Chapter 10

IntelliSoC: A System Level Design and Conception of a System-on-a-Chip (SoC) to Cognitive Agents Architecture

Diego Ferreira, Augusto Loureiro da Costa and
Wagner Luiz Alves De Oliveira

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.79265

Abstract

This chapter presents a system level design and conception 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. This cognitive model comprises three levels that run concurrently, namely the reactive level, the instinctive level and the cognitive level. The reactive level executes a fast perception-action cycle. The instinctive level receives perceptions from and sends the active behavior to the reactive level, and using a Knowledge Based System (KBS) executes plans by selecting reactive behaviors. The cognitive level receives symbolic information from the instinctive level to update its logical world model, used for planning and sends new local goals to instinctive level. Thus, this work proposes a novel SoC whose architecture fits the computational demands of the aforementioned cognitive model, allowing for fast, energy-efficient, embedded intelligent applications.

Keywords: cognitive agents, intelligent robots, mobile robots

1. Introduction

Every entity that can perceive its environment and perform actions upon it can be called an *agent* [1]. When this process is achieved using knowledge about the environment, then the agent is a *cognitive agent* [2]. Cognition, according to [3], is a process that allows to 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 [4].

An example of 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 [5–7]. Three parallel levels compose this architecture: the reactive, the instinctive and the cognitive levels. The first is responsible for executing the perception-action cycle, the second uses a knowledge based system (KBS) to select behaviors in the reactive level, and the third also uses a KBS, but for planning.

In [8] the CAA was embedded 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 System-on-a-Chip (SoC) dedicated to the execution of the CAA.

Hardware design for cognitivist systems (using [3] aforementioned classification) is not a recent concern. The first paper about the Rete matching algorithm [9] already presents a low-level (assembly) implementation of the matching algorithm for production systems. Years later, [10] 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 [11] proposes a parallelization strategy to use the parallel processing power of graphics processing units (GPUs) for Rete pattern matching.

Still searching for dedicated hardware for the *Rete* algorithm, [12] present a special purpose machine that is connected to a host computer as a support for agents execution in software. The authors also shown that increasing the number of inputs of the beta nodes in the network generated by the algorithm enhances the performance of the proposed hardware: using four or five inputs in a join node of the beta network allowed for a 15–20% increase in performance. For the operation of the processor the network is compiled into an inner representation and loaded into an external RAM. A Token Queue Unit stores partial matches, controlled by a microprogram of the Control Unit and by a Match Unit.

The authors in [13] propose a processor for the execution of production systems aiming applications of real-time embedded general artificial intelligence. The production system executed by the processor is the *Street* system. The processor, named *Street Engine*, interprets RISC instructions from its ISA, which is composed by only one type of instructions: a form of parallel production rules. Hence, instead of being executed sequentially, the instructions are executed in parallel by hardware units named producers. The producers are interconnected in a network-on-a-chip (NoC), and each one has only one associated production. The knowledge

http://dx.doi.org/10.5772/intechopen.79265

representation language employed by the Street Engine is the *Street Language*, designed to map a variant of the *Rete* algorithm to the hardware. The street engine is controlled by events, where each producer stores a subset of the working memory (corresponding to the alpha memories in the *Rete* algorithm) and changes in the working memory of one producer causes changes in other memories of producers affected by the change.

Deviating from production systems, a processor oriented to the real-time decision making of mobile robots is proposed by [14]. The processor comprises four search processors with 8 threads each for trajectory planning and a reinforcement learning acceleration module for obstacle avoidance. The search processors consist of a 6-stage pipeline that uses a three-level cache to implement a transposition and avoid redundant computation between threads. The reinforcement learning accelerator, in turn, uses a 2D array of processing elements to implement a SIMD architecture. A penalty is considered when a search processor tries to plan a trajectory that collides with an obstacle.

The common theme among these works is that they are concerned with expert (production) systems. According to [4], while cognitive architectures are intended to perform successfully in a broad range of domains, expert systems have a narrower range of applications. The authors then continues by saying that cognitive architectures "offers accounts of intelligent behaviors at the system level, rather than at the level of component methods designed for specialized tasks". Therefore, the design of a SoC for cognitive systems can accomplish the optimized performance of a dedicated low-level machine while maintaining its powerful intelligent behavior, which justifies the importance of this work.

