Chapter

System Level Design and Conception of a System-on-a-Chip (SoC) for Cognitive Robotics

Diego Stéfano Fonseca Ferreira, Augusto Loureiro da Costa, Wagner Luiz Alves De Oliveira and Alejandro Rafael Garcia Ramirez

Abstract

In this work, a system level design and conception of a System-on-a-Chip (SoC) for the execution of cognitive agents in robotics will be presented. The cognitive model of the Concurrent Autonomous Agent (CAA), which was already successfully applied in several robotics applications, is used as a reference for the development of the hardware architecture. This cognitive model comprises three levels that run concurrently, namely the reactive level (perception-action cycle that executes predefined behaviours), the instinctive level (receives goals from cognitive level and uses a knowledge based system for selecting behaviours in the reactive level) and the cognitive level (planning). For the development of such system level hardware model, the C++ library SystemC with Transaction Level Modelling (TLM) 2.0 will be used. A system model of a module that executes a knowledge based system is presented, followed by a system level description of a processor dedicated to the execution of the *Graphplan* planning algorithm. The buses interconnecting these modules are modelled by the TLM generic payload. Results from simulated experiments with complex knowledge bases for solving planning problems in different robotics contexts demonstrate the correctness of the proposed architecture. Finally, a discussion on performance gains takes place in the end.

Keywords: Autonomous Agents, Robotics, Hardware Design, Knowledge Based Systems, Transaction Level Modelling

1. Introduction

Behaviour-based robotics is a branch of robotics that studies techniques for the interaction of robotic agents with the environment using the perception-action cycle in a coordinated fashion. With the addition of cognition, these agents may use knowledge about the environment to perform more complex tasks [1–3]. In the context of artificial intelligence, the internal structure of those agents, i. e., their cognitive architectures, dictate how the problem-solving will take place [4].

An example of a cognitive architecture with successful applications in robotics is the Concurrent Autonomous Agent (CAA), an autonomous agent architecture for mobile robots that has already proven to be very powerful [5–7]. This agent possesses a three layer architecture, in which each layer is responsible for a different task: the reactive layer runs behaviours with a perception-action cycle; the instinctive layer coordinates reactive behaviour selection; and the cognitive layer does the high-level planning.

In this work, a system level hardware model of a System-on-a-Chip (SoC) for cognitive agents will be presented. This model was inspired by the cognitive architecture of the CAA. Therefore, the CAA will be described in Section 2. In Sections 3 and 4 the *Rete* and *Graphplan* algorithms are described, respectively, since they are at the core of the CAA. The SystemC and TLM 2.0 standards, the tools used to construct the models are presented in Section 5, followed by the presentation of the proposed architecture in Section 6. Results of experiments are shown in Section 7 and some final thoughts and conclusions are presented in Section 8.

2. The concurrent autonomous agent (CAA)

The Concurrent Autonomous Agent (CAA) is a cognitive architecture whose taxonomy is based on the generic model for cognitive agents, which is composed by the reactive, the instinctive and the cognitive levels [8]. The CAA levels are presented in **Figure 1** [5, 6], where the message passing between the levels is shown. The cognitive level generates plans that are executed by the instinctive level through the selection of reactive behaviours in the reactive level. [6].

Both the cognitive and instinctive levels apply a Knowledge Based System (KBS) for knowledge representation and inference. The KBS is composed by a facts base, a rules base and an inference engine, as shown in **Figure 2** [6].

The facts base contains atomic logical elements that represents knowledge that is known about the current state of the environment. The rules base contains a set of rules in the format `if PREMISE then CONSEQUENT`. The premise consists of a conjunction of ungrounded fact patterns that uses variables to increase expressiveness.

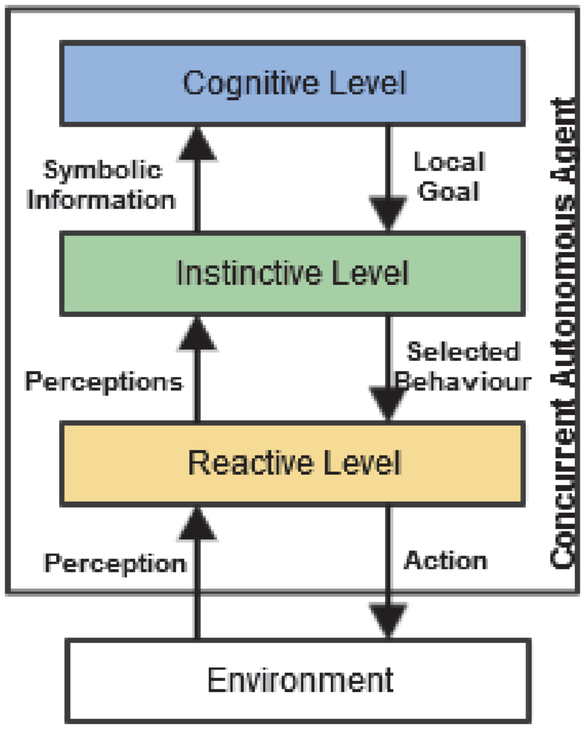

Figure 1.
Cocurrent autonomous agent architecture.

