A MODEL-DRIVEN APPROACH TO INTEGRATED COGNITION

by

Chad Peters

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Cognitive Science

Carleton University

Ottawa, Ontario

©2021

Chad Peters
Cognitive Architectures are used to test theoretical and conceptual frameworks identifying and explaining the underlying components of thought, namely the essential structures, mechanisms, and processes realized through models of human-like information processing. They define and prescribe those boundaries deemed both necessary and sufficient for intelligent agents based on our current understanding of human cognition.

The Common Model of Cognition (CMC) attempts to establish a community consensus on theoretical commitments and assumptions built into commonly used architectures, and group the assumptions according to structure and processing, memory, learning, and perceptual interfaces. The CMC remains a verbal-conceptual consensus across broad theories essential for general phenomenon (i.e., a Meta-Model of Cognition), however, lacks a formal specification to support domain-general task model comparisons for evaluation and validation of new theories implemented in new or existing architectures, or specific micro-theoretic implementations as cognitive architecture models and task models. Thus, a lack of a formal model supporting the CMC inhibits exploration of philosophical enquiries.
about core theoretical assumptions, and the adoption of refined peripheral theories across architectures.

This thesis presents a formal meta-model specific to the constraints represented at Newell’s Cognitive level using the principles of Model-Driven Design (MDD) to encapsulate these entities and relationships across architectures. This formal model can be used as a framework generator, and to specify the abstract interfaces common across Common Model agents, allowing modelers to explore verbal-conceptual theories through experimentation with virtual environments, further supporting a common ground. Frameworks generated through MDD support an empirical evaluation and comparison of variations on the Common Model for the purpose of application to Artificial Intelligence problem domains, lending additional credibility to computational cognitive modeling as a formal discipline, and the Cognitive Science research enterprise as a whole.
To Lenora, who never stopped believing in me.
ACKNOWLEDGMENTS

First and foremost I would like to thank my esteemed supervisors Dr. Babak Esfandiari and Dr. Robert West for their invaluable advice and guidance during my Ph.D. program. My gratitude extends to Mr. Tony Olsen for granting me the opportunity to complete my graduate studies unabated. I would like to thank my friends, lab mates, and research teams for so many memorable discussions about life, the universe, and everything in between. I also appreciate all of the support and encouragement I received from my parents, siblings, and extended family from around the world. Finally, I would like to express my most sincere and heartfelt gratitude to my wife and our four beautiful children — for their constant affirmation, patience, and understanding throughout these many years; without which it would have been impossible for me to complete my studies.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xiii</td>
</tr>
<tr>
<td>ABBREVIATIONS</td>
<td>xvi</td>
</tr>
<tr>
<td>GLOSSARY</td>
<td>xvii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 What’s in a Model?</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Theory versus Code</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Ontology versus Epistemology</td>
<td>5</td>
</tr>
<tr>
<td>1.4 The Common Model of Cognition</td>
<td>8</td>
</tr>
<tr>
<td>1.5 The Goal of Unification</td>
<td>9</td>
</tr>
<tr>
<td>1.6 Model-Driven Development</td>
<td>12</td>
</tr>
<tr>
<td>1.7 Research Problem</td>
<td>15</td>
</tr>
<tr>
<td>1.8 Moving Forward</td>
<td>18</td>
</tr>
<tr>
<td>2 Theoretical Background and State of the Art</td>
<td>22</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>22</td>
</tr>
<tr>
<td>2.2 Toward a Common Model of Cognition</td>
<td>23</td>
</tr>
<tr>
<td>2.3 Crossing System Levels</td>
<td>27</td>
</tr>
<tr>
<td>2.3.1 Issues with Level Specification</td>
<td>27</td>
</tr>
<tr>
<td>2.3.2 Model-Driven Design</td>
<td>29</td>
</tr>
<tr>
<td>2.4 Programming Cognition</td>
<td>32</td>
</tr>
<tr>
<td>2.4.1 Difficulty with Programming</td>
<td>33</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.5.2 Specification and Process</td>
<td>72</td>
</tr>
<tr>
<td>4.5.3 Meta-Model Definition</td>
<td>74</td>
</tr>
<tr>
<td>4.5.4 Metamodel Validation</td>
<td>82</td>
</tr>
<tr>
<td>4.5.5 Architecture Interface</td>
<td>102</td>
</tr>
<tr>
<td>4.5.6 Model-to-Text Generation</td>
<td>107</td>
</tr>
<tr>
<td>4.5.7 Summary of the Meta-Model</td>
<td>118</td>
</tr>
<tr>
<td>4.6 Mapping to a CMC Framework</td>
<td>118</td>
</tr>
<tr>
<td>4.6.1 Processing Cycle</td>
<td>118</td>
</tr>
<tr>
<td>4.6.2 Integrating CMU Lisp ACT-R</td>
<td>121</td>
</tr>
<tr>
<td>4.6.3 Integrating CCM Python ACT-R</td>
<td>132</td>
</tr>
<tr>
<td>4.6.4 Integrating Soar</td>
<td>137</td>
</tr>
<tr>
<td>4.6.5 Mapping to Virtual Environments</td>
<td>147</td>
</tr>
<tr>
<td>4.7 Experimental Design</td>
<td>150</td>
</tr>
<tr>
<td>4.7.1 Participants</td>
<td>151</td>
</tr>
<tr>
<td>4.7.2 Instruments</td>
<td>151</td>
</tr>
<tr>
<td>4.7.3 Procedures</td>
<td>152</td>
</tr>
<tr>
<td>4.7.4 Materials</td>
<td>152</td>
</tr>
<tr>
<td>4.8 Test Environments</td>
<td>153</td>
</tr>
<tr>
<td>4.8.1 Psychometric Tests</td>
<td>153</td>
</tr>
<tr>
<td>4.8.2 Classical Games</td>
<td>154</td>
</tr>
<tr>
<td>4.8.3 Real-Time Strategy</td>
<td>156</td>
</tr>
<tr>
<td>4.9 Measurements and Analysis</td>
<td>158</td>
</tr>
<tr>
<td>4.9.1 Evaluation Requirements</td>
<td>159</td>
</tr>
<tr>
<td>4.9.2 Architectural Measurements</td>
<td>160</td>
</tr>
<tr>
<td>4.9.3 Behavioral Measurements</td>
<td>163</td>
</tr>
<tr>
<td>4.9.4 Environmental Measurements</td>
<td>163</td>
</tr>
<tr>
<td>4.10 Validation of Research Questions</td>
<td>164</td>
</tr>
<tr>
<td>4.10.1 Meta-Model Definition</td>
<td>165</td>
</tr>
</tbody>
</table>
D.5 Learning in ACT-R ............................................. 262
E Soar Cognitive Architecture ................................. 264
   E.1 Evolution of Soar ........................................ 264
   E.2 Layout of Soar .......................................... 265
   E.3 Knowledge in Soar ...................................... 266
   E.4 Decision Making in Soar ................................. 266
   E.5 Learning in Soar ......................................... 267
F CMC Meta-Model Diagram .................................... 269
G Meta-Modeling Toolchain .................................... 270
   G.1 Eclipse Modeling Framework .............................. 270
   G.2 Graphical Modeling Framework .......................... 271
REFERENCES ......................................................... 273
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Newell’s System Levels.</td>
<td>6</td>
</tr>
<tr>
<td>1.2 Contributions to Knowledge and Methodology</td>
<td>19</td>
</tr>
<tr>
<td>1.3 Contributions to Practice</td>
<td>20</td>
</tr>
<tr>
<td>2.1 Comparison of Approaches for Mapping Architectures to a Common Model of Cognition</td>
<td>26</td>
</tr>
<tr>
<td>4.1 Functional Validation Requirements from CMC Assumptions</td>
<td>102</td>
</tr>
<tr>
<td>4.2 Mapping Measurement Methodology to Research Questions</td>
<td>103</td>
</tr>
<tr>
<td>5.1 Validated Mapping</td>
<td>173</td>
</tr>
<tr>
<td>5.2 Conflicts in Mapping</td>
<td>182</td>
</tr>
<tr>
<td>6.1 Updates to Meta-Model of Cognition</td>
<td>207</td>
</tr>
<tr>
<td>6.2 Suggested Updates to Architectures</td>
<td>214</td>
</tr>
<tr>
<td>A.1 Newell’s System Levels with Overlapping Time Scales</td>
<td>227</td>
</tr>
<tr>
<td>A.2 Hierarchy of Cognitive Architecture Sophistication</td>
<td>229</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 The Structure of the Standard Model, adapted from [Laird et al., 2017b]</td>
<td>8</td>
</tr>
<tr>
<td>1.2 Unification of Theories in Lakatosian Research</td>
<td>11</td>
</tr>
<tr>
<td>1.3 MDD provides layers of abstraction from representation to instantiation</td>
<td>13</td>
</tr>
<tr>
<td>1.4 Meta-Object Facility Layers for Newell's System Levels</td>
<td>16</td>
</tr>
<tr>
<td>2.1 The Structure of the Standard Model, from Laird et al. (2017b)</td>
<td>24</td>
</tr>
<tr>
<td>2.2 Metamodel Supporting Metacognition (MISM) levels of abstraction, adapted from [Caro et al., 2014]</td>
<td>30</td>
</tr>
<tr>
<td>3.1 Meta-Object Facility vs Newell's System Levels</td>
<td>43</td>
</tr>
<tr>
<td>3.2 MDD provides layers of abstraction from representation to instantiation</td>
<td>51</td>
</tr>
<tr>
<td>4.1 MDA software lifecycle, adapted from [Kriouile, 2015]</td>
<td>53</td>
</tr>
<tr>
<td>4.2 MDA processing cycle, adapted from [Kriouile, 2015]</td>
<td>55</td>
</tr>
<tr>
<td>4.3 Model Space vs Code Space in Model-Driven Architecture, adapted from Object Management Group [2014]</td>
<td>59</td>
</tr>
<tr>
<td>4.4 Meta-Object Facility layers for Common Model implementation, adapted from Caro et al. [2018]</td>
<td>61</td>
</tr>
<tr>
<td>4.5 CMC Platform interfaces and user groups</td>
<td>65</td>
</tr>
<tr>
<td>4.6 Example of ACT-R Key Monitor Binding</td>
<td>66</td>
</tr>
<tr>
<td>4.7 Example of ACT-R Model File with Key Press Command</td>
<td>67</td>
</tr>
<tr>
<td>4.8 Example of Soar Key Monitor Binding</td>
<td>67</td>
</tr>
<tr>
<td>4.9 Example of Soar Model File with Output Command</td>
<td>67</td>
</tr>
<tr>
<td>4.10 UML representation of Memory Classes</td>
<td>74</td>
</tr>
<tr>
<td>4.11 Meta-Modeling Process</td>
<td>78</td>
</tr>
<tr>
<td>4.12 CMC Meta-model Modules (connects with 4.14)</td>
<td>79</td>
</tr>
<tr>
<td>4.13 CMC Meta-model Buffers and Chunks</td>
<td>80</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>4.15</td>
<td>CMC Meta-model Productions</td>
</tr>
<tr>
<td>4.16</td>
<td>First candidate EMF class diagram for CMC Metamodel</td>
</tr>
<tr>
<td>4.17</td>
<td>CMC Metamodel Inter-Module Communications</td>
</tr>
<tr>
<td>4.18</td>
<td>Abstract Interface for Procedural Memory Factory</td>
</tr>
<tr>
<td>4.19</td>
<td>Abstract Interface for Declarative Memory Factory</td>
</tr>
<tr>
<td>4.20</td>
<td>Creation Methods for Abstract Factory Modules</td>
</tr>
<tr>
<td>4.21</td>
<td>Common Model Factory Structure</td>
</tr>
<tr>
<td>4.22</td>
<td>Model to Text Generation</td>
</tr>
<tr>
<td>4.23</td>
<td>CMC Meta-model Ecore XML</td>
</tr>
<tr>
<td>4.24</td>
<td>PyEcore Classes by MOF level [Aranega, 2017]</td>
</tr>
<tr>
<td>4.25</td>
<td>Architecture-Agnostic Production Tree</td>
</tr>
<tr>
<td>4.26</td>
<td>EMF Model definition in a tree layout</td>
</tr>
<tr>
<td>4.27</td>
<td>Interaction between Agent and Environment</td>
</tr>
<tr>
<td>4.28</td>
<td>Lisp ACT-R Model Initialization</td>
</tr>
<tr>
<td>4.29</td>
<td>Lisp ACT-R Model Cycle</td>
</tr>
<tr>
<td>4.30</td>
<td>Lisp ACT-R Perception Interface</td>
</tr>
<tr>
<td>4.31</td>
<td>Lisp ACT-R Perception Transduction and Update</td>
</tr>
<tr>
<td>4.32</td>
<td>Lisp ACT-R Motor Interface</td>
</tr>
<tr>
<td>4.33</td>
<td>Lisp ACT-R Motor Event Handling</td>
</tr>
<tr>
<td>4.34</td>
<td>Python ACT-R Class Structures</td>
</tr>
<tr>
<td>4.35</td>
<td>Soar SML Interface</td>
</tr>
<tr>
<td>4.36</td>
<td>Model trace of Soar counting task productions</td>
</tr>
<tr>
<td>4.37</td>
<td>OpenAI Gym Virtual Environments [Brockman et al., 2016]</td>
</tr>
<tr>
<td>4.38</td>
<td>Lisp ACT-R Playing StarCraft 2</td>
</tr>
<tr>
<td>4.39</td>
<td>Model trace of ACT-R counting task</td>
</tr>
<tr>
<td>4.40</td>
<td>Model trace of Soar counting task</td>
</tr>
<tr>
<td>5.1</td>
<td>Production System activity rate (per cycle) in Counting Model</td>
</tr>
<tr>
<td>5.2</td>
<td>Working Memory activity rate (per cycle) in Counting Model</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>5.3 Declarative Memory activity rate (per cycle) in Counting Model</td>
<td>197</td>
</tr>
<tr>
<td>5.4 Motor buffer activity in CartPole Environment</td>
<td>199</td>
</tr>
<tr>
<td>5.5 Lisp ACT-R Activity Frequency by Module in Beacon Test</td>
<td>200</td>
</tr>
<tr>
<td>5.6 Python ACT-R Activity Frequency by Module in Beacon Test</td>
<td>201</td>
</tr>
<tr>
<td>5.7 Soar Activity Frequency by Module in Beacon Test</td>
<td>201</td>
</tr>
<tr>
<td>6.1 CMC Validation Results</td>
<td>209</td>
</tr>
<tr>
<td>B.1 Taxonomy of Architectures by Memory and Learning (Anderson, 2003)</td>
<td>240</td>
</tr>
<tr>
<td>B.2 Taxonomy of Architectures by Traditional Representation (Duch et al., 2008)</td>
<td>241</td>
</tr>
<tr>
<td>B.3 Taxonomy of Architectures by Modularity (Sun, 2008)</td>
<td>242</td>
</tr>
<tr>
<td>B.4 Taxonomy of Architectures by Research Program (Gray et al., 2008)</td>
<td>242</td>
</tr>
<tr>
<td>C.1 Analysis of Soar, ACT-R, and Sigma with Respect to the Standard Model (Laird et al., 2017b)</td>
<td>244</td>
</tr>
<tr>
<td>D.1 The ACT-R Cognitive Architecture</td>
<td>258</td>
</tr>
<tr>
<td>D.2 ACT-R production system with buffers (Budiu, 2020)</td>
<td>260</td>
</tr>
<tr>
<td>E.1 Soar Cognitive Architecture (Laird et al., 2017b)</td>
<td>265</td>
</tr>
<tr>
<td>E.2 Explanation-Based Chunking in the Soar Cognitive Architecture (Laird, 2012)</td>
<td>268</td>
</tr>
<tr>
<td>F.1 EMF class diagram for CMC Metamodel</td>
<td>269</td>
</tr>
<tr>
<td>G.1 Eclipse Modeling Framework Tool Chain</td>
<td>271</td>
</tr>
<tr>
<td>G.2 Sirius graphical modeling example</td>
<td>272</td>
</tr>
</tbody>
</table>
ABBREVIATIONS