This chapter 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). Sections 4 and 5 explains the *Rete* and *Graphplan* algorithms, laying the basis for the following sections, which describes the hardware architectures proposed to implement these algorithms. First, in Section 6, the overall architecture is described. Then, Section 7 describes the *Rete* RISC machine. And in Section 8 the cognitive module is presented. Section 9 shows the results of simulations and some final considerations are presented in Section 10.

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 [15]. The CAA levels are shown in **Figure 1** [5, 6]. As can be seen in this figure, the reactive level is responsible for interacting with the environment. It contains reactive behaviors that enables the agent to perform a fast perception-action cycle. The perceptions are sent to the instinctive level which, in turn, uses it to update the state of the world that it maintains in a KBS. It also uses this information to update the logical model of the world in the cognitive level and select the reactive behavior in the reactive level. Finally, the cognitive level does the planning, sending local goals to the instinctive level, that will coordinate the execution of the plan [6].

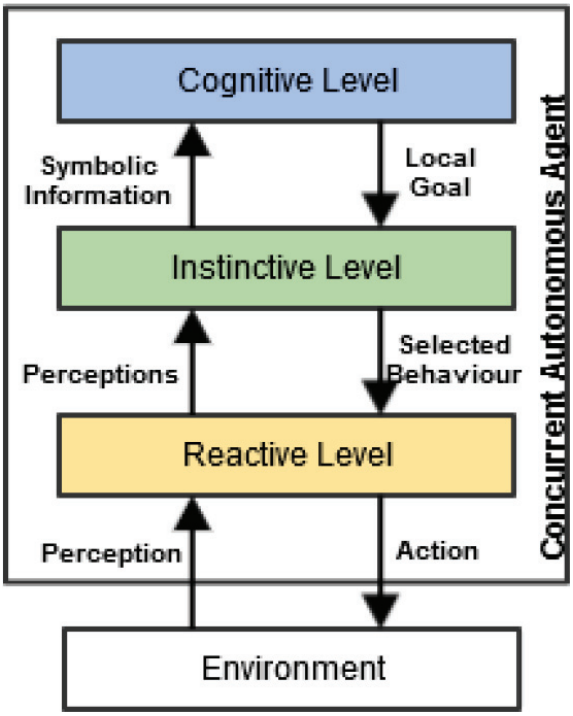

Figure 1. Concurrent autonomous agent architecture.

The level that interacts with the environment executing a fast-perception-action cycle is the reactive level. It consists 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 [8, 16] consists of a kinematic position controller for the omnidirectional robot *AxéBot*. The reactive behaviors were implemented based on the embedded kinematic controller. The behaviors implemented were simple: there is one behavior 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 [8]: 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.

http://dx.doi.org/10.5772/intechopen.79265

3. Knowledge-based systems (KBS)

The CAA uses a KBS in its two higher levels: the instinctive and the reactive. Its inner structure is shown in **Figure 2** [6].

All knowledge of the agent is stored in the *facts base*. The elements of the facts base use the format *logic*(*object attribute value*) to represent elements in the world. The facts base contains the states of the agent and of the environment in the KBS of the instinctive level and the logical model of the world in the cognitive level one.

The format above is used also to form the premises of the rules in the *rules base*. But in this case they are restrictions or specifications that the facts in the facts base must met in order to *fire* the rule and execute its consequence, which may contain instructions on how to modify its own facts base, or the facts base of another level (adding, removing or updating facts). Additionally, in the rules syntax the fields (*object*, *attribute* and *value*) of the logical elements may contain variables, which are denoted by a symbol preceded by a interrogation (?) token.

The rules may also contain filters to further specify restrictions on the values of the fields in the premise elements. Filters have the format *filter*(*operator parameter*1 *parameter*2), where *parameter*1 and *parameter*2 are variables or symbols present in some premise elements and *operator* tells how they must compare for the rule to be fired.

The inference engine uses the *Rete* matching algorithm to search the rules base for rules that are enable to fire by the facts in the facts base without having to iterate through both bases. The enabled rules then form the *conflict set*, and the inference engine must decide which rules in this sets should be fired. Finally, the inference engine executes the consequence of the fired rules and restart the process. This is the *forward chaining* algorithm.

