A REDUCED INSTRUCTION SET MACHINE FOR KNOWLEDGE BASED SYSTEMS

Diego Stéfano F. Ferreira*, Augusto Loureiro da Costa*

*Robotics Laboratory, Federal University of Bahia, Salvador, Bahia, Brazil

Emails: diego.stefano@gmail.com, augusto.loureiro@ufba.br

Abstract— This work proposes a system level design of a special purpose RISC architecture for the implementation of Knowledge Based Systems (KBS). The processor implements an Instruction Set Architecture (ISA) derived from the description of the Rete algorithm given in Forgy (1982) - the paper that proposes the algorithm - alongside some general purpose instructions (not described here). It is part of a System-on-a-Chip (SoC) for the execution of cognitive agents. The computational architecture of this SoC will be presented using the cognitive model of the Concurrent Autonomous Agent (CAA) as a reference, since it has been successfully applied in numerous intelligent robotics applications. The cognitive model of the CAA comprises three levels that run concurrently: the reactive level, which executes the perception-action cycle, the instinctive level, that manages plan execution, and the cognitive level, which performs planning. In the method here proposed, the KBS knowledge base is compiled into a program composed by the application specific instructions and the matching procedure is executed by the processor using this program. To validate the architecture of this processor, some experiments using an example knowledge base will then be presented. The experiments - performed using a system-level simulator written in the Scala language - shown that the simulated architecture reported the expected matches for additions and exclusions of facts from the Rete memories.

Keywords— Knowledge Based Systems, Expert Systems, Reduced Instruction Set Computer, Autonomous Agents.

Resumo— Este trabalho propõe um projeto em nível de sistema de uma arquitetura RISC de propósito especial para a implementação de Sistemas Baseados em Conhecimento (SBC). O processador implementa uma Arquitetura de Conjunto de Instruções (ISA, do inglês) derivada da descrição do algoritmo Rete dada em Forgy (1982) - o artigo que propõe o algoritmo - junto com algumas instruções de propósito geral (não descritas aqui). A arquitetura é parte de um System-on-a-Chip (SoC) dedicado à execução de agentes cognitivos. A arquitetura computacional desse SoC será apresentada utilizando o modelo cognitivo do Agente Autônomo Concorrente (AAC) como uma referência, dado que este modelo foi aplicado com sucesso em aplicações de robótica inteligente. O modelo cognitivo o AAC contém três níveis que executam concorrentemente: o nível reativo, que executa o ciclo percepção-ação, o nível instintivo, que trata da execução de planos, e o nível cognitivo, que executa o planejamento. No método proposto aqui, a base de conhecimento do SBC é compilada em um programa composto pelo ISA de aplicação específica e o processo de correspondência de padrões é executado pelo processador projetado utilizando este programa. Para validar a arquitetura desse processador, alguns experimentos utilizando uma base de conhecimento de exemplo serão apresentados. Os experimentos - realizados utilizando um simulador em nível de sistema escrito na linguagem de programação Scala - motram que a arquitetura simulada acusou as correspondências corretas para adições e exclusões de fatos.

Palavras-chave— Sistemas Baseados em Conhecimento, Sistemas Especialistas, Computador com Conjunto de Instruções Reduzido, Agentes Autônomos.

1 Introduction

An agent is any entity that can sense the environment and act upon it (Russell and Norvig, 2010). Additionally, according to (Huhns and Singh, 1998), knowledge about the environment can be used by the agent to relate perceptions and actions properly. In this case the agent is a cognitive agent. Cognition, according to (Vernon et al., 2007), is a process that allows a system to robustly behave adaptively and autonomously, with anticipatory capabilities. The authors proceed by classifying cognitive systems in two broad classes, namely the cognitivist and the emergent systems. Inside the cognitivist class goes systems that relies on symbolic representation and information processing. In the second class, the emergent systems, are connectionist, dynamical and enactive systems.

