Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2019) 000–000

Procedia
Computer Science

www.elsevier.com/locate/procedia

Postproceedings of the 9th Annual International Conference on Biologically Inspired Cognitive Architectures, BICA 2018 (Ninth Annual Meeting of the BICA Society)

CogArch-ADL: Toward a Formal Description of a Reference Architecture for the Common Model of Cognition

Oscar J. Romero*

Carnegie Mellon University, Department of Machine Learning, USA

Abstract

As the Common Model of Cognition (CMC) is getting more supporters from research fields such as AI, cognitive science, neuroscience, and robotics in an effort to contribute to the understanding of minds, a new requirement becomes imperative: standard modeling and specification mechanisms are needed to allow both developing new CMC-compliant computational frameworks and validating existing ones. Thus, this paper aims at proposing an approach to formally describe cognitive architectures by extending π-ADL, an Architecture Description Language based on π-Calculus. Four case studies illustrate the usefulness of our approach, and future work outlines how it can be used as a vehicle to formally meet architectural requirements, validate structural/behavioral equivalence among architectures, and model evolutionary cognitive systems.

© 2019 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the scientific committee of the 9th Annual International Conference on Biologically Inspired Cognitive Architectures.

Keywords: Cognitive Architectures; Common Model of Cognition; Architecture Description Language; π-Calculus

1. Introduction

The Standard Model of Mind [8] (later renamed as Common Model of Cognition – ***CMC*** for short) captures a community consensus over a coherent region of science, serving as accumulative reference point for the field that can provide guidance for both research and applications, while also focusing efforts to extend or revise it. For the time being, the CMC has been grounded in three main cognitive architectures (hypotheses about the fixed structure of the mind that can be concretized into computational frameworks): ACT-R [1], SOAR [9], and SIGMA [17]. Both key assumptions and constraints about the purpose of the structure of the model have been identified. Although the CMC reflects a sound beginning upon which further integrative research can be conducted, some open issues become evident at this point: in its current state, the CMC defines high-level assumptions reflecting a consensus on what must be included

* Corresponding Author: Oscar J. Romero
E-mail address: oromero@cmu.edu

1877-0509 © 2019 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the scientific committee of the 9th Annual International Conference on Biologically Inspired Cognitive Architectures.

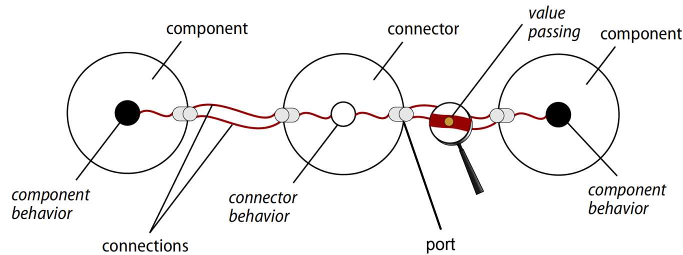

Fig. 1. Architectural Concepts of π-ADL. Source: [4]

in a cognitive architecture (***CA*** hereafter) in order to provide a human-like mind, but no formal description about how CMC-compliant CAs should be modeled/implemented is provided yet. Thus, there is a gap between the theoretical (abstract) assumptions set forth by the CMC and possible concretizations of those assumptions into appropriate computational frameworks (which can be either new implementations or validation of existing computational models). Quoting Bello and Bringsjord's own words [3]: *"The second problem [about CMC] is, put baldly, [its] perceived alignment of boxes and arrows doesn't constitute verifiable convergence. Yes, [it] perceive[s] a convergence between SOAR, ACT-R, and SIGMA, but absent any rigorous demonstration of such there is no rational reason to attribute this perception to underlying reality, rather than exuberance. Without (a) theorem(s) expressing that, at least with respect to some mental phenomena, initial convergence is in place, it is wishful thinking to aspire to reach, for the mind, what has been reached for the purely physical. Yet there is no such theorem, nor even an antecedent thereof (e.g., a rigorous conjecture), to be had in the inaugural paper [8]..."*

Therefore, the work presented in this paper is an attempt to partially bridge this gap by using and extending an Architecture Definition Language (ADL). An ADL, from a runtime perspective, provides a formal specification of the architecture in terms of components and connectors and how they are composed together. More specifically, we propose *CogArch-ADL*, an extension of π-ADL [15], a formal, well-founded theoretically language based on the higher-order typed π-Calculus. While most ADLs focus on describing software architectures from a structural viewpoint, π-ADL focuses on formally describing architectures encompassing both the structural and behavioral viewpoints. *CogArch-ADL* allows the CMC to be formalized in terms of: 1) a set of abstractions and architectural key elements identified by the CMC; 2) a set of architectural constraints, including legal compositions and cognitively plausible constraints; 3) a set of additional analyses that can be performed on CA descriptions constructed in the architectural style; and 4) formal validation of cognitive architectural models.