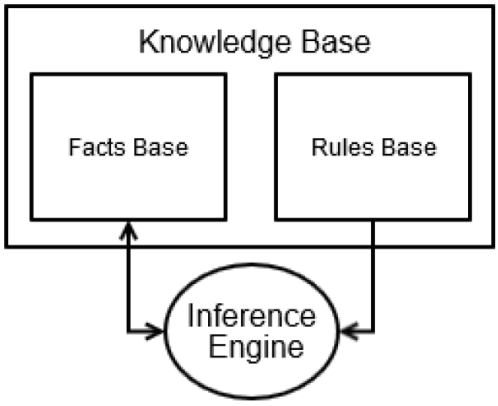

Figure 2. Block diagram of a KBS.

4. The Rete algorithm

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 [17]. 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 Graphplan algorithm

The cognitive level of the CAA contained a classical planner that used a KBS to perform a state-space search in order to generate the plan. This approach may produce an explosion in the number of states that, in turn, may overwhelm the memory capacity of the computational system that executes it. In the case of the present work, where a SoC is supposed to run the algorithm, this issue becomes specially expressive. The alternative considered here is the utilization of a planning graph algorithm search instead, which is known as *Graphplan*.

The Graphplan algorithm uses a propositional representation for states and actions, and its basic idea to reduce the search graph is to structure it as a sequence of layers comprising sets of states, or *propositions*. Following a state (proposition) layer there is an inclusive disjoint of actions, which is then followed by another propositional layer, and so on, alternately.

For a mathematical formulation of the *Graphplan* algorithm, one should first define mathematically a general planning problem. Let $\mathcal{P} = (\Sigma, s_j, g)$ be a planning problem, where:

http://dx.doi.org/10.5772/intechopen.79265

- $\Sigma = (S, A, \gamma)$ is the problem domain, with S being the set of states, A the set of actions and $\gamma = S \times A \rightarrow S$ is a state transformation application;
- $s_j \in S$ is the initial state; and
- g is the goal state.

An action $a \in A$ is composed by two sets of propositions: $prencond(a)$, its preconditions, and $effects^{(a)=effects^{+}(a)\cup effects^{-}(a)}$, its effects, where $effects^{+}(a)$ is the set of positive propositions and $effects^{-}(a)$ is the set of negative propositions in the effects of the action. For an action a to be applicable in a given state s, one should have $precond(a) \subseteq s$, and the new state would be given by $\gamma(s, a) = (s - effects^{-}(a)) \cup effects^{+}(a)$.

Now back to *Graphplan*, given the action layer A_j and the propositional layer P_{j-1} preceding it, the former contains all actions a such that $precond(a) \subseteq P_{j-1}$ and the later is composed by all propositions p such that $p \in P_{j-1}$. Also, there are three types of edges:

- connecting a proposition $p \in P_{j-1}$ to an action $a \in A_j$;
- connecting an action $a \in A_j$ to a proposition $p \in P_{j-1}$, such that $p \in effects^{+}(a)$ (positive arc); and
- connecting an action $a \in A_j$ to a proposition $p \in P_{j-1}$, such that $p \in effects^{-}(a)$ (negative arc).

Two actions $a_1, a_2 \in A_j$ are called *independent* if $effects^{-}(a_1) \cap \left(precond(a_2) \cup effects^{+}(a_2)\right) = \emptyset$ and $effects^{-}(a_2) \cap \left(precond(a_1) \cup effects^{+}(a_1)\right) = \emptyset$; otherwise they are said to be *dependent*.

If two actions are dependent, they are said to be *mutually exclusive*, or *mutex* for short. Another situation that makes two actions in the same layer to be mutex is when a precondition of one of them is mutex with a precondition of the other. And two propositions p and q are mutex if for all $a_1 \in A_j$ such that $p \in effects^{+}(a_1)$ is mutex with every $a_2 \in A_j$ such that $q \in effects^{+}(a_2)$, and $\nexists a \in A_j$ such that $p, q \in effects^{+}(a)$.