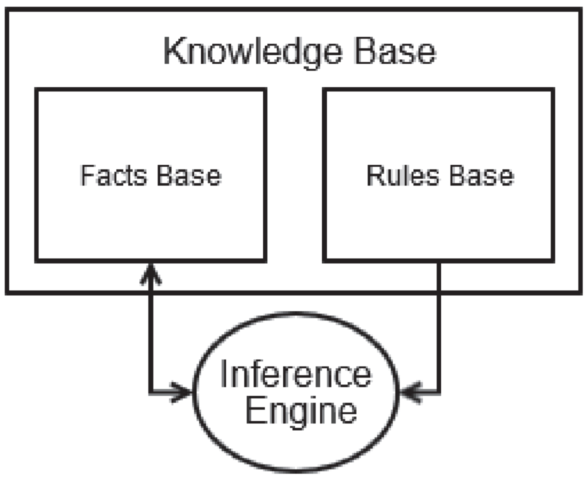

Figure 2.
KBS used by AAC.

The consequent, in turn, has instructions on how to modify the facts base and which message should be sent to other levels, if any. The KBS then goes through the following cycle [9].

- Recognition: identify which rules can be activated by checking if the premises matches the facts in the facts base;
- Conflict Resolution: among the activated rules (conflict resolution set), decide which should be executed; and
- Execution: the chosen rule in the conflict resolution phase has its consequent executed.

The *Rete* matching algorithm is applied in the recognition phase to generate the conflict resolution set. The instinctive level uses its KBS to select the appropriate reactive behaviour to be selected given the current world state. The cognitive level, in turn, uses its KBS inside the *Graphplan* algorithm (that will be described later in this chapter), in the state space expansion stage [10].

3. The *Rete* algorithm

As mentioned earlier in this chapter, the *Rete* matching algorithm is employed in the recognition stage the KBSs used by the CAA. It is proposed in [11], and is named after the latin word for "network".

The algorithm builds a graph out of the rules base of the KBS where each node has a special purpose. In the end, it avoids running through the entire facts base for each rule premise, every time a new fact arrives, by saving information about partial matches in some of its nodes [11].

The constructed graph has two portions: the alpha network, responsible for comparing the constants in the premises with the corresponding fields in the incoming fact; and the beta network, which checks for variable assignment consistency and maintenance of partial matches [10].

The nodes the compose the alpha network are the following [10]:

- *Root Node*: entry point for new facts;

- *Constant Test Nodes* (CTN): compares constant fields of premises with corresponding ones in the current fact; and
- *Alpha Memories* (AM): stores facts that successfully passed the tests in CTNs.

The beta network is composed by the following nodes [10]:

- *Join Nodes* (JN): perform tests that ensure variable assignment consistency inside a premise instance (partial match);
- *Beta Memories* (BM): stores partial matches produced in JNs; and
- *Production Nodes*: terminal nodes for full rule matches.

4. The *Graphplan* algorithm

The cognitive level uses the *Graphplan* algorithm to generate the plans that the other levels should execute. Originally, the algorithm used a propositional knowledge representation, so this will be adopted here for the algorithm description. The rest of this section uses [12, 13] as references.

Mathematically, a planning problem may be stated as $\mathcal{P} = \left(\Sigma, s_j, g\right)$, where $\Sigma = (S, A, \gamma)$ is the problem domain (that comprises the set of states S, the set of actions A and a state transformation function $\gamma = S \times A \rightarrow S$), s_j is the initial state and g is the goal state.

Each action $a \in A$ has a set $prencond(a)$ of precondition propositions and a set $effects(a) = effects^{+}(a) \cup effects^{-}(a)$ of effects. The effects, in turn, may be broken down into two subsets: $effects^{+}(a)$, the set of positive propositions (propositions to be added), and $effects^{-}(a)$, the set of negative propositions (propositions to be deleted). The applicability condition for an action a, in a given state s, may be written as $precond(a) \subseteq s$. The new state produced by the application of a would be $\gamma(s, a) = (s - effects^{-}(a)) \cup effects^{+}(a)$.