2. Concepts and Notation

2.1. Definitions

First of all, we need to introduce three concepts that we will use throughout the paper: Reference Model, Reference Architecture, and Software Architecture. According to [10], a *Reference Model* is a division of functionality together with data flow between the pieces, which defines a standard decomposition of a known problem into parts that cooperatively solve the problem. A *Reference Architecture* is a Reference Model mapped onto a system decomposition (software elements, and the data flows between them), which addresses the business rules, architectural styles that satisfy quality attributes, best practices of software development, and the software elements that support the development of systems for that domain. Finally, a *Software Architecture* is a particular characterization (concretization) of a Reference Architecture. In that context, the CMC corresponds to a Reference Model, and this paper presents an approach to the formal definition of a Reference Architecture for the CMC, more specifically, a *Reference Cognitive Architecture* (***RCA*** for short), which can be formally described through multiple levels of abstraction (as we will see in the remaining sections of this paper). Thus, the RCA will help to: provide a common language for the various scientist and developers in the field, provide consistency of implementation of new CAs, support the validation of existing CAs against a proven RCA, and encourage adherence to common standards, specifications, and patterns by using formal meta-modeling languages such as π-ADL.

π-ADL is a language encompassing both structural and behavioral architecture-centric constructs, defined as a domain-specific extension of the process algebra π-Calculus [12], that achieves Turing completeness (i.e., in π-Calculus every computation is possible but not necessarily easy to express) and high architecture expressiveness with a simple formal notation. In π-ADL, an architecture is described in terms of components, connectors, and their composition. Figure 1 depicts its main constituents. *Components* are described in terms of external ports and an internal behavior. Their architectural role is to specify computational elements of a software system. *Ports* are described in terms of connections between a component and its environment. Their role is to put together connections providing an interface between the component and its environment. *Protocols* may be enforced by ports. *Connections* are basic interaction points. Their role is to provide communication channels between two architectural elements. A component can send or receive values via connections, which can be declared as output, input, or input-output. *Connectors* are special-purpose components, described in terms of external ports and an internal behavior. However, their role is to connect together components. Components and connectors can be composed to construct composite elements, which in turn can be decomposed and recomposed in different ways in order to construct different compositions. Architectures are composite elements representing systems, so an architecture can itself be a composite component in another architecture, i.e. a sub-architecture. In π-ADL, architectures, components, and connectors are formally specified in terms of typed abstractions over behaviors.

2.2. Formal System

The semantics of π-ADL can be formalized by means of typing and transition rules.
Typing Rules: it can be read "if the statements in the premises listed above the line are established, then the conclusion below the line is derived" (see Figure 2). Thereby: if $Premise_1, ..., Premise_n$ are well-typed then Conclusion is well-typed. Rules with no premises are called axioms. Rules with one or more premises are called inference rules.

Typing rule $\dfrac{\text{Premise}_1 \ldots \text{Premise}_n}{\text{Conclusion}}$ Transition rule: $\dfrac{P_1 \xrightarrow{\alpha 1} P_1' \ldots P_n \xrightarrow{\alpha n} P_n'}{C \xrightarrow{\alpha} C' \textit{ where } \text{side effects}}$ side conditions

Fig. 2. Typing and Transition Rules. Source: [14]