The algorithm works as follows. It first expands the planning graph, generating the direct layered graph described before, with the successive proposition and action layers. Then when a propositional layer P_j contains g, the expansion stops and the graph is searched backwards for a sequence of sets of non-mutex actions. The plan is then extracted from this sequence.

An important difference between the algorithm used here and the one from the standard approach is that here we use a KBS to represent knowledge and execute the inference. So the expansion step The pseudo-code for the expansion step is given in Algorithm 1.

Algorithm 1 Planning graph expansion

1: **procedure** EXPAND(s_i) ▷s_i: i-th state layer

2: $A_{i+1} \leftarrow KBS.InferenceCycle(s_i, A)$ ▷A: action profiles

3: $s_{i+1} \leftarrow \cup A_{i+1}.effects^{+}$

4: $\mu A_{i+1} \leftarrow \{(a,b) \in A_{i+1}^2, a \neq b \,|\, Dependent(a,b) \vee \exists (p,q) \in \mu s_i : p \in preconds(a), q \in preconds(b)\}$

5: $\mu s_{i+1} \leftarrow \{(p,q) \in s_{i+1}^2, p \neq q \,|\, \forall (a,b) \in A_{i+1}^2 : p \in effects^+(a) \wedge q \in effects^+(b) \rightarrow (a,b) \in \mu A_{i+1}\}$

6: **end procedure**

Whenever the set of WME in g - the goal state - is contained in a given state layer s_i, a recursive procedure must be executed to search for sets of non-mutex actions in each layer that could have produced all the WMEs in the goal state. This procedure is composed by the functions Search and Extract.

Algorithm 2 Search for non-mutex actions.

1: **procedure** SEARCH(g, π_i, i)
2: **if** $g = \emptyset$ **then**
3: $\Pi \leftarrow Extract(\cup\{preconds(a) \,|\, \forall a \in \pi_i\}, i-1)$
4: **if** $\Pi = Failure$ **then**
5: **return** *Failure*
6: **end if**
7: **return** $\Pi.\pi_i$
8: **else**
9: select any $p \in g$
10: $resolvers \leftarrow \{a \in A_i \,|\, p \in effects^+(a) \wedge \forall b \in \pi_i : (a,b) \notin \mu A_i\}$
11: **if** $resolvers = \emptyset$ **then**
12: **return** *Failure*
13: **end if**
14: non-deterministically choose $a \in resolvers$
15: **return** Search($g - effects^+(a)$, $\pi_i \cup \{a\}$, i)
16: **end if**
17: **end procedure**

Algorithm 3 Extract a plan.

1: **procedure** EXTRACT(g, i)
2: **if** $i = 0$ **then**

3: **return** $\emptyset$

4: **end if**

5: $\pi_i \leftarrow Search(g, \emptyset, i)$

6: **if** $\pi_i \neq Failure$ **then**

7: **return** π_i

8: **end if**

9: **return** *Failure*

10: **end procedure**

6. The proposed architecture

Figure 3 shows an overview of the SoC architecture for cognitive agents proposed in this work. As mentioned before, the cognitive model of the CAA (**Figure 1**) was used as a base model for