CMC: Common Model of Cognition
DM: Declarative Memory
DSL: Domain Specification Language
HLL: High-Level Language
LTM: Long-Term Memory
MDA: Model-Driven Architecture
MDD: Model-Driven Design
MOF: Meta-Object Facility
MTL: Model Transformation Language
OCL: Object Constraint Language
OMG: Object Management Group
PM: Procedural Memory
PS: Production System
STM: Short-Term Memory
WM: Working Memory
UML: Unified Modeling Language
XML: Extensible Markup Language
XMI: XML Metadata Interchange
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>A system situated within and a part of an environment, that senses that environment and acts on it, over time, in pursuit of its own agenda so as to effect what it senses in the future.</td>
</tr>
<tr>
<td>Architecture</td>
<td>The fundamental structures of a system and the discipline of creating such structures and systems.</td>
</tr>
<tr>
<td>Cognitive Architecture</td>
<td>The theory about the structures of the human mind and how they work together to manage intelligent behavior in any complex environment.</td>
</tr>
<tr>
<td>Cognitive Model</td>
<td>A description or representation of human cognitive processes for the purposes of comprehension and prediction.</td>
</tr>
<tr>
<td>Environment Model</td>
<td>A description of the environment in which an agent is situated.</td>
</tr>
<tr>
<td>Framework</td>
<td>An abstraction of general functionality in software that can be modified according to specific requirements.</td>
</tr>
<tr>
<td>Formal Model</td>
<td>A precise statement of model components and the relationships among them.</td>
</tr>
<tr>
<td>Formal Language</td>
<td>A well-formed grammar constrained according to a set of rules.</td>
</tr>
<tr>
<td>Model</td>
<td>A systematic description of an object or phenomenon that shares important characteristics with the object or phenomenon.</td>
</tr>
<tr>
<td>Meta-Model</td>
<td>A model used to describe the structures, semantics and constraints for a family of models.</td>
</tr>
<tr>
<td>Platform</td>
<td>The environment in which a piece of software is executed.</td>
</tr>
<tr>
<td>Requirement</td>
<td>Description of features of a system.</td>
</tr>
<tr>
<td><strong>Specification:</strong></td>
<td>Description of the functionality of a system and how it will perform.</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Standard:</strong></td>
<td>An authoritative principle or rule that usually implies a model or pattern for guidance; an approved model.</td>
</tr>
<tr>
<td><strong>Task Model:</strong></td>
<td>A model of the knowledge, steps and decisions an agent makes to accomplish a specific task.</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

One method of understanding the human mind is by modeling processes. Cognitive Architectures provide a method of identifying and explaining the underlying unified components of thought as instantiated through models of human-like information processing. The Common Model of Cognition (CMC) strives to identify areas of consensus, or general norms, between the most commonly used cognitive architectures, using ACT-R, Soar, and Sigma as exemplars.