There are aspects of cognitive agents that remain invariant in time and over different tasks. These aspects generally include the short-term and long-term memories where knowledge is stored, the knowledge representation structure and the processes that performs over the previous elements (possibly changing its contents, like learning). The aspects cited above are comprised by a cognitive architecture (Langley et al., 2009).

An example of an architecture for cognitive agents is the Concurrent Autonomous Agent (CAA), an autonomous agent architecture for mobile robots that has already proven to be very powerful (Costa and Bittencourt, 1999; Costa and Bittencourt, 2001; Cerqueira et al., 2013). The architecture has three levels, namely, the reactive level, responsible for performing the perception-action cycle, the instinctive level, which governs the selection of predefined behaviors in the reactive level, and the cognitive level, which deals with planning. The later two uses a Knowledge Based System (KBS) as reasoning mechanism. These levels runs concurrently, which makes parallelism and fast inference cycles quintessential properties of any real-time application of the CAA.

(Ferreira et al., 2014) embedded the CAA in a microcontrollers network specially designed to fit its cognitive architecture. The intention of the authors was to optimize the performance of the agent for an embedded environment, allowing it to be physically embedded into a mobile robot. In this work, a step forward is given in that direction, since its main objective is to design a processor for the execution of a KBS which, in turn, will be part of a System-on-a-Chip (SoC) dedicated to the execution of cognitive agents, using the CAA as reference model.

Hardware design for cognitivist systems (using (Vernon et al., 2007) aforementioned classification) is not a recent concern. The first paper about the *Rete* matching algorithm (Forgy, 1982) already presents a low-level (assembly) implementation of the matching algorithm for production systems. Years later, (Lehr, 1985) designed a Reduced Instruction Set Computer (RISC) machine for OPS production systems, focusing on optimizing the branch prediction unit for the task. A more recent approach by (Peters et al., 2014) proposes a parallelization strategy to use the parallel processing power of Graphics Processing Units (GPUs) for *Rete* pattern matching.

In this work, an application specific Instruction Set Architecture (ISA) based on the implementation provided in (Forgy, 1982) is provided, defining a RISC processor architecture dedicated to the execution of the *Rete* matching algorithm. This is part of the more complex cognitive SoC mentioned above, and since the later will use the CAA as reference, which applies a *Rete*-based KBS on both its symbolic levels, the processor here presented will play a central role in its design.

This work is divided as follows. In Section 2, the CAA is presented, with its levels explained. Section 3 then exposes how knowledge is represented and inference is performed by the CAA KBS (in both instinctive and cognitive levels). The Section 5 describes the system proposed. Section 6 shows the results of simulations and some conclusions are presented in Section 7.

2 The Concurrent Autonomous Agent (CAA)

The architecture of the Concurrent Autonomous Agent (CAA) was inspired by the generic model for cognitive agents. This model comprises three levels: the reactive level, the instinctive level and the cognitive level. The CAA levels are shown in Figure 1 (Costa and Bittencourt, 1999; Costa and Bittencourt, 2001). In this figure, the reactive level, responsible for the real-time response of the agent, contains a set of reactive behaviours that are activated in specific situations. Only one behaviour may be active at a time. This level executes the perception-action cycles. The instinctive level, in turn, changes, after each perception-action cycle, the state of the world that it maintains in a KBS. It also updates the symbolic information used by the cognitive level and coordinates the selection of reactive behaviours. This level executes a plan (stored in the knowledge base of its KBS) to achieve the local goals and, when a goal is reached, a message is sent to the cognitive level signaling about it. Finally, the cognitive level stores a logical model of the world, creates global and local goals also using a KBS, and sends the local goals to the instinctive level (Costa and Bittencourt, 2001).

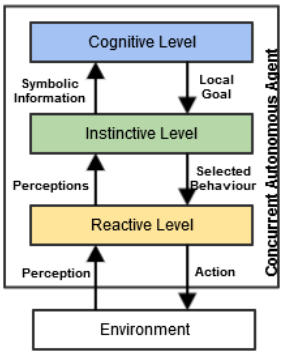

Figure 1: Cocurrent Autonomous Agent Architecture.

