Ioannis Hatzilygeroudis
Vasile Palade
Editors

SMART INNOVATION, SYSTEMS AND TECHNOLOGIES ■ 23

Combinations of Intelligent Methods and Applications

Proceedings of the 3rd International Workshop, CIMA 2012, Montpellier, France, August 2012

Springer

Smart Innovation, Systems and Technologies

Volume 23

Series Editors

For further volumes:
http://www.springer.com/series/8767

Ioannis Hatzilygeroudis
Vasile Palade
Editors

Combinations of Intelligent Methods and Applications

Proceedings of the 3rd International Workshop, CIMA 2012, Montpellier, France, August 2012

Springer

Editors
Ioannis Hatzilygeroudis
Department of Computer Engineering
and Informatics, School of Engineering
University of Patras
Patras
Greece

Vasile Palade
Department of Computer Science
University of Oxford
Oxford
UK

ISSN 2190-3018 ISSN 2190-3026 (electronic)
ISBN 978-3-642-36650-5 ISBN 978-3-642-36651-2 (eBook)
DOI 10.1007/978-3-642-36651-2
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013932646

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The combination of different intelligent methods is a very active research area in Artificial Intelligence (AI). The aim is to create integrated or hybrid methods that benefit from each of their components. It is generally believed that complex problems can be easily solved with such integrated or hybrid methods.

Some of the existing efforts combine what are called soft computing methods (fuzzy logic, neural networks, and genetic algorithms) either among themselves or with more traditional AI methods such as logic and rules. Another stream of efforts integrates case-based reasoning or machine learning with soft computing or traditional AI methods. Yet, another integrates agent-based approaches with logic and also nonsymbolic approaches. Some of the combinations have been quite important and more extensively used, like neuro-symbolic methods, neuro-fuzzy methods, and methods combining rule-based and case-based reasoning. However, there are other combinations that are still under investigation, such as those related to the Semantic Web. In some cases, combinations are based on first principles, whereas in other cases they are created in the context of specific applications.

The 3rd Workshop on "Combinations of Intelligent Methods and Applications" (CIMA 2012) was intended to become a forum for exchanging experience and ideas among researchers and practitioners who are dealing with combining intelligent methods either based on first principles or in the context of specific applications.

Important issues of the Workshop were (but not limited to) the following:

- Case-based reasoning integrations
- Genetic algorithms integrations
- Combinations for the Semantic Web
- Combinations and Web intelligence
- Combinations and Web mining
- Fuzzy-evolutionary systems
- Hybrid deterministic and stochastic optimization methods
- Hybrid knowledge representation approaches/systems
- Hybrid and distributed ontologies
- Information fusion techniques for hybrid intelligent systems
- Integrations of neural networks

- Intelligent agents Integrations
- Machine learning combinations
- Neuro-fuzzy approaches/systems
- Applications of combinations of intelligent methods to
  - Biology and bioinformatics
  - Education and distance learning
  - Medicine and health care

CIMA 2012 was held in conjunction with the 22nd European Conference on Artificial Intelligence (ECAI 2012).

This volume includes revised versions of the papers presented in CIMA 2012.

We would like to express our appreciation to all authors of submitted papers as well as to the members of CIMA-12 program committee for their excellent work. We would also like to thank the ECAI 2012 Workshop Chairs for accepting to host CIMA 2012.

We hope that this proceedings will be useful to both researchers and developers. Given the success of the first three Workshops on combinations of intelligent methods, we intend to continue our effort in the coming years.

Ioannis Hatzilygeroudis
Vasile Palade

Workshop Organization

Chairs-Organizers
Ioannis Hatzilygeroudis, University of Patras, Greece
Vasile Palade, University of Oxford, UK

Program Committee
Salha Alzahrani, Taif University, Saudi Arabia
Plamen Agelov, Lancaster University, UK
Soumya Banerjee, Birla Institute of Technology, India
Nick Bassiliades, Aristotle University of Thessaloniki, Greece
Kit Yan Chan, Curtin University of Technology, Australia
Artur S. d'Avila Garcez, City University, UK
Constantinos Koutsojannis, TEI of Patras, Greece
George Magoulas, Birkbeck College, University of London, UK
Toni Moreno, University Rovira i Virgili, Spain
Ciprian-Daniel Neagu, University of Bradford, UK
Jim Prentzas, Democritus University of Thrace, Greece
Roozbeh Razavi-Far, Universite Libre de Bruxelles, Belgium
Han Reichgelt, Southern Polytechnic State University, GA, USA
Jun Sun, Jiangnan University, China
George Tsihrintzis, University of Piraeus, Greece

Contact Chair
Ioannis Hatzilygeroudis
Department of Computer Engineering and Informatics
University of Patras, Greece
Email: ihatz@ceid.upatras.gr

Contents

Intelligent Agents: Integrating Multiple Components Through a Symbolic Structure

Razvan Dinu, Tiberiu Stratulat and Jacques Ferber

Abstract In order to handle complex situations, autonomous software agents need 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 proposes the use of a symbolic middleware to handle inter-component interactions. A generalized hyper-graph model is defined, a simple and straightforward representation language is proposed and a pattern matching mechanism is introduced together with a basic performance evaluation. Finally, the paper shows how a flexible symbolic middleware can be built and a few examples are presented.

Keywords multi-agent system · multiple component integration · symbolic middleware

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

R. Dinu (✉) · T. Stratulat · J. Ferber
Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier, Montpellier, France
e-mail: dinu@lirmm.fr

T. Stratulat
e-mail: stratulat@lirmm.fr

J. Ferber
e-mail: ferber@lirmm.fr

I. Hatzilygeroudis and V. Palade (eds.), *Combinations of Intelligent Methods and Applications*, Smart Innovation, Systems and Technologies 23,
DOI: 10.1007/978-3-642-36651-2_1,

"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" [16].

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 [11].

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 [17] 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 [12].

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. 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) [15]. 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 [14] 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 Sects. 3–5 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, Sect. 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 [13]. 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 $t = \{(0, \emptyset), (1, s_1), \ldots, (k, s_k)\}$ we use the equivalent notation $t = (s_1, s_2, \ldots, s_k)$. 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) $l : S \rightarrow T_S$ be a function called a *linking* function on S.
(iii) $i : S \rightarrow I$ be a function called an *information* function on S, where I is a set of elements.
(iv) $m : S' \rightarrow S$, 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 → symbol+                          (1)
symbol → [id|link][:[info|symbol]]?       (2)
  link → [id=]?{symbol+}                  (3)
    id → s ∈ S                            (4)
  info → x ∈ 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 → id[...]?     symbol.s = id.s
symbol → link[...]?   symbol.s = link.s
  link → id={symbol+} link.s = id.s
  link → {symbol+}    link.s = use/new
    id → s ∈ 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$_n$}	$l(use/new) = (s_1.s, ..., s_n.s)$
id={s$_1$ ... s$_n$}	$l(id.s) = (s_1.s, ..., s_n.s)$
[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 Sect. 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:

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

Fig. 1 "Examples of patterns"

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