In this thesis I will show that cognitive architectures can be formalized using an application of Model-Driven Design (MDD) methodology, and the theoretical and practical results of formalization at the Cognitive Level. The results presented in this thesis demonstrate the areas where the ACT-R and Soar architectures agree in implementation and can be mapped to a formal model, as well as some strong disagreements between ACT-R and Soar in both theory and code in categories purported to adhere to the same Common Model of Cognition. Finally, I argue that the results of this thesis demonstrate justification to consider model code as theory, as part of a formal Standard Model of Cognition, with ongoing contributions to a Lakatosian research programme.

1.1 What’s in a Model?

The word *model* connotes a different type of representation depending on the context in which it is used. In this thesis, it is used to describe abstracted representations that were created by two different disciplines — being Cognitive Science and Software Engineering — each with a unique history of theory and
This section will introduce and discuss some of these examples to help readers identify and understand their context and application in later chapters.

The term model, generally speaking, is a representation used to capture the properties and features of a concept or thing that is difficult to work with in its native form. Most readers are familiar with blueprints or models of airplanes, buildings, and bridges — all being smaller representations of a larger concept that is too difficult to discuss or modify at scale. Software models serve a similar purpose, and can be used to describe the most important properties of larger software systems using generalizations as a placeholder for a more complex implementation. The model itself is not a perfect replacement for thing being modeled, but is good enough for the sake of conversation and exploration of the most salient properties of that thing. The term meta-model is also used throughout this thesis, and generally means a representation of a family of models that all belong to a similar domain; a meta-model, then, is simply a model of related models.

In Cognitive Science, the term model is typically used to describe a computer simulation designed to test or predict some theory about human cognitive function or behavior. The cognitive model is typically captured in a text file that runs alone, or in conjunction with a suite of software tools designed to support it. When referring to models in a cognitive modeling context, I will use terms like Cognitive Model File to indicate the simulation file itself, a Task Model to describe the behavior that the file is designed to simulate, and Common Model to refer to the set of normative theories that are assumed by many cognitive modelers to exist in human cognition.

In Software Engineering, the term model is used to describe an abstract representation of a complex software system. A software model can take on different representations; examples include a graphical diagramming standard such as Unified Modeling Language (UML), or a text representation of abstract objects and
interfaces as used in Object Oriented Programming (OOP). These representations serve as a less-detailed placeholder of the system to be implemented on a specific platform, or for a particular purpose. A UML diagram, for example, serves as a visual guide to facilitate communication about software classes, software attributes, and their relationships with one another, in a way that is also easy to conceptualize without reading any software code. A software object, on the other hand, is used to describe a software application component or interface that can be used as a template for a specific implementation of that component or interface. Model-Driven Design (MDD), introduced in section 1.6 is a Software Engineering Methodology that applies these software meta-models, software models, and graphical diagrams, to design and generating these objects and interfaces for different computing platforms. The Meta-Object Facility (MOF), later discussed alongside MDD, is a more rigorous standard or application of MDD methodology, however, also uses similar terminology — such as models, meta-models, and architectures — in a different fashion than the Cognitive Modeling community might expect.

The reuse of terms, without additional context, will likely confuse the reader when referring to cognitive architectures, cognitive meta-models, or cognitive models implemented in, or represented at, a specific level of abstraction according to the MOF standard. Therefore, this thesis will preface these terms with Cognitive in the Cognitive Science context, and either MDD or MOF when referred to them in the context of Software Engineering.

1.2 Theory versus Code

Some argue that cognitive modelling code can constitute formal theories about cognition, similar to mathematical models in physics (Sun, 2009), while others argue...
that the code should be viewed as merely a tool for exploring the sort of conceptual models found in the humanities and the social sciences, particularly psychology (Cooper and Guest, 2014).

The main issue, which is agreed upon by everyone in this debate, is that the model code that runs on a computer incorporates a variety of elements that are unrelated to a specific theory being tested. Modelers must make significant choices between cognitive architectures, programming languages, platform implementations, different ways to frame the problem, different ways of connecting to the environment, assumptions about testing strategy, assumptions about prior knowledge, different sub-systems and modules, and various other assumptions needed to make the model run; however, many of these choices and assumptions are not part of the core model theory.

The question of whether cognitive models can be formally represented in a way that allows clear comparisons is extremely important. I would argue that until this question is resolved, it is unclear if cognitive modelling could ever become a formal scientific discipline, working toward a unified model of human cognition, as originally envisioned by Newell (1990) and later on by Sun (2009). If not, then Cognitive Modelling may end up being relegated to an inferior, dare I say pseudo scientific discipline, by the scientific community; limited to producing useful but incommensurate models in support of a variety of disparate psychology theories.

Despite this foreboding prelude, I also argue that it is possible to create a Standard Model of Cognition. This requires using a language that is unambiguous, supports defining abstract syntax, and follows a strict meta-modeling architecture. Further, I would also argue that the Meta-Object Facility can provide such a language, and that Model-Driven Development can be used to implement the process.
This thesis describes such a formal system, based on MDD, that achieves this goal, as well as a framework generated from this formal representation to test each cognitive architecture. When existing cognitive architectures are connected into a framework derived from this system, the models rendered in these architectures can be formally described, compared, evaluated, and perhaps most importantly — validated. To test this, the Soar architecture and two significantly different software instantiations of the ACT-R architecture were connected into the formal system. The results demonstrate that differences and similarities in theoretical and implementational commitments are clearly identifiable through an application of common model relationships defined in, and represented by, the formal system.

1.3 Ontology versus Epistemology

The goals of formalization and unification have a different meaning depending on whether cognitive models and cognitive architectures are understood as epistemic or ontological (see Appendix A for additional theoretical models). The epistemology of cognitive models deals with the different perspectives, or ways of looking at and analyzing them; we can use different lenses to see and describe and understand various facets of cognitive processes. The ontology of cognitive models, on the other hand, deals with the relationships between various concepts and components that may exist at varying levels of abstraction of the same perspective.

One example (and very popular) perspective, a cognitive epistemological system, was Marr’s Tri-Level Hypothesis. According to Marr (1982), there are three levels of analysis: computational, algorithmic, and implementational; each providing a slightly different way to define and communicate the why, what, and how behind cognitive processes. Dawson (1998) later added a fourth, architectural level, as a means to
describe the interplay between the algorithm and physical implementation (Dawson, 2013).

Computational models, then, can be thought of as epistemological tools. Cognitive models, and the architectures they run on, can be used to describe useful lenses for interpreting experimental data; however, they are not meant as formal models of brain functioning, as they are specific implementations that cannot be reused in other environments or tasks.

Another example used to understand cognitive processes, an ontological structure of cognition, is Newell’s Systems Levels. Newell (1982) first argued that cognitive processes can be understood by their levels of complexity through orders of chronometric structure, a relationship of behaviors interconnected by time scales. According to Newell, “A system level is a property of nature, and not just something in the head of the observer” (Newell, 1994, p.118). In this ontological structure, Newell defined many individual levels, ordered into groups, or bands.

<table>
<thead>
<tr>
<th>System</th>
<th>World (theory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Task</td>
<td></td>
</tr>
<tr>
<td>Unit Operations</td>
<td>Cognitive Band</td>
</tr>
<tr>
<td>Deliberate act</td>
<td></td>
</tr>
<tr>
<td>Neural circuit</td>
<td>Biological Band</td>
</tr>
<tr>
<td>Neuron</td>
<td></td>
</tr>
<tr>
<td>Organelle</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1.: Newell’s System Levels.

For Newell, a level defines those behaviors that occur encapsulated by the same order of magnitude for a given feature (e.g., time). Newell also used the term Band to refer to those levels that can be discussed using the same vocabulary without constraint. The Biological Band (Table 1.1), for example, subsumes activity produced by an organelle, a neuron, a neural circuit; all behaviors that use a similar vocabulary in the biological sciences. However, once we start talking about deliberate acts,
units of uninterrupted operation, units of a task, etc., we have difficulty using the same terms and must switch to another (Cognitive) band with more appropriate vocabulary; so on and so forth through orders of magnitude for Rational thought of an independent agent, and Social activity between multiple agents.