Transition Rules: In a transition rule (see Figure 2), premises and conclusions are transition relations. Thereby, if the transition relations labelled by $\alpha_1...\alpha_n$ can fire, then the transition relation labelled by α can fire, i.e. if P_1 can fire α_1 and become P_1'... and P_n can fire α_n and become P_n', then C can fire α and become C'. Side conditions can be seen as preconditions on terms expressed in the premises and side effects as post-conditions on terms expressed in the conclusion. This structural operational semantics represents behavior (and thereby computation) of the π-ADL by means of a deductive system, expressed by the transition system. Following this approach, the π-ADL semantics are completely formalized (see [14] for details).
Abstract Syntax: The abstract syntax of types for expressing typed behaviors is defined on Figure 3. Below, we briefly describe some of the syntax constructs (however, a detailed description can be found in [15, 16]) and the proposed extended types for *CogArch-ADL*:
choice: *choose* $\{behavior_0\}$ *... or* $\{behavior_n\}$ expresses the capability of a behavior to choose either the capability of behavior$_0$ or behavior$_n$. When one of the capabilities is exercised, the others are no longer available.
composition: *compose* $\{behavior_0$ ... and $behavior_n\}$ expresses the capability of a behavior to parallel compose the capabilities of $\{behavior_0$ ... and $behavior_n\}$.
prefixes: An output prefix ***via*** *c* ***send*** *v* expresses the capability to send a value *v* via the connection *c*. An input prefix ***via*** *c* ***receive*** *v* expresses the capability to receive *v* via *c*. The silent prefix ***unobservable*** expresses the capability to enact an action invisibly, i.e. internally.
CogArch-ADL types: we define an abstract construct called *Buffer* that servers to extend more specific types of buffers (e.g., visual, verbal, etc.). Furthermore, specific units (chunks) of information are defined depending on the kind of content to be stored (e.g., stimulus, percepts).
Structural Operational Semantics: every construct of the π-ADL expressed in its abstract syntax has its structural operational semantics specified by a set of transition rules (Figure 4). The three labeled transition rules, i.e. *Output, Input*, and *Unobservable* define the semantics of the π-ADL constructs expressing respectively the output action ***via***

Abstract syntax of behaviors and values

```
syntax of behaviors

behavior ::= type . behavior
           |   value . behavior
           |   prefix . behavior
           |   if (boolean)     then { behavior1 }
                                else { behavior2 }
           |   choose { behavior0... or behaviorn }
           |   compose {behavior0... and behaviorn }
           |   decompose behavior
           |   replicate behavior
           |   abstraction ( expression0..., expressionn)
           |   constraint name1 is (constraint1)
           |   heuristic name1 is (heuristic1)
           |   check( constraint1 | heuristic1 )
prefix    ::= via connectionValue send value
           |   via connectionValue receive variable : ValueType
           |   unobservable
           |   if boolean do prefix
connection ::= connection name1
```

```
syntax of types and values
BaseType ::=  Any | Natural | Integer | Real | Boolean | String
           |  Behaviour
ConstructedType ::= tuple [ ValueType1, ..., ValueTypen ]
           |  set [ ValueType]
           |  inout [ ValueType ] | in [ ValueType ]
           |  out [ ValueType ]
           | ...

syntax of types in CogArch-ADL
Buffer     |  abstraction ( expression0..., expressionn)
Stimuli    |  Real // internal or external stimuli
Percept    |  Any // units of perception
ProcCont   |  Any // units of procedural memory content
DecCont    |  Any // units of declarative memory content
WMCont     |  Any // units of working memory content
Action     |  Any // actions (procedural and motor modules)
```

Fig. 3. Abstract syntax of typed behaviors and values. Keywords are written in bold, non-terminals are written in regular, alternative choices are separated by the symbol |. Extended from [14]

connection$_1$ ***send*** *value*$_1$, the input action ***via*** *connection*$_1$ ***receive*** *value*$_1$, and the internal action ***unobservable***. It means that in these three cases, we have three axioms that can always apply for firing atomic behaviors. A complete description of the structural operational semantics for behaviors and constraints can be found in [16].

Output:

$$\frac{}{\textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}\\ \textbf{and}\ (\textbf{via}\ \text{connection}_1\ \textbf{send}\ \text{value}_1 . \text{behavior}_1)\end{array}\right\} \xrightarrow{\textbf{via}\ \text{connection}_1\ \textbf{send}\ \text{value}_1} \textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ \text{behavior}_1\right\}}$$

Input:

$$\frac{}{\textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}\\ \textbf{and}\ (\textbf{via}\ \text{connection}_1\ \textbf{receive}\ \text{value} . \text{behavior}_1)\end{array}\right\} \xrightarrow{\textbf{via}\ \text{connection}_1\ \textbf{receive}\ \text{value}_1} \textbf{compose}\left\{\begin{array}{l}\text{constraint}_{0..n}\\ \textbf{and}\ (\text{value} = \text{value}_1)\\ \textbf{and}\ \text{behavior}_1\end{array}\right\}}$$

where (constraint$_{0..n}$ **and** (value = value$_1$)) *is consistent, i.e. binding* (value = value$_1$) *can be consistently asserted together with* constraint$_{0..n}$

Unobservable:

$$\frac{}{\textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ (\textbf{unobservable} . \text{behavior}_1)\right\} \xrightarrow{\tau} \textbf{compose}\left\{\text{constraint}_{0..n}\ \textbf{and}\ \text{behavior}_1\right\}}$$

Fig. 4. Formal Semantics of π-ADL: labeled transition rules for actions. Source: [14]

3. Case Studies

In the following we present 4 use cases to illustrate how CogArch-ADLcan be used to formally describe RCAs.