Consider an action layer A_j and the propositional layer P_{j-1} preceding it. A_j contains all actions a such that $precond(a) \subseteq P_{j-1}$, and P_{j-1} all propositions p such that $p \in P_{j-1}$. The so called planning graph is the built by connecting elements in P_{j-1} to elements in A_j by edges:

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

If two actions $a_1, a_2 \in A_j$ obey $effects^{-}(a_1) \cap \left(precond(a_2) \cup effects^{+}(a_2)\right) = \varnothing$ and $effects^{-}(a_2) \cap \left(precond(a_1) \cup effects^{+}(a_1)\right) = \varnothing$, they a said to be *independent*; if not, they are *dependent*, or *mutually exclusive* (*mutex*).

Propositions can also be *mutex*: p and q are *mutex* if every action in A_j that adds p is *mutex* with every action in A_j that produces q, and there are no actions in A_j that adds both p and q. Also, if a precondition of an action is *mutex* with a precondition of another action, the actions are *mutex*.

The algorithm begins by expanding the graph. 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:     $\mu A_{i+1} \leftarrow \{(a,b) \in A^2_{i+1}, a \neq b \mid Dependent(a,b) \vee \exists (p,q) \in \mu s_i : p \in preconds$
   $(a), q \in preconds(b)\}$
5:     $\mu s_{i+1} \leftarrow \{(p,q) \in s^2_{i+1}, p \neq q \mid \forall (a,b) \in A^2_{i+1} : p \in effects^+$
   $(a) \wedge q \in effects^+(b) \rightarrow (a,b) \in \mu A_{i+1}\}$
6: end procedure
```

The expansion stops when the goal state g is detected in the state layer s_i. It then triggers a recursive search for non-*mutex* actions in all the preceding action layers that could have produced the goal state found in s_i. This procedure is composed by the functions Search (Algorithm 2) and Extract (Algorithm 3).

Algorithm 2 Search for *non-mutex* actions.

