slim {`notify user msg12:"New email!"`}. At that point the capability index will try to match the given slim, starting from its entry point which is the whole link, and will find the example pattern 1 as being a match, the corresponding component will be notified and for example the user will hear a beep.

The second pattern is more complicated and it can correspond to a component capable of playing albums from internet for example. But this pattern will be registered as a trigger pattern and not a capability pattern. For example a speech recognition component combined with a natural language processing module can recognize that the user wants to listen to an album and it will create a link stating that fact. When the link is created the trigger automaton will analyze the link and if the user is online and is allowed to play music than a match is found and the player component gets notified. If no component would have registered the second example pattern than the link created by the natural language processing module would have no effect and maybe it will be removed by a component that deletes links unused for a certain amount of time.

These examples should give the reader an idea of how the interactions between components with different functions will happen by using the proposed SLiM middleware.

6 Test implementation

The described SLiM middleware has been implemented and tested on a mobile device using the Android platform [1] which uses a message-based integration mechanism.

The implemented application asks the user the name of a city and then searches a predefined list of hotels. By showing the user a few options, and by integrating a simple clustering algorithm, the application is able to learn progressively which part of the city the user is interested in.

We had to integrate components already existing in the Android platform such as the text-to-speech engine, speech recognition, internet browsing and map view. For each of them we have created a wrapper that exposed the capabilities of each component through appropriate patterns such as {`show on map ?`} which looked for a link {`? address`} (where `?` is the same in both patterns) and then showed the address using the available map view.

The application behaved correctly and the components integration was very smooth.

7 Comments and limitations

We believe that integration of many of components is the key to making software agents smarter and make humans think of them as autonomous agents with which they can interact. Different works in AI and other connected domains are situated at different levels of abstraction and an integration middleware has to be able to deal with it. Also, the data that can be shared between different components is very diverse and an integration middleware would have to use a data model capable of handling this diversity. We believe the SLiM middleware, through the use of a very expressive data model and a flexible pattern matching mechanism, is a first step towards that.

One important limitation of the SLiM middleware comes from the fact that each created link or symbol has to be processed by the triggers index. This basically creates a bottleneck which means that slim updates cannot be performed at very high rates (on the android platform the maximum rate, as tested, is at about 1200 updates per second). This makes it not suitable for components that need interactions between them at a high rate. In that case, additional middlewares capable of handling such interactions should be used in the trunk together with the SLiM middleware. Alternatively, pattern checking can be performed in parallel on multiple cores which would also give better performance.

One interesting aspect of SLiM which has not been mentioned earlier is the fact that the SLiM language can be used as a scripting language for the interaction of components. We can have a default module which implements patterns for the usual control constructs we find in a scripting language (i.e. {`if Cond:? then Action: ?`}, {`for X:? in List:? do Action: ?`}, etc.). Together with the patterns registered by different components we will actually end up with a kind of domain specific language whose primitives are dictated by the capabilities of the agent.

Having a symbolic middleware like SLiM implemented in multiple agents, even developed by third parties, would create a more solid base for an ACL (Agent Communication Language). We can have the constructs of the language translated into symbolic representations which would then be executed by an agent using the registered components at a given time.

As it can be seen, having an explicit symbolic middleware in a software agent has many advantages and it opens very interesting perspectives.

8 Conclusions and future works

This paper proposes the integration of multiple components in a software agent through the use of a symbolic middleware based on the new SLiM model. Due to the expressivity of hyper-graphs and the flexibility of the proposed pattern matching mechanism this model is well suited for the integration of AI with non-AI components.

We provided a clear formal description of the SLiM model, a representation language that can be used to describe slims and a pattern matching mechanism. Based on them we proposed a model of a symbolic middleware which uses a slim at its core and two types of patterns, capability and trigger patterns.

The efficiency of the pattern matching algorithm and its small memory footprint show that the SLiM integration middleware is well suited for mobile platforms such as Android on which a test implementation was done.

The proposed approach combines the advantages of multiple generic integration techniques: a) the flexibility of a shared blackboard with an expressive hyper-graph based data model; b) a triggering mechanism based on patterns which generalizes the publisher-subscribe paradigm; c) *do* mechanism based on patterns similar to method invocation; d) straightforward integration through behaviour rules based on patterns.

Also, the use of an explicit symbolic middleware such as the SLiM middleware can lead to interesting application such as a component integration scripting language, easier implementation of agent communication languages and even the creation of lightweight or distributed agents.

REFERENCES

[1] Android. http://www.android.com., 2010.

[2] OpenAIR. www.mindmakers.org, 2011.

[3] H. Boley. Directed recursive labelnode hypergraphs: A new representation-language. *Artificial Intelligence*, 9(1):49 – 85, 1977.

[4] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1-3):139–159, 1991.

[5] h. CORBA, 2011.

[6] R. Engelmore and A. Morgan, editors. *Blackboard Systems*. Addison-Wesley, 1988.

[7] L. D. Erman and V. R. Lesser. The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12:213–253, 1980.

[8] B. Goertzel. Patterns, hypergraphs and embodied general intelligence. *International Joint Conference on Neural Networks*, pages 451 – 458, 2006.

[9] B. Goertzel, H. de Garis, C. Pennachin, N. Geisweiller, S. Araujo, J. Pitt, S. Chen, R. Lian, M. Jiang, Y. Yang, and D. Huang. OpenCogBot: Achieving generally intelligent virtual agent control and humanoid robotics via cognitive synergy. *ICAI*, 2010.

[10] H. Hirukawa, F. Kanehiro, and S. Kajita. OpenHRP: Open architecture humanoid robotics platform. In R. Jarvis and A. Zelinsky, editors, *Robotics Research*, volume 6 of *Springer Tracts in Advanced Robotics*, pages 99–112. Springer Berlin / Heidelberg, 2003.

[11] H.-H. Huang, A. Cerekovic, I. Pandzic, Y. Nakano, and T. Nishida. Scripting human-agent interactions in a generic ECA framework. In *Applications and Innovations in Intelligent Systems XV*, pages 103–115. Springer London, 2008.

[12] B. Iordanov. HyperGraphDB: A generalized graph database. *First International Workshop on Graph Database*, 2010.

[13] J. Muller. The agent architecture INTERRRAP. In *The Design of Intelligent Agents*, volume 1177 of *LNCS*, pages 45–123. Springer Berlin / Heidelberg, 1996.

[14] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[15] K. Thrisson. Integrated A.I. systems. *Minds and Machines*, 17:11–25, 2007.

[16] K. R. Thrisson, T. List, C. Pennock, and J. Dipirro. Whiteboards: Scheduling blackboards for semantic routing of messages & streams. In *AAAI-05 Workshop on Modular Construction of Human-Like Intelligence*, pages 8–15, 2005.