3.1. Describing a High-Level RCA

On Figure 5 we present a black-box formal description of the RCA focusing on the interfaces (i.e. ports and their connections) for components and connectors. In this first approach, we have described the five main cognitive modules of the CMC (i.e., *WorkMemMod, MotorMod, PercetionMod, DeclarativeMod*, and *ProceduralMod*), a connector that interfaces any two cognitive modules (i.e., *ModuleCon*), and the *RCA-CMC* architecture abstraction that provides bindings for components (modules) and connectors. ModuleCon connector declares two ports: *input* that comprises

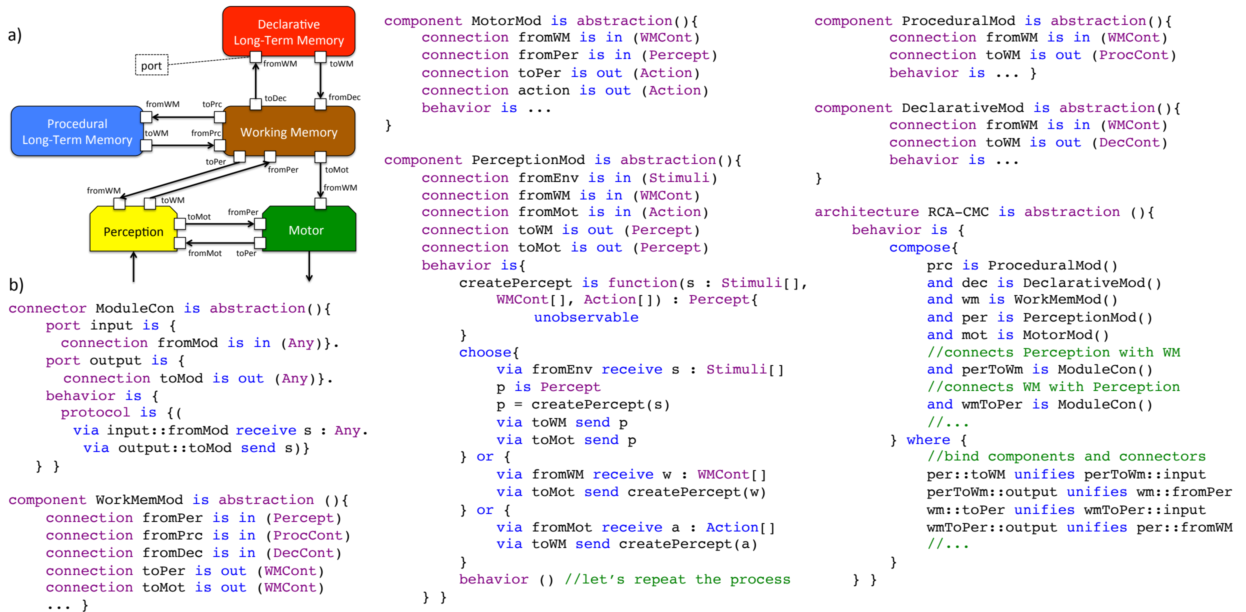

Fig. 5. High-level (black-box) formal description of a RCA for the CMC. a) Structure of CMC [8]. b) Excerpt of the Architectural Description using *CogArch-ADL*