The level that interacts with the environment executing a fast-perception-action cycle is the reactive level. It consist of a collection of reactive behaviors that determines the interaction of the agent with the environment. Only one behavior can be active at a time, and the instinctive level makes the selection. The architecture used in (Ferreira et al., 2013; Ferreira et al., 2014) consists of a kinematic position controller for the omnidirectional robot *AxéBot*. The reactive behaviours were implemented based on the embedded kinematic controller. The behaviors implemented were simple: there is one behaviour for each cardinal direction, i. e., selecting the behavior corresponds to selecting the direction (relative to the orientation of the robot) in which one wishes the robot to move onto.

The instinctive level, as the reactive level, is identical to the one purposed in (Ferreira et al., 2014): its reasoning mechanism consists of a KBS that executes a plan generated by the cognitive level, sending symbolic information about the environment to the latter. The plans are executed by coordinating behavior selection in the reactive level, which sends the perceptions to this level.

The cognitive level also uses a KBS as automatic reasoning method. Its facts base consists of a logical model of the world. The inference engine is multi-cycle, meaning that it keeps running independent of the update of the facts base by the

instinctive level. This level does the planning, coordinating the instinctive level for the execution of the plans. It is not used in this work.

3 Knowledge Based Systems (KBS)

In the instinctive and cognitive levels, as mentioned before, a KBS is used as automatic reasoning mechanism. The KBS comprises a facts base, a rules base and an inference engine, as shown in Figure 2 (Costa and Bittencourt, 2001).

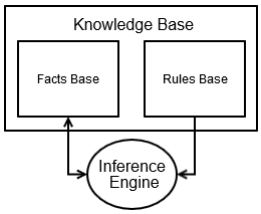

Figure 2: KBS used by AAC.

The facts base contains information known by the agent about its state and about the state of the environment. This information is stored in the format *logic*(*object attribute value*), where *object* refers to some element of the world, *attribute* is the attribute of this object and *value* is the value of this attribute.

This format is used also to form the premises of the rules, but in this case they are called specifications. The values of the attributes presents restriction that must be met by the facts in order to fire the rule. To help in the expression of such restriction, the language allows the use of filters and variables in the premise of a rule. A variable is simply a symbol preceded by a interrogation (?) token (e. g. ?x, ?*symbol*). And the filters have the format $filter$(*operator parameter*1 *parameter*2), where the *operator* field tells how to compare *parameter*1 against *parameter*2, and these last two may be variables or numbers.

The inference engine then executes a forward chaining algorithm, checking if the left side (or premise) of a rule is satisfied by the facts base, and if it does, right side of the rule (or consequent) is executed. The consequents may consist of a new fact being added to the facts base, an existing fact being updated or a message being sent to another level.

4 The *Rete* Algortihm

The *Rete* algorithm is used in the inference engine of a KBS to efficiently match rules premises with the facts without the need of looping through both rules and facts bases at each inference cycle, greatly improving performance. It was proposed by Charles Forgy in 1979, in its doctoral thesis (Forgy, 1979). Its name, *rete*, is latin for *network*, because of how it organizes the information in the knowledge base.

The *Rete* algorithm starts by constructing a network of nodes and memories based on the premises of the rules and eventual filters they may contain. This network is divided in two parts: the alpha and the beta networks.

The alpha network is composed by the following nodes:

- a *Root Node*, which is the entry point for new facts;
- *Constant Test Nodes* (CTN), which checks whether the non-variable (constant) fields of premises matches the corresponding ones in the current fact; and
- *Alpha Memories* (AM), that stores facts that successfully passed through *constant test nodes*.

The beta network, in turn, have:

- *Join Nodes* (JN), where a set of test are performed to check variable binding consistency;
- *Beta Memories* (BM), which conjunctively "accumulates" facts that passed the corresponding JN tests in *tokens*, which are partial matches to specific premises; and
- *Production Nodes*, which are terminal nodes for full matches.