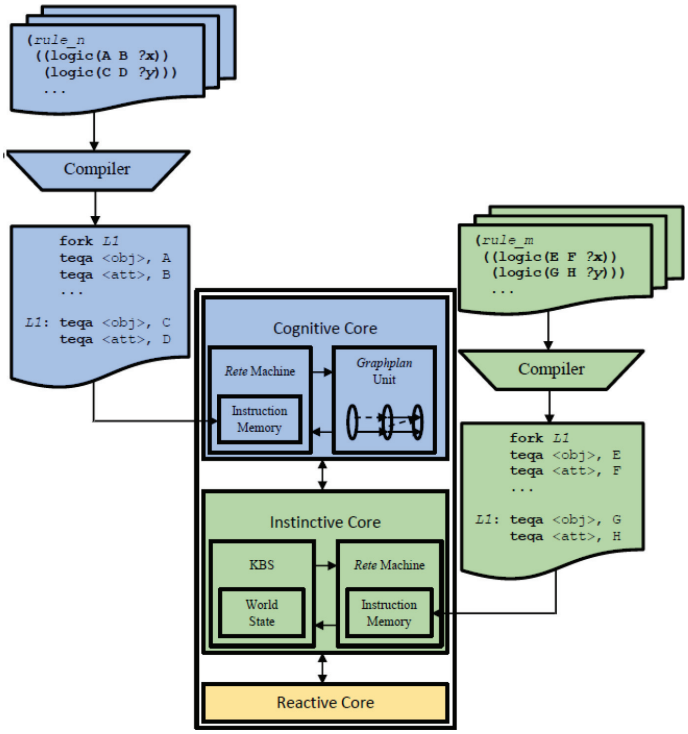

Figure 3. Block diagram of the proposed SoC.

the development of this architecture, and hence it can be seen in the image that each CAA level has a corresponding computational module in the SoC.

The modules were designed according to the task executed by the corresponding CAA level. Still referring to **Figure 3**:

- the cognitive level is implemented by a *Graphplan* module, responsible for the execution of graph planning algorithms and a module dedicated to the *Rete* algorithm. The later is used by the former for the SBC-based state space expansion;
- the instinctive level comprises only a *Rete* module that is applied to coordinate with the reactive level the execution of a plan; and
- the reactive level, which in fact interacts with the environment, must be as general as possible, and is implemented by a general purpose CPU.

7. 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 [9], 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 [9], 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 4**. The alpha and beta memories are pre-allocated at compiling time.

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 4**).

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. The instruction address corresponding to <label> is popped from stack when a mismatch occurs and the program jumps to it. If the stack is empty, the match failed.

http://dx.doi.org/10.5772/intechopen.79265

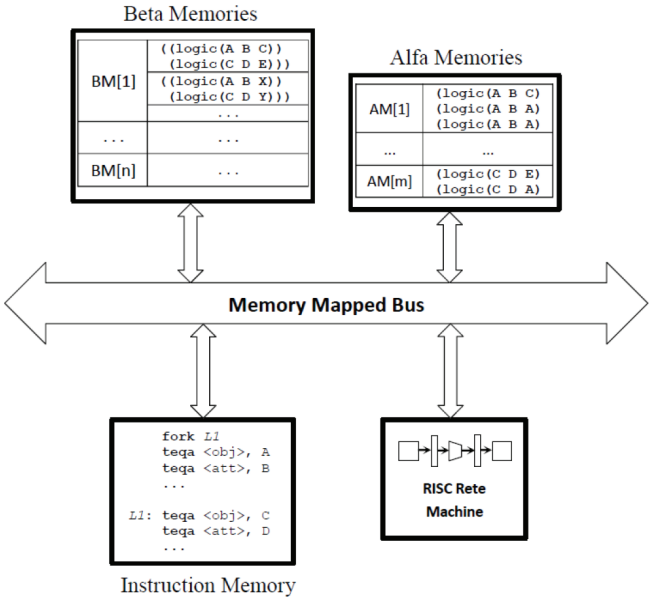

Figure 4. Architecture of the *Rete* processor.

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

8. The cognitive module

Figure 3 shows that the computational module associated with the cognitive level of the CAA (and from now on referred to as cognitive module) contains a *Graphplan* module that communicates with a *Rete* module, whose architecture was presented in the previous section. The objective of the cognitive module is to solve planning problems using graph planning algorithms. The *Graphplan* algorithm was used as an inspiration for the conception of the architecture of this module.

This module needs to be generic enough to execute not only the *Graphplan* algorithm, but also algorithms based on planning graphs or that uses *Graphplan* as heuristic function. Also, it should have two execution modes: the search mode, where a solution for the planning problem is searched, and the monitor mode, the monitors the execution of found plans.

The system level model of this module consists of two general purpose processors with individual instruction memory (i.e. executing independent programs) and a *Rete* module communicating with each other through a bus. **Figure 5** illustrates this architecture.

The taxonomy of the task this module should execute justifies this architecture. During planning, two main processes should be executed: the expansion of the states and actions graph, and the recursive search for a solution whenever the goal state is found in the graph. With the architecture described it is possible to execute these processes in parallel.