The System Level paradigm is based on modeling digital processors, as used to define and describe increasing levels of complexity for components, sub-systems, and systems, without having to understand and discuss every detail at once. Newell uses the engineered system levels in computers as evidence that system levels can physically exist (Newell, 1982, p.11-12, see also Causey, 1977). Newell’s ontological claim, based on Simon’s evolutionary work (1969), is that the brain is necessarily built this way, and “...any instantiation of a level can be used to create any instantiation of the next higher level” (Newell, 1982, p.10).

Overall, the goal of unification makes more sense if the cognitive system “exists” as a systems level as Newell hypothesizes. These levels are not imaginary, but a medium. Newell states “The medium is realized by state-like properties of matter, which remain passive until acted upon by the level’s components” (Newell, 1982, p.16). If so, it should be possible to describe it in computational terms. That is, it should be possible to create a formal system, in the form of a cognitive architecture, that accurately describes the human cognitive system. Cognitive models or architectures, then, are theories about the Cognitive level of mental activity, and the computational languages that implement them must be necessarily sufficient languages for theorizing about the brain. If the cognitive system exists only as an epistemology, that is, the language of computation is a useful but ultimately misleading lens through which to understand the brain, then the goals of formalization and unification are less compelling.
1.4 The Common Model of Cognition

Well known Cognitive Architectures, such as ACT-R (Anderson and Schooler, 1991), Soar (Laird, 2008), and more recently Sigma (Rosenbloom, 2013), all stem from a blended history of thought, and are based on very common terminology and micro-theoretic structures. In Laird et al. (2017), the Standard Model of Cognition — now Common Model of Cognition (Laird and Mohan, 2018) — was positioned as a community effort to bring these architectures under a single banner by adhering to a set of assumptions on how the human mind works. The Common Model of Cognition (CMC) is an attempt to discover areas of consensus, or general norms, between the most prolific cognitive architectures, using ACT-R, Soar, and Sigma as exemplars. For a more detailed introduction to various classes of cognitive architectures, see Appendix B.

![Diagram of the Standard Model](image)

**Fig. 1.1.:** The Structure of the Standard Model, adapted from (Laird et al., 2017b)

It is a generalized framework that describes the commonalities between these popular cognitive architectures, and does a good job describing characteristics and qualitative measurements. These assumptions are based on our current understanding of four general areas: (i) structure and processing; (ii) memory and content; (iii) learning; and (iv) interacting with the environment through perception.
and motor skills. Despite these areas of commonality, we also see that the underlying design principles are indeed different, and take on the respective patterns of the Cognitive Science disciplines from which they were derived. For example, ACT-R micro-theories are heavily based on psychology theory, whereas Soar is heavily based on artificial intelligence theory. A more thorough review of the CMC is covered in Appendix C.

The CMC community literature describes how various architectures are related to the Common Model, or aspects of human cognition that should be added to the common model research agenda; these efforts, to date, have been largely theoretical, with ample qualitative review, and some quantitative benchmark comparison within the confines of their respective research programs (Appendix C.2).

CMC architectures use similar conceptual constructs, but all use different syntax, different underlying data structures, and support different levels of functionality. Yet, despite our doubts, the existence of a common model begs the very question: how much of each architecture can be abstracted away from the implementation? We should be able to use what we already know about the human mind, and the design patterns the cognitive modeling community agrees upon, to explicitly define areas of redundancy and reusability; to *carve nature at its joints*.

Can the Common Model be used as the basis for formalization?

### 1.5 The Goal of Unification

Cognitive Architectures have been used to test theories stemming from cognitive psychology, and successfully applied to multiple practical domains, including adversarial game theory (Gonzalez et al., 2017), real-time strategy gaming (Peters et al., 2018b), human-machine teaming (Maymi and Thomson, 2018), and cyber security (Jones et al., 2019). Most recently, the Common Model community has
explored new avenues and paradigms for modeling more complex aspects of cognition, such as emotion (West and Young 2017), meta-cognition (Kralik et al. 2018), and holographic memory representation (Arora et al. 2018). Yet, despite these advancements in cognitive modeling, the community suffers from a proliferation of various similar-yet-different modeling architectures, resulting in a plethora of disparate modeling tools and frameworks.

In “You can’t play 20 Questions with Nature and Win”, Newell (1973) argued against the behaviorist approach to creating disparate theories of cognition through disjoint experiments with subjects generating stimulus-response models, and the community’s habit of producing what he termed micro theories, each producing what Newell thought of as small pieces of a larger tapestry, with little effort to stitch them together. Micro-theories, in this context, are represented in and tested by micro-models; models that instantiate psychology theories, and show how specific mechanisms and specific representational structures can describe specific cognitive psychology experimental results. Cognitive architectures, in general, serve to weave micro-theories for specific mechanisms and structures into a unified whole; they are, in essence, in the service of unification.

When we talk about cognitive micro-theories, and the cognitive architectures that contain them, we are (so to speak) talking about differentiating the trees from the forest, respectively. The task of creating a cognitive architecture is one of combining the different micro models that need to be unified into an architecture to explain how they work together. Some architectures even contain and implement multiple competing micro-theories that are capable of explaining and predicting some or all of a specific phenomenon produced by experimental results.

Different cognitive architectures may, at their core, implement some of the same micro-theories, and over time are meant to evolve into one architecture (Lakatos
This form of unification is the essence of a Lakatosian program of unification; a research program that contains a hard core of protected normative theories that have been exhaustively tested, as well as a periphery of competing theories that depend on the standardized core during their time of refinement (Figure 1.2).

The discipline of cognitive modeling has not yet moved to this second stage of unification. Instead, there are still multiple architectures and many independent micro models. Without a rigorous, universally accepted description, multiple micro models implemented in disparate cognitive architectures rely on a weak predictive nature driven by self-reported compliance; therefore, they cannot be formally compared and tested against each other to move toward unification.

There are different aspects of the current representation that precludes and ultimately prevents formalization: the architectures that implement micro models are developed using different computer languages and cannot be cross-compiled or integrated; the model files that demonstrate specific micro-theories run using task models developed by the same designers of the architecture, for that architecture, and subject to bias; model files developed for that task-model cannot be run on other architectures due to conflicts of syntax and semantics; many architectures do
not expose their core functionality through a standard interface, forcing inquiring modelers to learn to use their programming language, using their model definition syntax; and some architectures are (for lack of a better description) immature, incomplete, or too difficult for new researchers to pick up without significant effort overcoming technical hurdles unrelated to the cognitive theories of interest.

1.6 Model-Driven Development

Large engineering projects require conceptual designs that are correct by design, with specifics in realization pushed to lower levels of implementation, that end up being correct by construction. The practice of Model-Driven Engineering, also called Model-Driven Development (MDD), is a development paradigm that excels at top-down problem solving by first specifying the structure and intent of a system, such that lower levels of abstraction can be implemented in different forms and environments, and yet ensures they conform to the software meta-model (Figure 1.3). Modern MDD in practice uses a combination of software modeling languages, software model validation, model-to-model transformations, model-to-text transformations, and in some cases, model interpretation and execution (Rodrigues Da Silva, 2015).

The Meta Object Facility (MOF), created by the Object Management Group (published as ISO/IEC:19508), is a standard to guide the creation of software meta-models, their concepts, attributes, and relationships, in a way that can be described, encoded, and transformed between architectures, and yet remain formally validated (Object Management Group, 2014). The MOF standard defines how any formally represented Meta-Model can be used to prescribe the higher-level ontology and taxonomy of constituent sub-systems, without specifying exactly how they are implemented or realized, and can be used as the basis of any MDD project.
It is important to point out that references to a type of formal system in the MDD context, is one that involves reasoning about the formality of the system itself, as opposed to the form of reasoning that manifests within the system that is implemented formally. In [Hofstadter 1999] we find examples of this version of formality as a fractal system that manifests greater levels of complexity at geometric scales (as seen in Newell’s system levels); every level is more general or specific than the one that preceded or succeeded it (such as a logarithmic scale of time), and we hope to capture a single band of this world system (i.e. Newell’s Cognitive Level) through a formal Meta-Model, using MDD.

MDD methodology applied to cognitive modeling can ensure resulting cognitive agents conform to a standardized consensus of both function and form, while remaining falsifiable at different levels of abstraction. This is important, because falsification in this context allows researchers to formally evaluate whether a meta-model is correct or not, i.e., whether a specific meta-model of the Common
Model is correctly capturing what has been proposed, and whether the mappings to the Cognitive Architectures are correct, or not, in true Popperian form.

The proposed approach to modeling also supports a Lakatosian research program, that is, a way for Cognitive Science to move forward by producing emergent behavior, without having to disprove or correct existing micro-theories of cognition that are already hard-coded in many current architectures.

Cognitive Modeling using MDD methodology may help to draw connections between high-level abstractions and low-level implementation to analytically identify issues that cross the boundaries of world views; resulting models then support stronger statements about how the human brain works through convergence of these world theories. A systematic evaluation of the design patterns of cognitive processes in architectures that are only seemingly disparate, and replicating those processes through a process of abstraction and refinement, not only lends credence to the creators of these architectures, but extends to and empowers researchers seeking to bridge the common ground. An architecture or agent developed by or based on MDD methodology still supports these efforts while ensuring compliance with the CMC, and may help to identify areas of divergence between architectures by viewing them through different epistemic lenses.