5 The RISC *Rete* Machine

In this section, the proposed processor system level architecture is described. As it was stated in the introduction, this processor is part of the design of a SoC for cognitive agents. The idea is to have this processor working with another unit for planning in the cognitive level and with a plan execution unit in the instinctive level. The knowledge bases should first be compiled into the application specific ISA of the *Rete* processor and then downloaded into its instruction memory

The system level architecture here presented is a RISC Application Specific Processor (ASP) whose special purpose ISA was inspired on the description of the *Rete* algorithm given in (Forgy, 1982), the first paper written about the algorithm. The author uses an assembly-like set of instructions to describe the operation of the algorithm. But they serve only as guidelines for a high-level implementation described afterwards in that paper; no hardware implementations are presented.

Inspired by the aforementioned (pseudo-)instructions presented in (Forgy, 1982), this work purposes an actual machine for the execution of the *Rete* matching algorithm whose ISA implements a modified version of the pseudo-instructions presented in Forgy's seminal paper.

The overall processor architecture is shown in Figure 3. The alpha and beta memories are pre-allocated at compiling time.

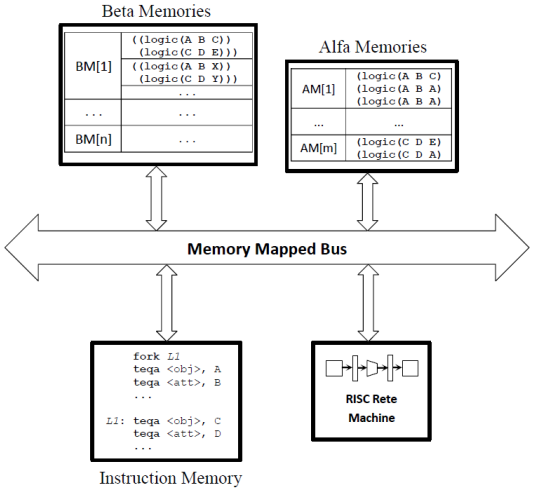

Figure 3: Architecture of the *Rete* processor

The rules are compiled in a sequence of these instructions instead of being used for the creation of the *Rete* tree in memory. The alpha and beta memories will still exist but the nodes (constant test and join nodes) will be implemented by instructions (Figure 3).

The new fact is stored in a register together with a bit indicating whether it is being added or deleted. The instructions arguments and operation are detailed below:

- `FORK label`: Represents a branch in the tree, with one of the nodes represented by a node instruction immediately after it and the other at the instruction address given by `label`. This address is stacked and the next instruction is fetched. `label` is popped from stack when a mismatch occurs and the program jumps to it. If the stack is empty, the match failed.

- `TEQA field, constant`: Implements a CTN: `field` is the field to be tested (object, attribute or value) of the fact register and `constant` is the value this field must equal.

- `FILTER field1, field2, comparison`: Compares two fields inside a fact using a given comparison operation. If the comparison fails, so does the match.

- `MERGE parent_bm, bm, am, next_join`: Saves in registers the addresses of its parent memories and of the next `MERGE` instruction. Also, it updates an alpha memory in a right activation.

- `TEST field1, premise, field2, comparison`: Deals with left and right activations of the JN. It triggers an interruption, jumping to a pre-programmed routine that runs through the memories testing whether or not the the `field1` compares to `field2` of `premise`-th premise. The comparison operation is given in the field `comparison`.

- `JOIN lbl`: jumps to a JN (`MERGE` instruction) defined previously in the code, on a right activation.

- `TERM rule, nsubs`: Represents production nodes. It saves the address of the consequence of the matched rule in a register, for further use. `nsubs` is the number of substitutions this rule has, so that it can jump to the last one (for popping the test stack) in the case where the current fact is to be excluded instead of added.