```
1: procedure SEARCH($g$, $\pi_i$, $i$)
2:     if $g = \emptyset$ then
3:         $\Pi \leftarrow Extract(\cup\{preconds(a) \mid \forall a \in \pi_i\}, i-1)$
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:        $resolvers \leftarrow \{a \in A_i \mid p \in effects^+(a) \wedge \forall b \in \pi_i : (a,b) \notin \mu A_i\}$
11:        if $resolvers = \emptyset$ then
12:            return $Failure$
13:        end if
14:        nondeterministically 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 $\pi_i$
8:     end if
9:     return $Failure$
10: end procedure
```

5. SystemC and transaction level modelling

This work uses SystemC and the TLM 2.0 as modelling and simulation tools, so in this section they will be described.

5.1 SystemC

In the design of complex digital systems, obtaining a high-level executable specification of the project in early stages of the design process is useful for detecting errors or validate functionality prior to implementation. This is one of the main advantages of SystemC, a C++ class library for hardware design at various abstraction levels - from system level to Register Transfer Level (RTL). **Figure 3** shows the typical design flow for SystemC projects [14].

The SystemC library contains elements that facilitates representation of hardware systems parallelism. Hardware models in SystemC are represented by modules that may run in parallel interconnected by ports and channels (**Figure 4**). In this way, the initial model may contain a few modules representing system level

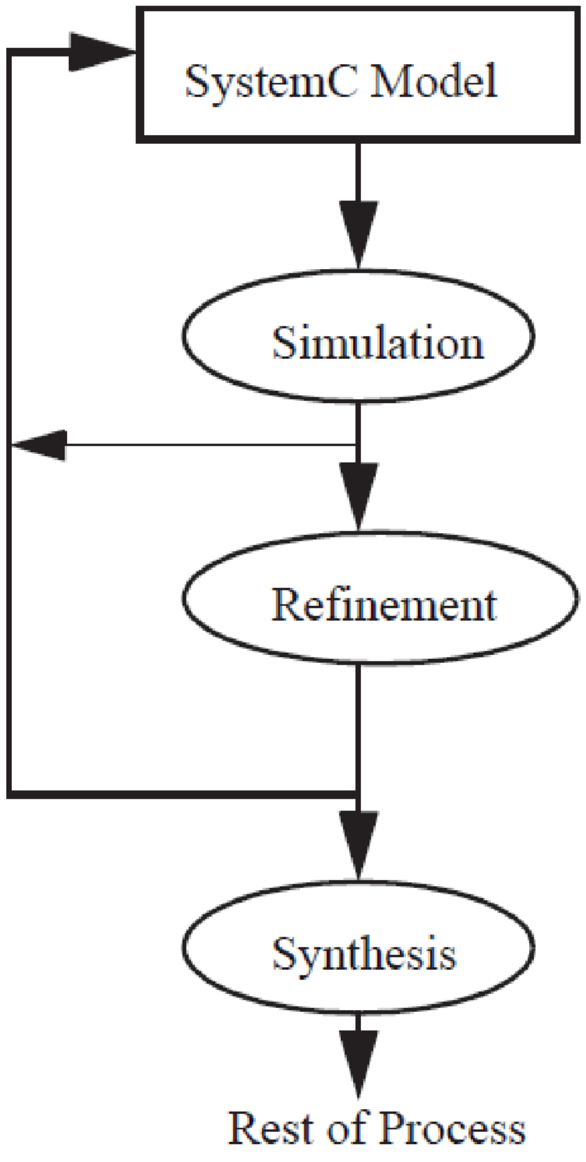

Figure 3.
SystemC hardware design flow [14].

functionality and, as the model gets refined, those initial high-level modules are further divided into more specific interconnected modules, until the RTL is reached [14].

5.2 Transaction level modelling

In hardware models of higher levels of abstraction, executing all modules at each time step may produce an unnecessary overhead. Thinking of a digital systems as components connected by a bus, reading from and writing to it, it would be more efficient to execute modules only when they have something massages to send/receive. This is the rationale behind Transaction Level Modelling (TLM), the message exchange being called a transaction [15].