the connection *fromMod* for receiving data from a specific module and *output* that comprises the connection *toMod* for passing the data to another module. The protocol enforced by these two ports is that the data received by the connector is automatically forwarded to the corresponding cognitive module. For the sake of simplicity, modules WorkMemMod, MotorMod, ProceduralMod, and DeclarativeMod do not explicitly show the "port" syntax as we do for ModuleCon, we abbreviate it and only present the "connection" syntax. Also, the behavior definition for these modules is hidden. Now, for illustration purposes, we have described the *behavior* construct for the PerceptionMod. This behavior is composed of two parts: 1) a function called *createPercept* that takes as input an array of type *Stimuli*, *WMCont*, or *Action*, then processes it (i.e., it may perform feature detection, object recognition, categorization, etc.), and returns a value of type Percept as a result; and 2) a construct that allows the behavior to repeatedly operate while choosing between perceiving (external) signals from the environment and perceiving (internal) signals from either the working memory (generated by proprioception or other internal processes) or the motor module (e.g., in order to create sensory-motor automatisms). PerceptionMod defines multiple ports and connections (e.g., fromEnv, fromWM, toWM, etc.) and then links them to the corresponding modules through the *receive* and *send* semantics, e.g., when a new external stimuli is received by *fromEnv* connection, a new percept is sent to the corresponding modules WorkMemMod and MotorMod through the connections *toWM* and *toMot* respectively. Finally, the *RCA-CMC* abstraction declares a composition of components (i.e., cognitive modules) and connectors, and defines the bindings for linking those elements through the formal construct *unifies*.

3.2. Describing a Cognitive Module

On Figure 5 we presented five modules as unitary components using a high-level (black-box) description, however, these modules can be further decomposed into multiple modules or sub-modules, such as multiple perceptual and motor modalities, multiple working memory buffers, semantic vs. episodic declarative memory, etc. In the following, we describe the internal behavior of a single component (the WorkMemMod) and its connectors in a white-box description. It is worth noting that we are not adopting any particular position regarding how working memory (WM for short) functions, we only want to exemplify how it could be structurally and behaviorally modeled using *CogArch-ADL* focusing on the decomposition of components (sub-modules) and the definition of component behaviors, but a consensus on which kind of sub-modules it is composed of is still necessary. For illustration purposes, we use the

```
component Buffer is abstraction (){unobservable}
component VisualBuffer is Buffer(){unobservable}
component VerbalBuffer is Buffer(){unobservable}
component GoalBuffer is Buffer(){unobservable}
component RetrievalBuffer is Buffer(){unobservable}
component MotorBuffer is Buffer(){unobservable}
//other possible components: PhonologicalLoop, etc.

component WorkMemMod is abstraction (){
    behavior aggregate() is { compose {
        vb is VisualBuffer()      and vbb is VerbalBuffer()
        and gb is GoalBuffer()    and mb is MotorBuffer()
        and rb is RetrievalBuffer() //...
    } where {//bindings...} }

    //(global) component functions
    getWMContents() : WMCont[] is function() {unobservable}
    updateWM(content : Any) is function() {unobservable}

    behavior retrieveFromLTM () is {
        via fromPrc receive pc : ProcCont[]     //sync
        forall{ d in pc suchthat
            (d.type = 'declarative') implies
            via toDec send d
            via fromDec receive dec : DecCont   //sync
            updateWM(dec)  // ... } }

    behavior initiateMotorAction () is {
        via fromPrc receive pc : ProcCont[]     //sync
        forall{ m in pc suchthat (m.type = 'motor') implies
            via toMot send content              //sync
            updateWM(content)  // ... } }

behavior influencePerception () is {
    via fromPrc receive pc : ProcCont[]     //sync
    forall{p in pc suchthat(p.type='perception') implies
        via toPer send p                    //sync
        updateWM(p)  // ... } }

behavior propagateActivation () is {
    spread(act : Real, contents : WMCont[]) is
        function() {unobservable}
    contents is WMCont[], contents = getWMContents()
    forall{ wm in contents suchthat
        (wm.isGoal = true) implies
            spread(wm.activation, contents)
        //... }
    propagateActivation() }

behavior decay() is {
    decayFx is function(wm : WMCont) : Real{unobservable}
    removeFromWM is function (wm : WMCont) {unobservable}
    contents is WMCont[], contents = getWMContents()
    forall{ wm in contents suchthat
        (wm.activation <> 0) implies
            wm.activation = decayFx(wm)
            if (wm.activation < threshold) then
                removeFromWM(wm)
            //... }
    decay() }
```