- `SUBST p1, f1, p2, f2, lst`: Uses the matched token to create substitutions for the variable in the consequence. The pairs `(p1, f1)` and `(p2, f2)` are "coordinates" of occurrences of the same variable in the premises and the consequence, respectively. `lst` indicates whether it is the last substitution for that match or not. If it is, the test stack must be popped to proceed with interrupted tests.

6 Case Study

The architecture were simulated using a program written in the Scala programming language, using array structures for the instruction, alpha and beta memories, lists for fork and JN test stacks and variables for registers (program counter, direction flags, stacks counters, alpha and beta indices etc.). The dynamics of the program was dictated by the way the instructions changed the program counter.

6.1 Domain Definition

The architecture will be validated using the block world domain example, from (Russell and Norvig, 2010). The Figure 4 shows the knowledge base that is going to be used in the simulation. The same filter shown in the *Move* rule could be present in the *MoveToTable* rule, but it was omitted for simplicity. In spite of the fact that the consequences won't be used here (only the matching procedure is important), one should notice the *add* and the *rem* symbols. Those are simply instructions on how to modify the facts base in case of a match.

6.2 Simulated Tests

The knowledge base shown in Figure 4 is then compiled into the program presented in Figure 5. This program is the input for the simulator, in

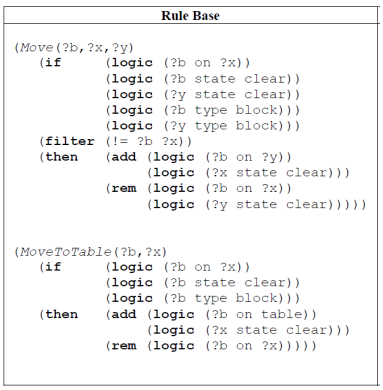

Figure 4: Knowledge base for the block world domain example.

the form of an array of instruction objects (objects from an instruction class defined inside the simulator code). It then waits for a new fact to come in and then executes the program once for that fact, changing the memories accordingly.

```
        fork    ctn_state_clear
        teqa    attribute, "on"
        filter  object, value, "!="
        merge   dummy_bm, am1, join2
join2:  merge   bm1, am2, jfrk
        test    obj, 0, obj, "=="
jfrk:   fork    join6
join3:  merge   bm2, am2, join4
        test    obj, 1, obj, "!="
join4:  merge   bm3, am_type_block, join5
        test    obj, 0, obj, "=="
join5:  merge   bm4, am_type_block, join6
        jtest   obj, 2, obj, "=="
        term    Move, 6
        subst   (0, object), (0, object), last=false
        subst   (0, object), (2, object), last=false
        subst   (0, value), (1, object), last=false
        subst   (0, value), (2, value), last=false
        subst   (2, object), (0, value), last=false
        subst   (2, object), (3, object), last=true
join6:  merge   bm2, am_type_block, NULL
        jtest   obj, 0, obj, "=="
        term    MoveToTable, 4
        subst   (0, object), (0, object), last=false
        subst   (0, object), (2, object), last=false
        subst   (0, value), (1, object), last=false
        subst   (0, value), (2, value), last=true
ctn_state_clear:
        fork    ctn_type_block
        teqa    attribute, "state"
        teqa    value, "clear"
        fork    join23
        join    join2
join23: join    join3
ctn_type_block:
        teqa    attribute, "type"
        teqa    value, "block"
        fork    join456
        join    join4
join456:
        fork    join56
        join    join5
join56: join    join6
```

Figure 5: Code for the *Rete* network of the block world example.

The tests performed consisted of feeding some facts that are known to cause a match, one by one, to the simulator and check whether the system detects that match. The facts $(logic(C, on, A))$, $(logic(C, type, block))$ and $(logic(C, state, clear))$ must match the premise of the rule $MoveToTable$.