Furthermore, the Meta-Objects used to describe CMC allows modelers to specify the standard characteristics and requirements of perceptual interfaces, forming a model-generated middle-ware capable of translating between architectures and environments. This is important, as the application of MDD to evaluation also allows modelers to extend beyond the Common Model by introducing a general model of the Environment (to be refined via mappings if need be); experiments (and their metrics) can therefore be represented at the same level(s) of abstraction as the models that interface with them. The value of this approach gives confidence that
cognitive agents, their interface points, and the standardized environment stimuli all inform the prospective modeler on the applicability of various architectures to various problem domains. This perspective may very well save development efforts by allowing modelers to concentrate on those variables most applicable to the research problem at hand, and confidently ignore the aspects that are generated through a tested and testable template of models common to all architectures, regardless of the platform of implementation, or target environment.

1.7 Research Problem

The research addressed by this thesis is based on the goal to understand how to apply formal model definitions to cognitive models based on what we already understand about the human mind; to generate, test and evaluate the performance of virtual agents through various stages of model-driven development and growth; and to draw these inferences through time and space, both within and without each model.

The current state of the art, by extension, does not support comparison of these models against a standard of environments or fitness functions. We face an issue of architectural disparity, a lack of uniform test metrics or environments, and the many difficulties experienced when trying to perform direct translations between model files.

The research problem addressed by this thesis is the feasibility of formalizing the Common Model of Cognition to define, test, compare, and validate standard micro-theories and functions across Cognitive Architectures.

This thesis explores this issue through the creation of a formal Meta-Model of Cognition, and the generation of architecture-agnostic interfaces, to test and validate multiple architectures on common tasks according to a Common Model standard. We
use a Model-Driven Development (MDD) approach to factor, abstract, and template meta-model interfaces to demonstrate the common features of model syntax common to architectures based on a Common Model of Cognition (CMC), and to demarcate those aspects that require extra programming for the target environment or type of problem to be solved.

In Figure 1.4 we see an example matrix of Newell’s ontological system levels intersecting the different epistemological perspectives afforded by the more formal Meta-Object Facility.

<table>
<thead>
<tr>
<th>MOF / System</th>
<th>M3 (Vocabulary)</th>
<th>M2 (Specification)</th>
<th>M1 (Architecture)</th>
<th>M0 (Instantiation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Social</td>
<td>Meta-Model of Macro-Cognition</td>
<td>Common Model of Macro-Cognition</td>
<td>Multi-Agent Systems</td>
<td>Real-Time Interaction</td>
</tr>
<tr>
<td>Rational</td>
<td>Meta-Model of Rationality</td>
<td>Common Model of Rationality</td>
<td>Problem Solving</td>
<td>Interruptible Sequence</td>
</tr>
<tr>
<td>Implementation</td>
<td>Meta-Model of Implementation</td>
<td>Common Model of Implementation</td>
<td>Hard-/Soft- /Wetware</td>
<td>My Computer Your Brain</td>
</tr>
</tbody>
</table>

Fig. 1.4.: Meta-Object Facility Layers for Newell’s System Levels

The Cognitive row deals with questions of how to go from Meta-Model definition, to the Common Model specification, to specific architectures, and finally to an instance of a model. The Architecture column deals with questions of how to design Macro-Cognitive, Rational, Cognitive, and Implementation architectures manifest at different time scales with varying degrees of complexity. The Meta-Object Levels provide graduating levels of abstraction and specificity, in this example dividing up the vocabulary used to discuss a particular level, the specification of agreed-upon standards or rules, the design of a system based on these rules, and finally the instantiation of a design in a tangible application. As most readers are probably unfamiliar with the MOF layers, and how they can be
applied in this context, we will come back to this conceptual framework and discuss the approach in greater detail in Chapter 2 as well as a detailed application of MOF through MDD in Chapter 4.

A formal definition and meta-model of the core components of the CMC can be used to represent and derive a formally-generated and extensible framework, that supports directly identifying and validating their requirements at design time, as well as during run-time. This meta-model can then be used for the generation of a middleware system that couples these agents and environments across layers of abstraction afforded by a top-down approach to model design. A middleware platform, based on a Meta-Model of Cognition, is positioned to support the generation and capture of experimental replay data to test the theories of the CMC in empirical fashion. A generated and refined research platform for the CMC is not designed to replace any of the features of Cognitive Architectures, but rather to make a variety of cognitive architectures and existing virtual agent environments accessible to cognitive modelers. The main idea behind such a platform, then, is to find measurements that everybody can reproduce regardless of modeling background or experience.

This approach leads to a number of more specific research questions that will be addressed in the remainder of this thesis:

- How much of the Common Model of Cognition can be defined formally?
- How much of an architecture-specific model file can be generated from a formal Meta-Model of Cognition?
- What is the effect of applying a formal Meta-Model of Cognition to specific components?
- What kind of predictive power is in a Meta-Model of Cognition?
• Can a Meta-Model define constraints to validate or invalidate cognitive architectures?

• Does a formal Meta-Model of Cognition specify environment interfaces that work on all CMC architectures?

Answering these questions informs researchers on the viability of generating and using quantitative and qualitative information to systematically evaluate different cognitive architectures that purport to implement some or all of the characteristics and requirements of a Common Model of Cognition. To address and answer these questions, then, is to inform the larger research question at hand: can the code define the theory?

1.8 Moving Forward

The remainder of this thesis describes a formal Meta-Modeling methodology, framework, and tool chain to demonstrate the theoretical and practical feasibility of using a formal model of the Common Model of Cognition to formally define the Meta-Model of Cognition, to template and generate those features and modules inherent in and inherited by the CMC architectures, an application of using these model templates to systematically generate the required interfaces used for comparing these implementations in various target domains, and, the effects and outcomes of measuring user agents from different cognitive architectures performing the same task model.

This research in this thesis produced multiple contributions to knowledge, methodology, and practical artifacts that can be reused. The application of methodology and experimental artifacts were used to answer our research questions about the CMC as a Lakatosian research programme, and a formal solution for
considering a standardized CMC theory as code, addressing our philosophical questions about the consideration for code generated from theory, ultimately producing novel contributions to knowledge in the field of Cognitive Science (Table 1.2).

<table>
<thead>
<tr>
<th>Description</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formalized the CMC as a MOF-compliant Meta-model that encapsulates the Technical and Non-technical requirements of the CMC Assumptions.</td>
<td>4.5</td>
</tr>
<tr>
<td>Mapped between MOF and Newell’s System Levels that can be used to guide future integration and transformation between a Meta-Model of Cognition, and formal models that reside at the Knowledge and Implementation levels.</td>
<td>1.7, 3.1, 4.3.1</td>
</tr>
<tr>
<td>Mapped between MOF levels and the Cognitive Architectures that conform to the CMC to guide transformations between Cognitive Meta-Models and Cognitive Models.</td>
<td>4.2.1, 4.3.1, 4.6</td>
</tr>
<tr>
<td>Created a Meta-Model representation of the abstract classes, attributes, operations, and relations described by the CMC, to aid modelers in understanding the conceptual underpinnings of all CMC architectures.</td>
<td>4.5.3</td>
</tr>
<tr>
<td>Applied Meta-Modeling methodology to identify, extract, group, and filter the logical and functional elements required to define a Common Meta-Model of Cognition according to the CMC and related extant literature.</td>
<td>4.5.3</td>
</tr>
<tr>
<td>Applied MDD methodology to the Meta-Model of Cognition to transform between graphical UML and meta-data XMI interchange formats.</td>
<td>4.5.6</td>
</tr>
<tr>
<td>Applied Object Constraints to the Common Meta-Model of Cognition to specify common invariants, attribute values, and pre- and post-operator conditions.</td>
<td>4.2.4, 4.5.4</td>
</tr>
<tr>
<td>Design-time validation of CMC Meta-Model relationships using OCL expressions in the Model Transformation Language used to transform the CMC Meta-Model into a Platform-Specific-Model.</td>
<td>4.10, 5.2.1</td>
</tr>
<tr>
<td>Run-Time validation of CMC Meta-Model invariants using operators and attributes defined by the CMC Meta-Model.</td>
<td>5.3, 5.4</td>
</tr>
<tr>
<td>Applied MDD transformations to the CMC Meta-Model to create an abstract factory that conforms to and expects a standard interface for interacting with architectures based on the CMC.</td>
<td>4.2.2, 4.3.2, 4.5.6</td>
</tr>
<tr>
<td>Contributed to a Lakatosian research programme by capturing the essence of the theoretical commitments as a formal standard without prescribing unnecessary function representation, and incrementally evolving the standard representation through verification of peripheral theories as code.</td>
<td>6.1</td>
</tr>
</tbody>
</table>

Table 1.2.: Contributions to Knowledge and Methodology

The application of MDD — a formal software engineering methodology — to the Common Model of Cognition, was an unprecedented application of MDD specific to the CMC, producing contributions to methodology. The firmware source code
generated from the MOF-compliant Meta-Model, the MTL-generated templates, and interfaces mapped to ACT-R and Soar (Table 1.3), are all contributions to the Practice of Cognitive Modeling, and freely accessible to those who wish to use (or reproduce) them.