Fig. 6. Simplified excerpt for the Working Memory module using *CogArch-ADL*

buffer decomposition proposed in ACTR [1]. Figure 6 depicts five modality-specific memory buffers that extend the *Buffer* abstraction: *VisualBuffer, VerbalBuffer, GoalBuffer, RetrievalBuffer* and *MotorBuffer*, which together constitute an aggregate WM (the behavior of each of these buffers can be further described in detail, but for simplicity we keep them unobservable). The WM provides a temporary global workspace within which symbol structures can be dynamically composed from the outputs of perception and long-term memories. This composition is reflected by the use of the formal construct *compose* in the *aggregate* behavior, that way, WorkMemMod acts as the inter-component communication buffer for components. Furthermore, the WorkMemMod is composed of multiple capabilities (behaviors) that run in parallel performing multiple modifications on WM content as a consequence of a deliberate act initiated by the ProceduralMod on a single cognitive cycle. These behaviors are: 1) *retrieveFromLTM* that initiates retrievals from long-term declarative memories when a signal (fromPrc) coming from the ProceduralMod is received, then the results are deposited back in WM through the function *updateWM*; 2) *initiateMotorAction* that processes signals from ProceduralMod and initializes motor actions and/or performs internal simulation of an action; 3) *influencePerception* which performs perceptual acquisition and top-down influence to perception; 4) *propagateActivation* which spreads source activation from the current goal(s) to related nodes in WM to maintain them in a more active state relative to the rest of the memory (since it is not clear whether the goals should be generated by a separate module, e.g., an Intentional module, we simply assume they are stored in the WM by some external mechanism); and 5) *decay* that reduces the activation level of memory contents as time passes, and removes those contents that do not reach an activation threshold. The latter two behaviors run indefinitely, that is the reason why they call themselves at the end of their definition. Despite the fact of the significant parallelism that occurs across processes and behaviors in WorkMemMod, we have defined specific synchronization points (e.g., *via* [*port*] *send* [*data*], and *via* [*port*] *receive* [*data*]) where concurrent processes can talk to each other, providing support to the seriality necessary for coherent thought.

3.3. Describing Cognitive Constraints

In the previous case study we described a possible mechanism to model seriality constraints by synchronizing parallel behaviors via connections. In the following case study we show how to model two cognitive limitations imposed by the mind as architectural constraints that can be formally validated: duration of a cognitive cycle and WM capacity. For the former constraint, let's consider that a cognitive cycle operate at roughly 100 ms, corresponding to the deliberate act level in Newell's levels of cognition [13], assuming that a cognitive cycle is the sequence of

sense → *cognize* → *act*; and for the latter constraint, let's simply assume that WM has a limited capacity of 7 ± 2 chunks as proposed by [11]. Figure 7 depicts the architectural description of these two constraints.

```
architecture RCA-CMC is abstraction(){
    type starts is tuple[Any, Time]
    type ends is tuple[Any, Time]
    behavior ...
    // ...
    constraint cogCycleDuration(id : Any,
        prc : ProceduralMod) is {
        assert ( ends[id] – starts[id] <= 100 )
            when (prc.isActionSelected = true)
    } }
```

```
component WorkMemMod is abstraction(){
    type units is set[WMCont]
    behavior decay is {
        // ...
        check MemoryCapacity()
    }
    heuristic memoryCapacity is {
        assert (units.size >= 5
            and units.size <= 9)
    } }
```

Fig. 7. Modeling Cognitive Limitations as Constraints

Since the cognitive cycle's duration constraint involves the interaction of multiple (if not all the) modules and components, it should be defined at the architecture abstraction level (see *cogCycleDuration*). It is important to notice that due to cascading cognitive cycles (i.e., multiple overlapping cycles iterating at an asynchronous rate) we need to define at least two tuples (starts and ends) that map a cognitive cycle "id" (Any) with a Timestamp variable (Time), that is, when the cognitive cycle starts, the corresponding module (e.g., PerceptionMod) will create a new entry with metadata for the new cognitive cycle ([id, time]), and when the cycle finishes (e.g., when ProceduralMod selects and action) then the constraint is validated using the id to retrieve both the start and end times of the cycle. Now, the memory capacity constraint should be defined at the component abstraction level and should be considered a soft constraint (heuristic) rather than a hard constraint, that can be relaxed since the number 7 ± 2 may vary (in humans) depending on subject's gender, age, race, etc. A Heuristic constraint is taken to be a rule that should be observed, but may be selectively violated (see *memoryCapacity*). This constraint is intended to be satisfied at the very end of the decay behavior.