The Figure 6 shows the output of the simulator after feeding it with the fact $(logic(C, on, A))$. This output contains all the instruction executed by the processor for the given fact. As the entry point of the the network is the root node and it has three CTNs as children, a `FORK` is always processed first. For the current fact, the forked address is only taken after the fact is stored in $AM1$ (and consequently in $BM1$ too, since the parent BM is a dummy one), when the JN test fails ($pc = 5$) due to the absence of facts in $AM2$. It is noteworthy that the instruction at forked address is another fork, because the root node has three children. In the last CTN ($pc = 33$) the fork stack is empty, and as the test failed, the program finishes.

```
================================
(+)(C,on,A):
================================
pc=0>   fork    27
pc=1>   teqa    attr, on
pc=2>   filter  obj, val, "!="
pc=3>   merge   dummy, bm1, am0, 4
pc=4>   merge   bm1, bm2, am1, 6
pc=5>   jtest   obj, 0, obj, "==" // TEST FAILED: FORKING
pc=27>  fork    33
pc=28>  teqa    attr, state // TEST FAILED: FORKING
pc=33>  teqa    attr, type // TEST FAILED
```

Figure 6: Output of the simulator for $(logic(C, on, A))$.

For $(logic(C, type, block))$ the processing mechanism is the same, but the execution path is different, since the fact is going to a different alpha memory.

When $(logic(C, state, clear))$ is added, a match occurs ($pc = 22$), as can be seen in Figure 7. In this output it is possible to see the JN tests being stacked once a partial match is found ($pc = 5$). After that ($pc = 8$), a test fails, but the previously stacked test is not popped yet: there is a `FORK` at $pc = 6$, and forks have priority. Also, in this execution four substitutions take place ($pc = 23$ to 26).

```
================================
(+)(C,state,clear):
================================
pc=0>   fork    27
pc=1>   teqa    attr, on // TEST FAILED: FORKING
pc=27>  fork    33
pc=28>  teqa    attr, state
pc=29>  teqa    val, clear
pc=30>  fork    32
pc=31>  join    4
pc=4>   merge   bm1, bm2, am1, 6
pc=5>   jtest   obj, 0, obj, "==" // R.A. OK: PUSH TEST
pc=6>   fork    20
pc=7>   merge   bm2, bm3, am1, 9
pc=8>   jtest   obj, 1, obj, "!=" // TEST FAILED: FORKING
pc=20>  merge   bm2, bm6, am2, -1
pc=21>  jtest   obj, 0, obj, "==" // L.A. OK: PUSH TEST
pc=22>  term    MoveToTable, 4 // ** (+) MATCH **
pc=23>  subst   (0,0), (0,0), last=false
pc=24>  subst   (0,0), (2,0), last=false
pc=25>  subst   (0,2), (1,0), last=false
pc=26>  subst   (0,2), (2,2), last=true // POP TEST
pc=21>  jtest   obj, 0, obj, "==" // TEST FAILED: FORKING
pc=32>  join    7
pc=7>   merge   bm2, bm3, am1, 9
pc=8>   jtest   obj, 1, obj, "!=" // TEST FAILED: FORKING
pc=33>  teqa    attr, type // TEST FAILED: UNSTACKING TEST
pc=5>   jtest   obj, 0, obj, "==" // TEST FAILED
```

Figure 7: Output of the simulator for $(logic(C, state, clear))$.

Finally, Figure 8 shows the output of the simulator for the exclusion of the fact $(logic(C, on, A))$. The procedure for exclusion