With this approach, when the goal state is found, the solution search can be started while the graph expansion goes on. Hence, even if the found goal set do not produce a solution, the expansion made progress in parallel, until another valid goal set is found, when the search is triggered again. The dynamics of *Graphplan* was modified for this module: a KBS is responsible for calculating the applicable actions; this KBS is implemented by the *Rete* module. The sequence diagram in **Figure 6** illustrates this operation.

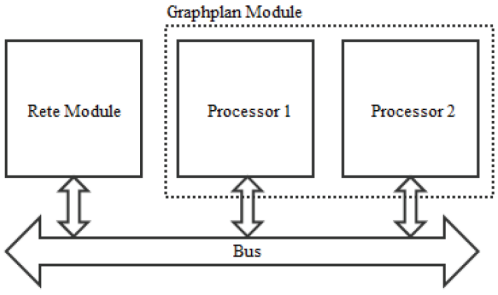

Figure 5. Block diagram of the cognitive module.

http://dx.doi.org/10.5772/intechopen.79265

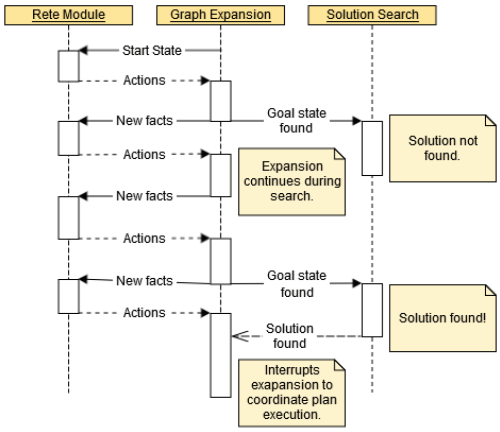

Figure 6. Sequence diagram of the cognitive module operation.

9. 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.

9.1. Domain definition

The architecture will be validated using the block world domain example, from [1]. **Figure 7** 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 will not 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.

9.2. Simulated tests: *Rete* module

The knowledge base shown in **Figure 7** is then compiled into the program presented in **Figure 8**. This program is the input for the simulator, in 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.

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*.

Rule Base

```
(Move(?b,?x,?y)
   (if      (logic (?b on ?x))
            (logic (?b state clear))
            (logic (?y state clear))
            (logic (?b type block)))
            (logic (?y type block)))
   (filter (!= ?b ?x))
   (then    (add (logic (?b on ?y))
                 (logic (?x state clear)))
            (rem (logic (?b on ?x))
                 (logic (?y state clear)))))


(MoveToTable(?b,?x)
   (if      (logic (?b on ?x))
            (logic (?b state clear))
            (logic (?b type block)))
   (then    (add (logic (?b on table))
                 (logic (?x state clear)))
            (rem (logic (?b on ?x)))))
```

Figure 7. Knowledge base for the block world domain example.

Figure 9 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 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.

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 10**. 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).

Finally, **Figure 11** 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.

```
        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 8. Code for the *Rete* network of the block world example.

```
================================
(+)(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 9. Output of the simulator for $(logic(C, on, A))$.

```
================================
(+)(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 10. Output of the simulator for $(logic(C, state, clear))$.

```
================================
(-)(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 11. Output of the simulator for deleting $(logic(C, on, A))$.

9.3. Simulated tests: cognitive module

For the cognitive module simulation the Akka library was used. The reason is that Akka implements the actor model of parallelism, which allows the program to contain routines running in parallel but with isolated execution contexts, communicating only through message passing; this suits the module specification given in Section 8.

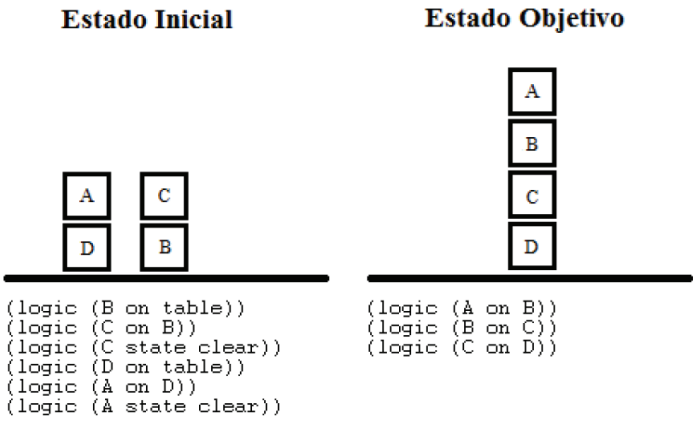

Figure 12. Start and goal states for the planning simulation.