With SystemC, an implementation of the TLM called TLM 2.0 is provided. It inherits all the SystemC capabilities, mainly the module concept, extending it with sockets, transactions and payloads (**Figure 5**).

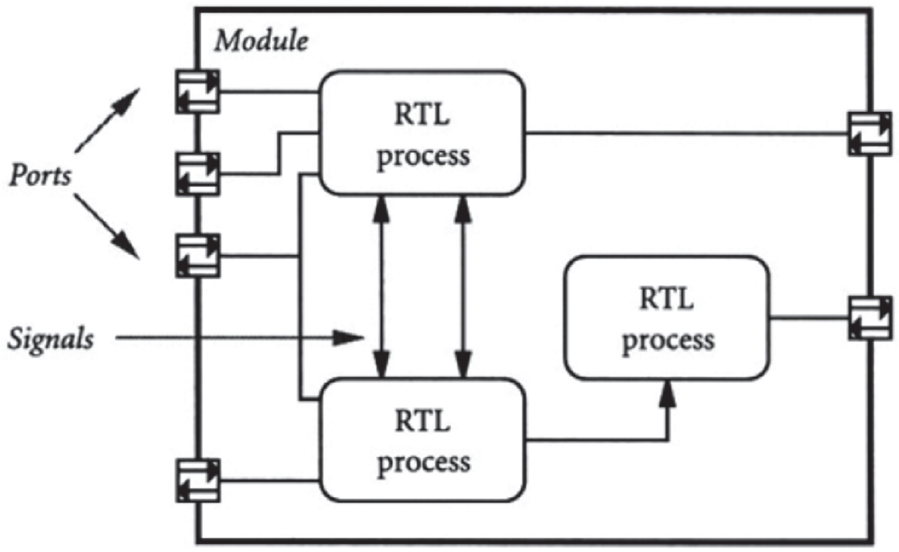

Figure 4.
Typical SystemC RTL module [14].

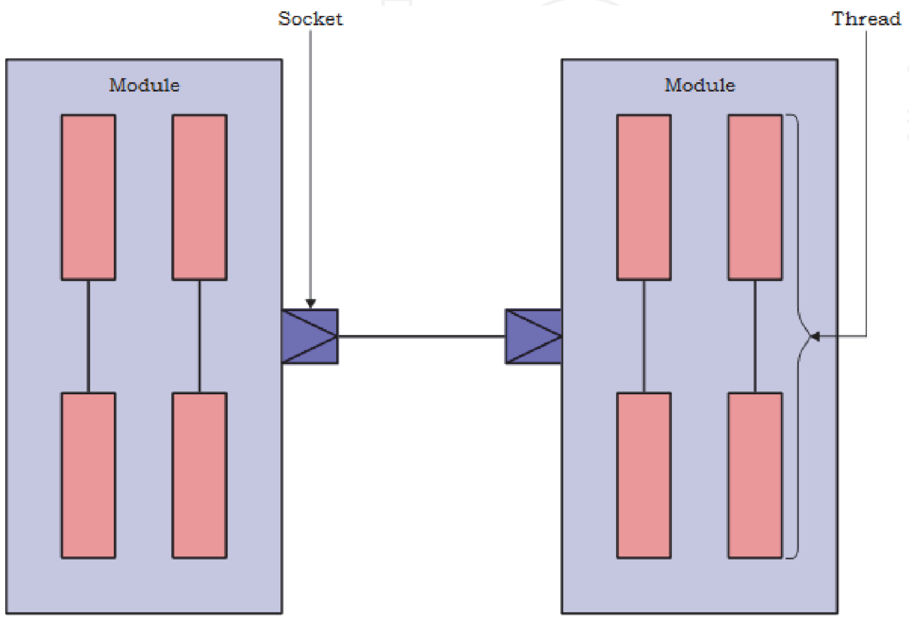

Figure 5.
TLM basic elements [15].

6. Proposed architecture

The hardware architecture proposed in this work takes full advantage of SystemC and TLM 2.0 capability of developing executable specifications from system level to RTL. In this sense, the approach employed was to obtain a high level model and validate its functionality using experiments in a robotics context.

The TLM model proposed is shown in **Figure 6**. It consists of modified SystemC model of the *Rete* processor presented in the authors previous work [10]. As can be seen in **Figure 6**, the instinctive module now implements a detailed *Rete* processor, that uses two Content Addressable Memories (CAMs) to implement the knowledge base and an auxiliary stage for test execution.

The Instruction Set Architecture (ISA) of the *Rete* processor described in [10] is still employed in this model, but now some tasks related to join node in the Rete network are performed separately in the Join Node Module.

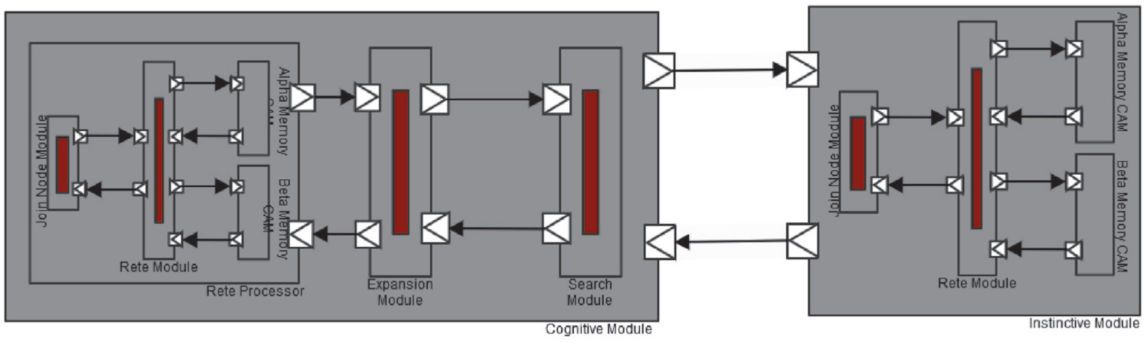

Figure 6.
TLM model of the SoC.

7. Results

7.1 Problem domain and simulation environment

The experiments were performed using the Webots R2021a robotics simulator. In the context of the CAA, the reactive level of the agent was

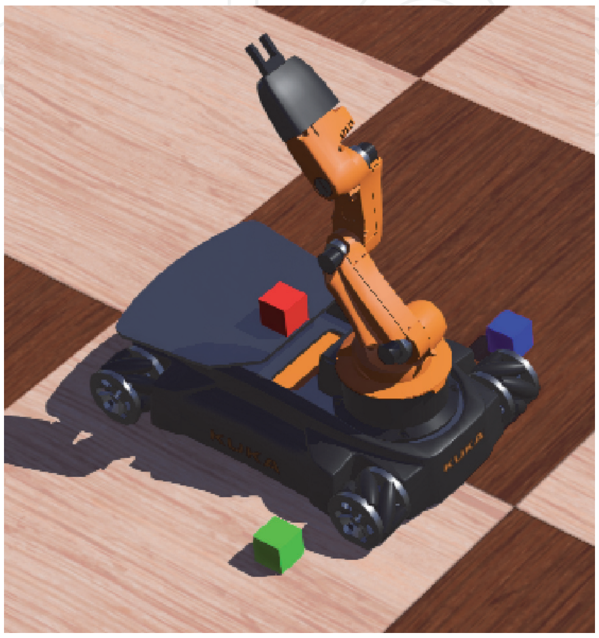

Figure 7.
Simulation environment for start state.

implemented inside this simulator, in the form of behaviours and controllers the interface with the environment. In the simulator, the planning problem domain was constructed: a simplified version of the blocks domain. The simulation consisted of three coloured boxes (red, green and blue) disposed in a given order around KUKA Youbot robot, which is a mobile robot with a robotic arm and a plate. The simulation environment and the robot in the initial state are shown in the **Figure 7**.