3.4. Architectural Reduction

The purpose of *Architectural Reduction* is to build a system that is as smallest as possible (rewriting it to the minimal), but meaningful for the analysis of its properties. This kind of approach may allow us to validate whether two or more CAs are structurally/behaviorally equivalent (i.e., isomorphic in [3]'s words, and convergent in [8]'s words), whether a CA is RCA-compliant, among others. In order to achieve this goal, π-Calculus provides a set of reduction and labeled transition semantics based on concepts such as structural congruence (i.e., if no difference between equivalent processes can be observed, then they can be considered behaviorally equivalent), bisimilarity (i.e., binary relation between state transition systems associating systems that behave in the same way, so one system simulates the other and vice versa), etc. Consider the following set of rules [18]:

$$P \equiv Q : \text{ structural congruence, where P and Q are processes} \quad (1)$$

$$P|Q \equiv Q|P : \text{ commutative relation, where | indicates parallel composition} \quad (2)$$

$$(P|Q)|R \equiv P|(Q|R) : \text{ associative relation} \quad (3)$$

$$\text{if } P \rightarrow Q\text{, then also } P|R \rightarrow Q|R : \text{ parallel composition does not inhibit computation} \quad (4)$$

$$\text{if } P \equiv P' \text{ and } P1 \rightarrow Q' \text{ and } Q' \equiv Q \text{ then } P \rightarrow Q : \text{ structurally congruent processes have the same reductions} \quad (5)$$

Now, a hypothetical, simplified and minimalist example using the 5 rules described above will help illustrate how the architectural reduction could be achieved. Suppose there are two CAs (CA_1 and CA_2) that use different representational components (symbolic and subsymbolic, respectively) that need to be validated whether they are isomorphic. Also assume both CAs can be defined by using these high-level processes: P: retrieve content from episodic memory, Q: retrieve content from semantic memory, R: do action selection (procedural), S: do episodic

learning, T: do semantic learning, and U: do procedural learning. Each process has two modalities, e.g., process R can be either R_1, a rule-based action selection, or R_2, a neural network model. Both CAs are supposed to have a similar operation: the procedural module selects an action (R), then content form declarative memories is retrieved (P and Q), and finally some learning mechanisms are triggered to create new symbols and relations (S, T, U). One of the main differences between both CAs is that CA_2 assumes that procedural learning may occur at different stages of the cognitive cycle. So, assume cognitive cycles of both CAs can be formally described (in a reductionist way) as follows:

$$CA_1 ::= R_1 \rightarrow (P_1|Q_1) \rightarrow (S_1|T_1)|U_1$$
$$cA_2 ::= (R_2|U_2) \rightarrow Q_2|(U_2|P_2) \rightarrow (T_2|U_2)|S_2$$

Thus, the architectural reduction would be:

$$\begin{aligned} CA_1 &\equiv CA_2 && Rules: \\ R_1 \rightarrow (P_1|Q_1) \rightarrow (S_1|T_1)|U_1 &\equiv U_2|R_2 \rightarrow U_2|(Q_2|P_2) \rightarrow (S_2|T_2)|U_2 && (1,2,3) \\ R_1 \rightarrow P_1|Q_1 \rightarrow S_1|T_1|U_1 &\equiv R_2 \rightarrow Q_2|P_2 \rightarrow S_2|T_2|U_2 && (1,3,4) \\ R_1 \rightarrow P_1|Q_1 \rightarrow S_1|T_1|U_1 &\equiv R_2 \rightarrow P_2|Q_2 \rightarrow S_2|T_2|U_2 && (1,2) \\ R_1 \rightarrow (S_1|T_1|U_1) &\equiv R_2 \rightarrow (S_2|T_2|U_2) && (1,4,5) \\ R_1 \rightarrow S_2|T_2|U_2 &\equiv R_2 \rightarrow S_1|T_1|U_1 && (1,3) \\ ... &\equiv ... \end{aligned}$$

After applying the reduction rules and rewriting both CAs to a "minimal" representation, they both seem to be isomorphic, however, due to the internal complexity of each module and since this is still a high-level description, further decomposition into processes and components should be done at such granularity level that atomic structural and behavioral equivalences can be identified.

4. Related Work

According to the evidence, efforts in defining a CMC have been focused more on the theoretical side rather than the practical, and do not provide a bridge between these two. One recent attempt to create a unified computational framework was the work on a "generic architecture for human-like cognition" [7], which conceptually amalgamated key ideas from the CogPrime, CogAff [19], LIDA [6], MicroPsi [2], and 4D/RCS architectures. A number of the goals of that effort were similar to those identified for the CMC; however, the result was more of a pastiche than a consensus assembling disparate pieces from across these architectures rather than identifying what is common among them [8]. A less recent work on the definition of an "agnostic" computational framework is COGENT [5], a computational modeling system that provides researchers with a flexible environment (using a box and arrow style specification language) within which to develop and explore symbolic and hybrid models of cognitive processes. The system provides a range of functions that allow scientists to explore their theories without commitment to a particular architecture. However, COGENT does not provide formal mechanisms to validate the structural/behavioral congruence of the cognitive models that can be created when using it.