<table>
<thead>
<tr>
<th>Description</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>A list of technical Requirements for Architectures that are, or strive to be</td>
<td>4.5.3</td>
</tr>
<tr>
<td>based on the CMC; mapped back to the original CMC Assumptions and Categories.</td>
<td></td>
</tr>
<tr>
<td>An example UML Meta-Model of Cognition based on the entities defined by CMC</td>
<td>4.5.5</td>
</tr>
<tr>
<td>Requirements.</td>
<td></td>
</tr>
<tr>
<td>A prototype graphical Meta-Modeling workbench based on Sirius that can be</td>
<td>G.2, Metaverse</td>
</tr>
<tr>
<td>used to edit the Meta-Model, create cognitive models based on the meta-model</td>
<td>(Peters, 2021)</td>
</tr>
<tr>
<td>of cognition, and generate model templates based on pre-defined transformations.</td>
<td></td>
</tr>
<tr>
<td>Sample MTL templates that use the CMC Meta-Model to generate ACT-R and</td>
<td>4.5.6, Metaverse</td>
</tr>
<tr>
<td>Soar model templates that conform to the CMC.</td>
<td>(Peters, 2021)</td>
</tr>
<tr>
<td>An abstract factory interface generated in PyExcore from EMF definitions that</td>
<td>6.2.1</td>
</tr>
<tr>
<td>ensures all attributes, values, and relationships between Meta-Model objects</td>
<td></td>
</tr>
<tr>
<td>remain compliant with the EMF constraints throughout model run-time.</td>
<td></td>
</tr>
<tr>
<td>Mapped the modules and operators defined in the factory interfaces generated</td>
<td>5.2.1, 5.2.2</td>
</tr>
<tr>
<td>from the CMC Meta-model to their equivalent functional modules and functions</td>
<td></td>
</tr>
<tr>
<td>in ACT-R and Soar.</td>
<td></td>
</tr>
<tr>
<td>A custom Gym Environment wrapper that can be used to interface agents</td>
<td>4.6.5, Metaverse</td>
</tr>
<tr>
<td>originally written for the Gym ecosystems with the StarCraft II Real-Time</td>
<td>(Peters, 2021)</td>
</tr>
<tr>
<td>Strategy game.</td>
<td></td>
</tr>
<tr>
<td>Sample cognitive models in ACT-R and Soar to demonstrate the evaluation of</td>
<td>5.3, Metaverse</td>
</tr>
<tr>
<td>core and peripheral cognitive modules across architectures, using different</td>
<td>(Peters, 2021)</td>
</tr>
<tr>
<td>virtual environments developed independently of any cognitive architecture.</td>
<td></td>
</tr>
<tr>
<td>A list of CMC Requirements validated against original CMC assumptions.</td>
<td>6.2.2</td>
</tr>
<tr>
<td>A list of CMC assumptions that could be validated and are candidates to further</td>
<td>6.2.3</td>
</tr>
<tr>
<td>refinement and specification across all tested cognitive architectures.</td>
<td></td>
</tr>
<tr>
<td>A list of CMC assumptions that conflict with implementation or mapping across</td>
<td>5.2.2</td>
</tr>
<tr>
<td>the cognitive architectures and require further deconfliction with tested</td>
<td></td>
</tr>
<tr>
<td>cognitive architectures.</td>
<td></td>
</tr>
<tr>
<td>A list of suggested changes that tested cognitive architectures could</td>
<td>6.2.4</td>
</tr>
<tr>
<td>implement in order to better align with the Requirements of the CMC Meta-Model.</td>
<td></td>
</tr>
<tr>
<td>A refined Meta-Model of Cognition based on interface norms and requirements</td>
<td>6.2.3, F.1</td>
</tr>
<tr>
<td>that have been validated across tested architectures.</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1.3.: Contributions to Practice**

The fundamental issues and discussions are more thoroughly addressed throughout the remainder of this thesis as follows: Chapter 2 introduces the Common Model of Cognition, current issues related to task model design and comparison, and solutions for evaluating the agents that purport to model them. Chapter 3 identifies and explores the main research problems inherent in the state of the art, the questions
they generate, and motivation for this research. Chapter 4 covers the methodology of this research, including design patterns, experimental platform, and test criteria used to explore the research questions. Chapter 5 presents a comprehensive summary of the results generated by the experimental design and testing framework, and the notable patterns of interest. Chapter 6 sums up this research through discussion and commentary on how these results answer the preliminary research questions.
2. THEORETICAL BACKGROUND AND STATE OF THE ART

2.1 Introduction

This Chapter presents a more recent review of literature dealing with the issues that have arisen out of previous attempts to merge cognitive architectures under a single banner. Many of the issues associated with defining a unified model of cognition can be traced to recurring patterns. First, cognitive modelers are missing the point of cognitive modeling, and try to escape from the hassles of learning to code in “cognitive speak” in favour of easier, but less precise, languages. One common approach to this problem is by using high level languages that compile to cognitive model code; this approach has met with some success to generate task-specific models, and some of this may be applicable to a Common Meta-Model, like the CMC, as well. However, this approach also forces the modeler to make unwanted theoretical commitments.

Another recurring issue is a tension between designing models from a high-level, or defining the problem before implementing the micro-theories, versus starting with the micro-theories that support cognition, and then writing the abstraction on top of it. A less common, but highly effective approach, is to use hierarchical systems that decouple the relationship between macro- and micro-cognitive applications.

Last, but not least, modelers frequently conflate system levels and research perspectives while comparing cognitive architectures and task models. Docking studies attempting to compare and contrast equivalent task models across different cognitive architectures, or near-identical implementations across different ask
models, frequently run into significant problems due to disparate implementation of micro-theories, or incompatibility with task model definitions.

Architecture-specific implementation is a natural consequence of taking a stance on any theoretical position, but the more assumptions made in the implementation, the less a formal model can generalize across other architectures. Model-Driven Design methodology has been successfully applied to a tangential issue of comparing Meta-Cognition across agent architectures; specifically using a Meta-model of Meta-Cognition, with an example architecture that relies on it to define higher-order introspection. Could this approach be applied to comparing Cognitive Architectures without getting trapped by the recurring usual issues?

The rest of the Chapter provides the theoretical basis and discussion around these issues and various approaches to wrestle with them.

2.2 Toward a Common Model of Cognition

The Standard Model of the Mind (Laird et al. 2017b), later renamed to the Common Model of Cognition (CMC), organizes and describes in human readable terms a series of abstract components that encapsulates the common views held by a large body of cognitive modeling researchers; the initial attempt is grounded upon three cognitive architectures: ACT-R, Soar, and Sigma.

The CMC is organized into conceptually distinct functional areas (or regions of interest) of the human brain (Figure 2.1): structure and processing; memory and content; learning; and perception and motor. Each of these functional areas of cognition describes foundational human behaviours observed from years of collected data based on the cumulative research of the three architectures, while prescribing the most abstract design principles inherent across all of them.
Structure and processing is based on the assumption of some form of cognitive cycle that acts upon a (roughly) 50ms clock rate, with Working Memory acting as the central hub of activity, and a production system with only peripheral access to modular buffers or interfaces. The choice of inter-module serial and parallel processing is only loosely defined, with intra-module processing completely undefined.

Memory and content of both symbolic and sub-symbolic representations are governed by metadata that captures the historical and stochastic properties of storage and recall, since many of these are also fundamental to learning and forgetting. The three memory systems defined by CMC are working, procedural, and declarative. Working memory serves as a global work space for temporary storage and manipulation, procedural memory stores the condition-action rules, and declarative memory serves as a form of long term storage.

Learning, according to the CMC, includes the creation and tuning of new symbol structures and metadata associated with the various forms of memory. The CMC holds that all LTM is learnable, based on an incremental form of feedback. Procedural memories can be tuned based on perceived utility and recall of individual components.
Perception and motor functions are described as basic input/output interfaces that turn signals into processable representations/action respectively. The CMC also offers the affordance of a direct signal pathway from working memory to perception based on theory of expectation. There is no consensus as to how symbol structures and associated signals are converted into their respective effectors or actuators in situated and embodied cognition.

The Common Model suggests a plausible standard of functional characteristics that most architectures can at least partially conform to (see Appendix C for additional details on each). There are, however, still lingering issues with how to formalize and map cognitive architectures onto the levels and categories of the Common Model.

Table 2.1 presents an analysis of the various approaches used to compare and map existing cognitive architectures between and within the Common Model. Each approach can be considered one of: transformation of one architecture to another; mapping one system on to another; addition of functionality directly to the CMC; consideration for extra features adjacent to (or outside) of the CMC; or suggesting a methodology for comparing and validating cognitive task models written for one of the architectures that contributed to the CMC.

For each of the approaches reviewed, I have identified them according to their adherence to a (F)ormalized method (such as Meta-Object Facility) that can be replicated by a third party researcher, use of a (M)odel-driven approach (such as Model-Driven Design) to support generalization or specification of a model, and adherence to the functionality specified to (and constrained by) Newell’s (C)oognitive System Level. For a comprehensive summary of each approach, see Appendix C.