As mentioned early in this chapter, one of the tasks of the cognitive module is to monitor plan execution. But this simulated experiment deals only with the execution of the *Graphplan* algorithm for planning problem solution, since this is the computationally heavier task that inspired the architecture.

The problem domain chosen for this test was, as in the *Rete* module, the block world domain. But this time, for the establishment of a planning problem, a start state and a goal state must be provided. Those are shown in **Figure 12**.

To show the execution of the experiment, the sequence diagram in the **Figure 13** was generated by the simulation[1]. In the image it is possible to see that the expansion sub-module (that represents the processor implementing the graph expansion) communicates with the *Rete* module to generate the state and action layers of the planning graph. This sub-module is also responsible for the calculation of mutex relations in each layer and for detecting the presence of the goal set in the most recent state layer. It can be seen that after three expansion cycles the goal is found and the search sub-module is triggered, starting the recursive search for a solution.

The solution is not found, and this is the point where the main advantage of the architecture became apparent: while the search sub-module tries to find a plan in the graph, the expansion and the *Rete* modules work together to generate a new layer in the graph. And, for this experiment, the new layer also has the objective state. The search module finds a solution this time, namely: $\langle \{moveToTable(A, D),\ moveToTable(C, B)\},\ \{move(C, table, D)\},\ \{move(B, table, C)\},\ \{move(A, table, B)\}\ \rangle$. The planning is then halted.

[1]The simulation generated a sequence of commands corresponding to the task being executed by each sub-module and the messages being exchanged between them, and the commands were transformed in the sequence diagram by the tool WebSequenceDiagrams (https://www.websequencediagrams.com/).

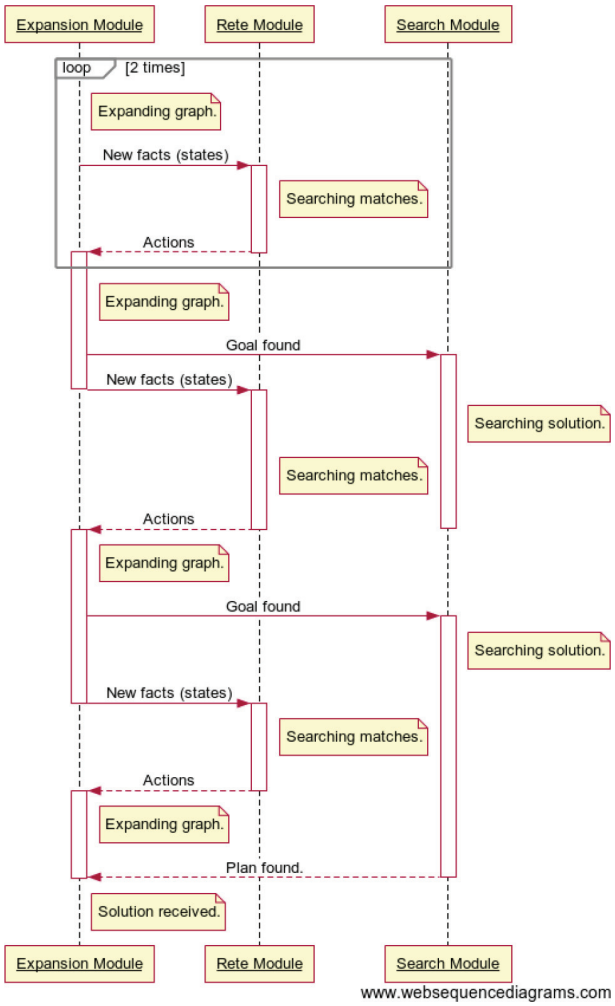

Figure 13. Sequence diagram for the simulation of the cognitive module.