The planning problem consisted of reordering the blocks from the initial position shown in **Figure 7** so that the red block is in the left side or the arm, the blue in the right and the green in the front.

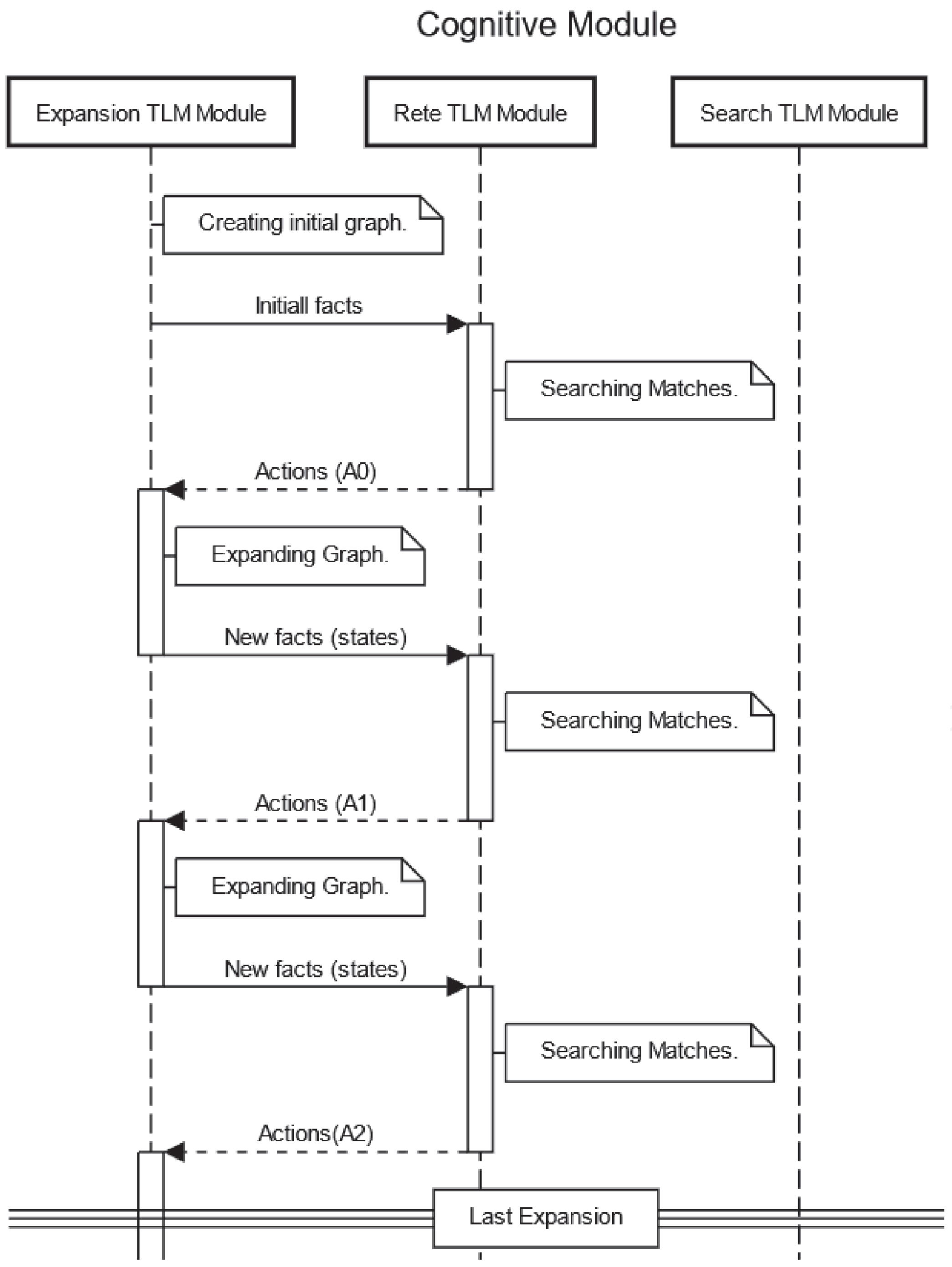

Figure 8.
Sequence diagram for graph expansion.

7.2 Cognitive module results

The operation of this module will be presented through a sequence diagram. The first part of this diagram, shown in **Figure 8**, shows how this level expanded the planning graph until what was labelled as last expansion.

The *Rete* and Expansion TLM modules together expand the planning graph: the current state is given as an input for the *Rete* module, that gives in return the next action layer. This is done 3 times, until action layer A2 is reached. The Expansion Module then processes the consequence of the newly added actions, updating the state layer. But this time, the goal state is present in the state layer, so a transaction is sent to the search module to backtrack the goal state checking if the actions that produced are mutex with any other. If no mutex relation is found, those actions form the plan. And, as shown in **Figure 9**, this plan is, indeed, found in the first backtrack attempt.

As can be seen in **Figure 9**, during the search for a solution, the expansion continues to take place, but is interrupted when the Search Module reports the solution. The plan found to the given problem was composed by the actions *move (green, right, front)*, *move(blue, left, right)* and *move(red, back, left)*.

7.3 Instinctive module results

The reactive behaviours for the robotic arm were defined as: going to a reset position; moving to left, right, front or back; grip and release. In order to execute the actions produced by the cognitive level, a knowledge base was created and

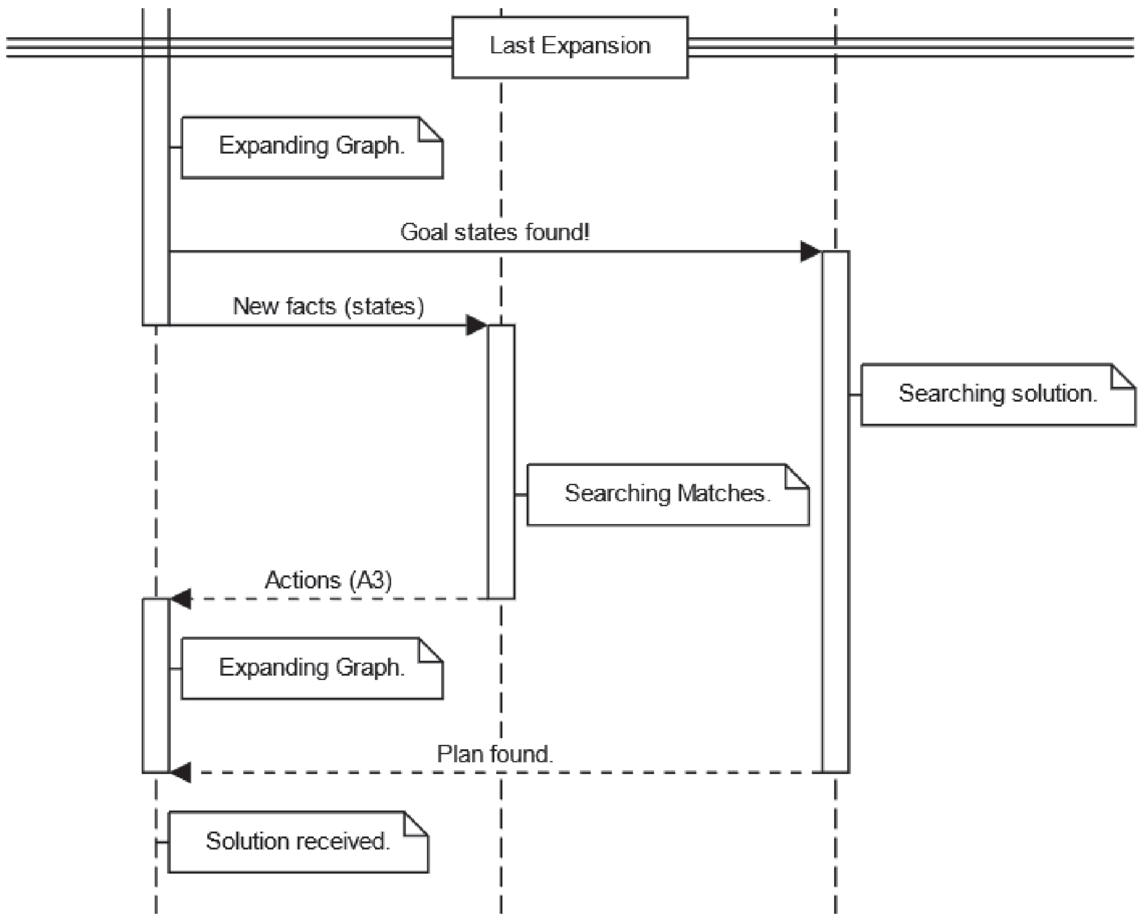

Figure 9.
Sequence diagram for finding a plan (continuation of ***Figure 8****).*

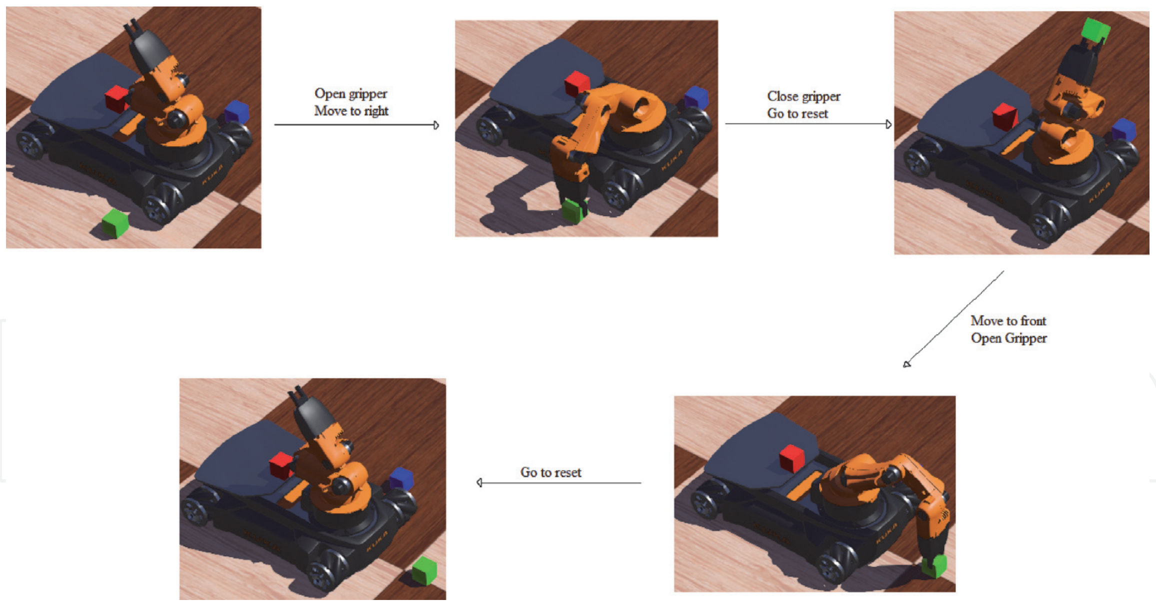

Figure 10.