The majority of current issues stem from the understanding of why Production Systems are used in the first place, the difficulty arising out of programming them, and
differentiating between the ontological relationships and epistemological perspectives that are too often recited to justify (or promote) one method over another.

The following sections discuss the fundamental issues that each of these approaches is attempting to solve, such as different ways of programming cognitive models, using different paradigms of model design, and dealing with transitions between system levels.

First, I revisit and recast these issues against Newell’s System Levels to show that basing an architecture on strong claims across multiple levels not only fails to leverage the strengths of natural hierarchies in engineering (per Simon), but also limits novel progress in integrated cognition across system levels by forcing researchers to adopt these positions by default.
Next, I will discuss how an emphasis on abstracting away from production systems, and concentrating on compiling high-level languages into model code, may not be the right approach to convergence.

Last, I will reflect on how the different methods such as docking studies contribute to the goal of task model analysis and comparing across architectures, with suggestions for improvement based on the findings of this thesis.

2.3 Crossing System Levels

The issue of system levels occurs when describing and analyzing levels of complexity in models of human cognition. It comes as no surprise that we are keen on separation of levels used to describe and evaluate a model; after all, this helps us to describe and discuss our models using different terminology when dealing with different levels of system complexity, or at different time scales. However, we must be careful to make the distinction between perspectives of analysis, and system levels of complexity.

2.3.1 Issues with Level Specification

Newell was very clear on the use of System Levels, and by doing so he described an ontology; these are not a metaphor – these levels are real, are inter-related, and manifest at geometric scales of time. This is already observed in computer systems engineering, where system levels describe integrated circuits, digital logic, computer architecture, and the languages we use to describe and manipulate them. Newell did make the distinction between “strong” and “leaky” levels with tight or loose boundaries, just as observed in the human brain with organelles, neurons, distal connections, and functional regions. Hence, system levels can be viewed as distinct,
physical levels of reality that produce increasing complexity of behavior at larger
time scales. Newell’s use of *Bands*, are like languages used to discuss these levels.
We use Biological terms at the lowest level, and when the vocabulary starts to
become strained, then we switch to Cognitive terminology for higher level concepts,
then on to the Rational, and finally Social band.

The CMC discusses its relation to the neural, cognitive, and rational bands; it is
clearly situated at the *Cognitive* level, and does not make any strong claims about
influences from, or on, adjacent bands. The lower Biological level supports the
common model, but is unspecified in medium, implementation, or substrate. The
Cognitive level is the seat of the CMC, and deals in the components and interplay of
specialized constructs that support and represent our task models. The higher,
Rational band, is the Knowledge level described in and by most computer programs,
and is the vocabulary of independent problem solving. The highest Social/Macro
band is the realm of what most people think of as long-term strategic behavior, and
the interplay between multiple independent agents with a shared context and
environment.

Consider the Problem Space Computational Model (PSCM), that underpins the
cognitive cycle in the Soar architecture. The PSCM is built upon and utilizes the
Production System, memories, and buffers described in the CMC, but also makes *very*
strong assumptions about problem space, states, goals, and the structure of attribute-
value pairs in working memory elements. According to Newell, “The Soar architecture
is built to accomplish these functions directly...it may fairly be characterized as a
*problem-space architecture*” ([Newell](1990) p. 163), thereby making strong assumptions
about, and directly influencing, Knowledge system levels that are above the Cognitive
Band.
The issue of system levels is even more obvious when people ask for different languages for the ease of working with each one them. We see this in architectures and vocabularies such as Nengo (Biological), ACT-R (Cognitive), and SGOMS (Macro). We also see this in projects using high level languages to describe one band (e.g., Knowledge in Herbal and JASS), which can then be translated or compiled into lower level bands and systems (ACT-R, Soar), usually resulting in the difficult task of encoding and decoding assumptions in meta-data to compile to lower levels. Some modelling toolkits do this by design, and have their various reasons, but in most cases are catering to strong assumptions that result in a highly constrained methodology.

2.3.2 Model-Driven Design

In computer software engineering more generally, the effort to abstract away from platform-specific code originated with varying degrees of optimism, and like many other “Grand Unified” technical panaceas was met with a dose of reality curtailed by a decade of practice. In (Thomas, 2004) we see a sobering review of Model-Driven Engineering applied to various platforms and technology stacks, and the struggles associated with linking platforms using APIs generated from meta-models. Thomas (2004) later concludes that meta-models and corresponding MDDs have their place as useful tools when and where applied properly, so long as modelers have access to a Domain Specific Language (DSL) that can be leveraged by their domain expertise. In the case of the CMC, a DSL consists of the terms and relationships that define, connect, and constraint cognitive components.

Speaking specifically with respect to intelligent agent design, Caro et al. (2014) introduce Metamodel Supporting Metacognition (MISM), a general model designed to support and facilitate self-monitoring and awareness in traditional intelligent agents. The MISM system is based on the MOF standard, and applies a layering approach for
conceptual instantiation between the conceptual Metamodel, domain-specific Model, and finally realized through the problem-specific User Model (Figure 2.2).

Caro et al. later built on this work with a novel application of MDD towards the development of MetaThink, a visual DSL (or DSVL) toolkit and corresponding language M++ to support model developers using the more abstract MISM system Caro et al. (2015). Finally, in (Caro et al. 2018), they present an application of MetaThink to developing a model for CARINA, a cognitive architecture purpose-built to model meta-cognition in intelligent agents.

Caro et al. later built on this work with a novel application of MDD towards the development of MetaThink, a visual DSL (or DSVL) toolkit and corresponding language M++ to support model developers using the more abstract MISM system Caro et al. (2015). Finally, in (Caro et al. 2018), they present an application of MetaThink to developing a model for CARINA, a cognitive architecture purpose-built to model meta-cognition in intelligent agents.

The approach taken by Caro et al. demonstrates a promising application of MDD to the field of Cognitive Science in general, and especially the Cognitive Models represented by the CMC. For example, they use methodical analysis to derive the tasks and processes for meta-cognition in intelligent agents. As described in (Caro et al. 2014), the seven step process consists of: identifying sources of information in the metacognitive models; classification of types of models; extraction of concepts; selection of the most common concepts; classification of concepts; identification of
relationships between concepts; and finally creation of the metamodel using derived
concepts and relationships.

There are two differences between the MDD processes used for MISM/CARINA,
and the process used by the CMC. The first is that while this process has already
been carried out to some degree by Laird et al. (2017a), Jeronimo et al. (2018) take
it one step further by defining a Metacore package (based on the MOF standard)
that is used as the metamodel for all further metamodel instantiations, and has been
shown to be mathematically verifiable through formal specification for all instantiated
models.

The second major difference lies in how far the MISM metamodel is used in
application and realization. In Caro et al. (2018) the framework implements all
levels of the MOF standard, specifically: the Meta-Meta-Model (M3), realized
through UML; the Meta-Model (M2) realized as MISM; the Model (M1) realized in
the CARINA meta-cognitive architecture; and finally, User Models (M0) as realized
through instantiated meta-cognitive models within specific real-world intelligent
cognitive agents. The CMC, when compared to a standard such as MOF, prescribes
the Meta-Model (M2) without conforming to any standardized Meta-Meta-Model
(M3), and is realized through the Model (M1) layer as ACT-R, Soar, or Sigma.
Since there is no traceability between CMC models, specific User Model (M0)
instances of any of these architectures may or may not conform to the CMC in
practice.

Last, while the work in Caro et al. (2014) is focused more on meta-cognitive
processes, the CMC in Laird et al. (2017b) is focused on the underlying cognitive
processes that introspective meta-cognitive processes are supposed to support,
without making any further claims towards meta-cognitive processes, and therefore
could directly benefit from the same methodology of design through formal specification.

At the time of this writing, only Caro et al. (2018) has proposed an approach to transformation, mapping, or a platform that can demonstrate a formalized, verifiable approach that uses model-driven methodology within the Cognitive Level; however, their approach is specifically additive for augmenting existing Cognitive Architectures with Meta-Cognitive capability, an aspect that has not yet been addressed by the CMC.

The methodology and solution in this thesis is inspired by the recent work of Caro et al. (2018), by creating a tangible application of the MOF standard for a formalized CMC Meta-model in support of an architecture-agnostic DSL, and generation of corresponding APIs.

This presents an opportunity for the cognitive modeling community to design similar formal modeling specification for the CMC so that all derived architectures and user models can be compared across levels of abstraction. This thesis describes how to apply this methodology, and provides a specification for mapping within the cognitive system level, and a loose coupling across levels.

### 2.4 Programming Cognition

The issues of modeling cognition programmatically, and how to do it, stem from the creation of scripting languages for cognitive production systems, and the emphasis on abstracting away from the architectures that support them.
2.4.1 Difficulty with Programming

The point of using cognitive architectures isn’t to solve problems easily or as well as possible, but to model human behavior and prove theories about human behavior. This makes sense on paper, but in practice it is all too easy to side with another more familiar way of doing things (such as coding imperatively), instead of going through the pains of struggling with a new way of thinking, or adopting a new paradigm (such as declaratively).