starts as the same as the one for addition: the instructions guide the traverse of the tree looking for a match. The difference is that no changes are made to the memories. Instead, for every activation or match caused by the input fact, the index of the corresponding memory and its position inside that memory are stacked. At the end of the execution, when there are no more forks or tests stacked, the exclusion stack is pop and the elements of the memories given by the indices stored in it are deleted.

```
================================
(-)(C,on,A):
================================
pc=0>   fork    27
pc=1>   teqa    attr, on
pc=2>   filter  obj, val, "!= "
pc=3>   merge   dummy, bm1, am0, 4
pc=4>   merge   bm1, bm2, am1, 6
pc=5>   jtest   obj, 0, obj, "==" // L.A. OK: PUSH TEST
pc=6>   fork    20
pc=7>   merge   bm2, bm3, am1, 9
pc=8>   jtest   obj, 1, obj, "!=" // TEST FAILED: FORKING
pc=20>  merge   bm2, bm6, am2, -1
pc=21>  jtest   obj, 0, obj, "==" // L.A. OK: PUSH TEST
pc=22>  term    MoveToTable, 4 // ** (-) MATCH **
pc=26>  subst   (0,2), (2,2), last=true// UNSTACKING TEST
pc=21>  jtest   obj, 0, obj, "==" // TEST FAILED: FORKING
pc=27>  fork    33
pc=28>  teqa    attr, state // TEST FAILED: FORKING
pc=33>  teqa    attr, type // TEST FAILED: UNSTACKING TEST
pc=5>   jtest   obj, 0, obj, "==" // TEST FAILED
```

Figure 8: Output of the simulator for deleting $(logic(C, on, A))$.

7 Final Considerations

In this paper the system level design of an application specific RISC machine for the execution of the Rete algorithm was proposed. It consists of a special purpose ISA based on the algorithm proposal (Forgy, 1982). The architecture was presented and simulated using a simple problem domain: the block world domain. After running the program written for the processor on different facts, for addition and exclusion, the expected matches were successfully detected.

The proposed architecture allows for low-level symbolic processing, which can give embedded and on-chip systems fast automatic reasoning capabilities, with low power consumption.

As future works, the processor architecture here presented should be included in the design of the a SoC for the execution of cognitive agents (which is actually the purpose of this design). Also, some more complex tests are being applied to it, which will help ensure its correctness and efficiency.

References

Cerqueira, R. G., Costa, A. L. d., McGill, S. G., Lee, D. and Pappas, G. (2013). From reactive to cognitive agents: Extending reinforcement learning to generate symbolic knowledge bases, *SBAI Simpósio Brasileiro de Automação Inteligente 2013*.

Costa, A. L. d. and Bittencourt, G. (1999). From a concurrent architecture to a concurrent autonomous agents architecture, *Lecture Notes in Artificial Inteligence* **1856**: 85–90.

Costa, A. L. d. and Bittencourt, G. (2001). Hybrid cognitive model, *The Third International Conference on Cognitive Science ICCS'2001 Workshop on Cognitive Angents and Agent Interaction*.

Ferreira, D. S. F., Paim, C. C. and da Costa, A. L. (2013). Using a real-time operational system to embed a kinematic controller in an omnidirectional mobile robot, *XI SBAI Simpósio Brasileiro de Automação Inteligente*.

Ferreira, D. S. F., Silva, R. R. d. and Costa, A. L. d. (2014). Embedding a concurrent autonomous agent in a microcontrollers network, *Biosignals and Biorobotics Conference (2014): Biosignals and Robotics for Better and Safer Living (BRC), 5th ISSNIP-IEEE*, IEEE, pp. 1–6.

Forgy, C. L. (1979). *On the efficient implementation of production systems*, PhD thesis, Carnegie-Mellon University.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem, *Artificial intelligence* **19**(1): 17–37.

Huhns, M. N. and Singh, M. P. (1998). Cognitive agents, *Internet Computing, IEEE* **2**(6): 87–89.

Langley, P., Laird, J. E. and Rogers, S. (2009). Cognitive architectures: Research issues and challenges, *Cognitive Systems Research* **10**(2): 141–160.

Lehr, T. F. (1985). The implementation of a production system machine. Technical Report CMU-CS-85-126.

Peters, M., Brink, C., Sachweh, S. and Zündorf, A. (2014). Scaling parallel rule-based reasoning, *The Semantic Web: Trends and Challenges*, Springer, pp. 270–285.

Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence*, Pearson Education.

Vernon, D., Metta, G. and Sandini, G. (2007). A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents, *IEEE Transactions on Evolutionary Computation* **11**(2): 151.