Sequence of arm configurations and the reactive behaviours executed between them.

compiled for the *Rete* processor using its application specific ISA. The rules in this knowledge based were "grab" and "put". Both has as precondition that the arm is in the reset position and variables to specify the side where to grab from and the side where to put. The sequence of reactive behaviours activate by the instinctive level is show for the execution of the first action of the plan (*move(green, right, front)*) in **Figure 10**.

8. Conclusions

This chapter presented a SoC for cognitive agents that can perform symbolic computations at the hardware level. The cognitive model of the CAA was used as a reference for the hardware system-level model development, mapping its instinctive level to module with an application specific processor that executes the *Rete* matching algorithm, and with its cognitive level mapped into a module specifically designed for running the *Graphplan* planning algorithm (also with the use of the *Rete* processor). The SystemC and the TLM were used to build executable specifications that could validate its functionality in a robotics context. This version of the model was presented in a unified fashion, using SystemC/TLM modules and threads for the executable specification generation.

The results shown that the planning problem was solved by the Cognitive Module of the proposed architecture and successfully executed by its Instinctive Module, that consists of a *Rete* processor. By using a parallel architecture, the Cognitive Module broke the planning task into concurrent tasks in such a way that the backtrack search of the plan could take place while the graph were still expanding, as shown in **Figure 9**. In a complex planning problem this is advantageous because the solution usually does not come from the first backtrack search; thus, by not stalling the graph expansion, performance is gained.

In future works, tests with more complex knowledge bases and planning domains will be performed. Also, further refinements should be made in the architecture aiming synthesis.

Author details