5. Conclusions and Future Work

In this work we introduced *CogArch-ADL*, an approach to formally describe the structure and behavior of a Reference Architecture for the CMC. Some of the main benefits we envision when using *CogArch-ADL* are: 1) it will establish a proper mechanism to communicate architectural assumptions between scientists, researchers, and developers; 2) it will provide formalized models that can be used to further refine a comprehensive theory of cognition; 3) it

will serve as a vehicle to formally represent architectural commonalities among disparate CAs; 4) it will allow formal validation of architectural decisions; 5) it will allow the creation of not only ad hoc models but also architectural meta-models that can be described at several abstraction levels and will provide a basis for further implementation; and 6) it will potentially serve as a mechanism to formally determine whether two or more CAs are structurally/behaviorally equivalent with respect to the RCA. Future work is manifold: we need to deeply explore whether π-Calculus can be used as a tool to perform reverse-engineering on existing CAs and validate both their structural and behavioral isomorphic level. Also, we need to seek for tools that help us move from abstract ADL models to auto-generated code, through formal validation of π-Calculus semantic rules, in order to bridge the gap between abstract theoretical models and concretization of computational models. Finally, we want to investigate mechanism that allow us to define a RCA which modules and components are dynamic and evolve over time, that is, CAs that are RCA-complaint may support reconfiguration capabilities such as (original)composition→decomposition→(new)compositions.

Acknowledgments

This research was supported by Yahoo! and Verizon through the CMU-Yahoo InMind project.

References

[1] John R Anderson, Daniel Bothell, Michael D Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An Integrated Theory of the Mind. *Psychological review*, 111(4):1036, 2004.
[2] J. Bach. *Principles of Synthetic Intelligence*. Oxford University Press, 2009.
[3] Paul Bello and Selmer Bringsjord. Two Problems Afflicting the Search for a Standard Model of the Mind. IAAA Technical Report FS-17-05, U.S. Naval Research Laboratory, 2017.
[4] E. Cavalcante, T. Batista, and F. Oquendo. Supporting dynamic software architectures: From architectural description to implementation. In *IFIP CSA.*, pages 31–40, 2015.
[5] Richard P. Cooper, John Fox, and David W. Glasspool. *Modeling High-Level Cognitive Processes*. L. Erlbaum., NJ, USA, 2002.
[6] Stan Franklin and FG Patterson. The LIDA architecture: Adding new modes of learning to an intelligent, autonomous, software agent. *pat*, 703:764–1004, 2006.
[7] B. Goertzel, C. Pennachin, and N. Geisweiller. *Engineering General Intelligence, Part 1: A Path to Advanced AGI via Embodied Learning and Cognitive Synergy*. Atlants, 2014.
[8] J. E. Laird, C. Lebiere, and P. S. Rosenbloom. A standard model of the mind: Toward a common computational framework across artificial intelligence, cognitive science, neuroscience, and robotics. *AI Magazine*, 38:1–19, 2017.
[9] John Laird. *The Soar cognitive architecture*. MIT Press, 2012.
[10] P. Len Bass and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.
[11] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, pages 81–97, 1956.
[12] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.
[13] Allen Newell. *Unified theories of cognition*. Harvard University Press, 1990.
[14] F. Oquendo and I. Alloui. The archware architecture description language: Abstract syntax and formal semantics. In *ArchWare European RTD Project*, 2002.
[15] Flavio Oquendo. π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile. *SIGSOFT*, pages 1–14, 2004.
[16] Flavio Oquendo. Formally describing the software architecture of Systems-of-Systems with SosADL. In *SoSE*, June 2016.
[17] Paul S. Rosenbloom, Abram Demski, and Volkan Ustun. The Sigma Cognitive Architecture and System: Towards Functionally Elegant Grand Unification. *AGI*, July 2016.
[18] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
[19] A. Sloman. Varieties of Affect and the CogAff Architecture Schema. In *Emotion, Cog., and Affective Computing*, 2001.