10. Final considerations

This chapter presented the system level design of a SoC for the execution of cognitive agents. It uses the cognitive model of a agent architecture named concurrent autonomous agent (CAA). The SoC architecture was described in terms of the levels of the CAA and the tasks these levels execute. Two main computational modules of the SoC are the RISC *Rete* machine that employs an application specific ISA to detect matches in a KBS (in both the SoC correspondents of the instinctive and cognitive levels of the CAA) and the multiprocessed cognitive level dedicated to (graph) planning.

Simulated experiments shown that the architecture proposed for the aforementioned modules are valid in the sense of performing correctly their tasks. As future works, a formal verification method should be applied in order to validate the all modules separately and working together, as well as their computational and energetic performance, to show that the proposed architecture allows for low-level symbolic processing, which can give embedded and on-chip systems fast automatic reasoning capabilities, with low energy consumption.

Author details

Diego Ferreira*, Augusto Loureiro da Costa and Wagner Luiz Alves De Oliveira

*Address all correspondence to: diego.stefano@gmail.com

Federal University of Bahia, Salvador, BA, Brazil

References

[1] Russel S, Norvig P. Inteligencia Artificial. Rio de Janeiro: Elsevier; 2004

[2] Huhns MN, Singh MP. Cognitive agents. IEEE Internet Computing. 1998;**2**(6):87-89

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

[4] Langley P, Laird JE, Rogers S. Cognitive architectures: Research issues and challenges. Cognitive Systems Research. 2009;**10**(2):141-160

[5] Costa AL, Bittencourt G. From a concurrent architecture to a concurrent autonomous agents architecture. Lecture Notes in Artificial Intelligence. 1999;**1856**:85-90

[6] Bittencourt G, Costa ALd. Hybrid cognitive model. In: Third International Conference on Cognitive Science ICCS'2001 Workshop on Cognitive Agents and Agent Interaction; 2001

[7] Cerqueira RG, Costa ALd, McGill SG, Lee D, Pappas G. From reactive to cognitive agents: Extending reinforcement learning to generate symbolic knowledge bases. In: Simpsio Brasileiro de Automao Inteligente; 2013

[8] Ferreira DSF, Silva RRd, Costa ALd. Embedding a concurrent autonomous agent in a microcontrollers network. In: 5th ISSNIP-IEEE Biosignals and Biorobotics Conference (2014): Biosignals and Robotics for Better and Safer Living (BRC). IEEE; 2014. pp. 1-6

[9] Forgy CL. Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence. 1982;**19**(1):17-37

[10] Lehr TF. The implementation of a production system machine; 1985. Technical Report CMU-CS-85-126

[11] Peters M, Brink C, Sachweh S, Zündorf A. Scaling parallel rule-based reasoning. In: The Semantic Web: Trends and Challenges. Cham: Springer; 2014. pp. 270-285

[12] Panteleyev MG, Puzankov DV, Kolosov GG, Govorukhin IB. Design and implementation of hardware for real-time intelligent agents. In: 2002 IEEE International Conference on Artificial Intelligence Systems (ICAIS 2002). IEEE; 2002. pp. 6-11

[13] Frost J, Numan M, Liebelt M, Phillips B. A new computer for cognitive computing. In: 2015 IEEE 14th International Conference on Cognitive Informatics & Cognitive Computing (ICCI* CC). IEEE; 2015. pp. 33-38

[14] Kim Y, Shin D, Lee J, Lee Y, Yoo HJ. A 0.55 V 1.1m Wartificial-intelligence processor with PVT compensation for micro robots. In: 2016 IEEE International Solid-State Circuits Conference (ISSCC). IEEE; 2016. pp. 258-259

[15] Bittencourt G. The quest of the missing link. In: International Joint Conference of Artificial Intelligence; 1997

[16] Ferreira DSF, Paim CC, Costa ALd. Using a real-time operational system to embed a kinematic controller in an omnidirectional mobile robot. In: XI Simpósio Brasileiro de Automação Inteligente (SBAI) e XI Conferência Brasileira de Dinâmica, Controle e Aplicações (DINCON); 2013. pp. 1-6

[17] Forgy CL. On the Efficient Implementation of Production Systems. Dissertation. Department of Computer Science, Carnegie-Mellon University; 1979