Diego Stéfano Fonseca Ferreira[1]†, Augusto Loureiro da Costa[1]*†,
Wagner Luiz Alves De Oliveira[1]† and Alejandro Rafael Garcia Ramirez[2]†

1 Robotics Laboratory, Electrical Engineering Department, Federal University of Bahia, Salvador, BA, Brazil

2 Computer Engineering Department, University of Vale de Itajaí, Itajaí, SC, Brazil

*Address all correspondence to: augusto.loureiro@ufba.br

† These authors contributed equally.

References

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

[2] Guzel MS, Bicker R. A behaviour-based architecture for mapless navigation using vision. International Journal of Advanced Robotic Systems. 2012;9(1):18.

[3] Nattharith P, Güzel MS. Machine vision and fuzzy logic-based navigation control of a goal-oriented mobile robot. Adaptive Behavior. 2016;24(3):168–180.

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

[5] Costa ALd, Bittencourt G. From a concurrent architecture to a concurrent autonomous agents architecture. Lecture Notes in Artificial Inteligence. 1999;1856:85–90.

[6] Bittencourt G, Costa ALd. Hybrid Cognitive Model. In: The Third International Conference on Cognitive Science ICCS'2001 Workshop on Cognitive Angents 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: Simpósio Brasileiro de Automação Inteligente 2013; 2013.

[8] Bittencourt G. In the quest of the missing link. International Joint Conference of Artificial Intelligence. 1997.

[9] Brachman R, Levesque H. Knowledge Representation and Reasoning. Elsevier; 2004.

[10] Ferreira D, da Costa AL, De Oliveira WLA. IntelliSoC: A system level design and conception of a system-on-a-Chip (SoC) to cognitive agents architecture. Applications of Mobile Robots. 2019:199.

[11] Forgy CL. On the Efficient Implementation of Production Systems. Carnegie-Mellon University; 1979.

[12] Ghallab M, Nau D, Traverso P. Automated Planning: Theory and Practice. Elsevier; 2004.

[13] Blum AL, Furst ML. Fast planning through planning graph analysis. Artificial intelligence. 1997;90(1):281–300.

[14] Synopsys I. SystemC 2.0 User's Guide; 2002].

[15] Bennett J. Building a Loosely Timed SoC Model with OSCI TLM 2.0. embecosm Application Note 1. 2008;(1).