For example, we could make a comparison of the evolution of computer programming languages to the evolution of cognitive model scripts. First- and second-generation computer languages that deal with hardware design, digital logic, or low-level operators (such as Verilog, parallel processing in C, etc.) are generally considered more difficult to use and master, yet still very efficient for their domain as they are purpose-built for a specific hardware architecture, and force you into a certain way of thinking and writing that is very different than higher-level programming languages. Third- and fourth-generation languages (for example C/C++ and Java) were designed to compile human-readable text into these low level languages, with the primary goals of code reuse, and making the programmer more productive without making such strong assumptions about the underlying hardware.

Cognitive modeling languages are similar to early generation grammars as they also operate on elementary components, and yet sometimes cater to various levels of abstraction. If we are using a specific computer programming language (such as C or Lisp), rather than a cognitive modeling language to design the components of an architecture, we are only doing so to solve a really hard problem or hard-code a micro-theory of cognition, and need to know where things are going on. In general, we want to be programming in the language of human cognition, not another language that
was designed for other domains such as video games, web applications, etc., so that we can remain independent of the influences and assumptions made when compiling them into machine-readable format.

The second aspect of this issue of confusion is around the assumption of Production Systems as the \textit{key} to modeling human cognition, and whether or not they are necessary to do so. Production Systems were designed to model the same constraints observed in human thought, and by extension require a domain-specific language that facilitates this level of abstraction in a way that takes advantage of stochastic rule choices.

Humans don’t write cognitive models naturally, and try to remove the difficulty or go around it by writing higher level languages that can accomplish the same thing. For example, we write multi-threaded applications using libraries because thinking as parallel hardware (in multiple timelines) is counter-intuitive. Yet, if your goal is to understand how the hardware works, writing in a language like Java or Python is counter-productive.

In Cognitive Science, we don’t gain anything by using design patterns to compile our code at a higher level relative to lower level without first understanding why those patterns exist, and which assumptions are made in using them. The point is to question “can I actually reason using these Production Systems”, and “are they sufficient to produce human-like behaviors?”. If this is not the point, then using high level languages to make it easier to compile into ACT-R or Soar simply avoids the important questions altogether. However, there are ways to make this task a little easier, as discussed in the next section.
2.4.2 Model Compilers

The Herbal development environment (Cohen et al., 2005) was an attempt to address this issue by providing a high level language, compiler, and a theory for generating explanations for models created in Soar. The Herbal high-level language is based on the same Problem Space Computational Model (PSCM) used by Newell (1980), featuring an ontology to represent concepts such as models, states, operators, elaborations, impasses, conditions, actions and working memory. The authors found this tool helped new model developers produce similar code and had fewer problems, as well as a frame of reference for understanding Soar code written by other developers. The Herbal solution did not claim to save model developers from learning Soar programming (in Soar Markup Language) as a prerequisite to generating the Soar models, only to ease the transition.

Herbal has also been used as a tool to better understand and explain intelligent agents. Haynes et al. (2009) used Herbal to generate agents in Jess and Soar, and as a mechanism to explore ontological explanations to better understand the problems, solutions, and consequences of using various design patterns using a B2B e-commerce environment as an example. They concluded the Herbal IDE can be used to demonstrate and understand various design patterns and help to build better agents.

Paik et al. (2009) extended the Herbal IDE to generate simple ACT-R models capable of representing more than KLM-GOMS or GOMS models, and compared the prediction times of dialing a simple phone number. The authors added a Declarative Memory component to Herbal to facilitate the representation of hierarchical or sequential tasks using a GOMS-like language for generating ACT-R production rules. They found the efficiency of the model files were widely distributed depending if Herbal was configured to generate code that resembles task
model patterns used by novice or expert modelers. Cohen et al. (2010) later expanded on this work to demonstrate how patterns are applicable to more general intelligent agent designs. Harrison (2020) used a different approach by creating the \textit{jACT-R Assembler} (JASS), an imperative programming language for modelers already familiar with the theoretic underpinnings of ACT-R, but also supporting declarative functions to generate common patterns. The JASS compiler uses a one-to-one mapping of instructions to productions, and offers a plug-in module for goal management. The authors also point out the method used in JASS to generate design patterns must make serious commitments on how language constructs are mapped to production rules.

The methods used in this thesis are somewhat inspired by the Herbal solution to bridge between architectures, however, the target level for theory and code generation is at the Cognitive band, rather than the Rational band, which is already addressed by PSCM and other knowledge level theories. Future related work outside of this thesis could very well look at using Herbal to transform between the Knowledge level ontology represented in Herbal, and Cognitive levels represented by the CMC, but without making the same level of theoretical commitments as a solution like JASS.

2.5 Model Comparison and Analysis

Comparing agents developed in different modeling environments is certainly a challenge across more complex problem domains and spaces, and comparing the components inherent in each architecture more so. Direct comparison between cognitive models running in ACT-R and Soar produces conflicts due to underlying assumptions about knowledge representation.
2.5.1 Patterns and Constraints

Jones et al. (2007) compared the patterns and constraints imposed by each of the ACT-R and Soar cognitive architectures. The authors compared the patterns generated by low level reasoning tasks and their effects on decision and action sequences, memory retrieval processes, conflict resolution, and search processing. Jones et al. found that both cognitive architectures can produce very different (and measurable) outputs, as well as opposing constraints and bottlenecks that make equivalent behavior model processing an extremely difficult task without implementing intermediate model constraints.

In Muller et al. (2008) we find a similar attempt to testing and comparing like-for-like ACT-R and Soar models using a tactical picture compilation task from the naval warfare problem domain with agents constructed based on a Belief-Desire-Intention (BDI) goal-solving paradigm, while measuring the performance of retrieving beliefs, reasoning, and control of actions in pursuit of the goal. The authors found the memory structure of Soar was similar to ACT-R (at that time), and there were elements that could be compared between them. For example, they found that while Soar did not provide the means to implement sequential controls capable of committing to the desired intentions (as ACT-R did), Soar did facilitate easy creation of sub-goals for impasse resolution, and responded more efficiently to external environmental cues. ACT-R models, in contrast, did not scale well to memory requirements and tended to become slower than humans over time.

These examples show how making comparisons between models can be extremely difficult without introducing some constraints. Docking, discussed in the following section, is one form of introducing and managing these constraints.
2.5.2 Docking Studies

One method of testing and comparing different models on the same task is through a docking approach, by creating like-for-like task design (and environments) the same between two cognitive task models. [Zhao et al. (2010)] used the docking approach to evaluate the measurable differences between ACT-R models generated with Herbal, and those created by hand, using Pirolli’s price finding task model as the problem domain. The authors used this methodology to compare model component equivalence with a focus on declarative memory use and efficiency, while largely ignoring perceptual-motor or sub-symbolic computations. Zhao et al. found that the two versions of the source code were structurally equivalent, however, differed in how the goal buffer is checked before production rule firing. The authors also found that when reverse compiling example Soar models generated by Herbal, that not all published models followed the PSCM on which Soar is theoretically based.

Docking can be used as a solution to set certain constraints on the task model and architecture; however, even if we are willing to make significant theoretical commitments using automatically generated productions, there is still a possibility that the underlying structure is no longer based on architecture-specific commitments. This thesis addresses this issue by identifying which areas of each architecture are based on the CMC, and generate these portions of cognitive model files without loss of fidelity across docking studies.

There are a few options for addressing the issue of how to choose one programming paradigm over another. The most successful solutions do not attempt to replace the model code, but rather to empower new modelers to work with it more effectively. Modeling paradigms are an integral part of this, as it frames the issue of a knowledge level separately from cognitive code, and presents a cleaner representation. The
issue of system levels is best approached through the use of conceptual models that explicitly represent these structures at their applicable levels.

2.6 Summary

The Common Model of Cognition isn’t here to simplify how we use the architectures, but to unify and open conversations we have about them, rather than imposing a specific structure on the brain. The community has no objection about using the ACT-R or Soar languages, as theories of cognition have already been formalized through their respective languages. Cognitive modelers should be not overly committed to high level macros, production systems, or low-level neurons as a be-all-end-all solution or approach.

The MISM meta-model framework and the CARINA meta-cognitive architecture are a fully realized implementation of a standardized meta-model based on Model-Driven Design that can be formally validated to the MOF User Model (M0) level; in other words, they provide an end-to-end solution for supporting meta-cognition in traditional BDI agents. The MISM solution is specific to Meta-Cognitive processes for agents based on Engineering AI, however, could be applied to the Common Model of Cognition.

Modeling tools like Herbal, JASS, and others, have smoothed the road for future students interested in cognitive modeling. However, task models generated by these tools for different architectures may solve the same problem in different ways, make different assumptions based on imposing constraints, and in some cases are largely incompatible with the target environment or problem space, making task-specific comparisons between them difficult. This discrepancy and disparity between architectures demands an approach that does not assume a one-to-one
transformation at the Cognitive level, but can still guarantee that the most
important theoretical assumptions are not violated.

Docking studies between task models in ACT-R and Soar compare these
architectures based on specific, equivalent micro-theories, but are written
specifically for these architecture components and, for the most part, cannot be
generalized to other CMC architectures.

The community should follow a formal method when working at and between
specific system levels, instead of arguing that programming at specific levels or in
certain languages is a final solution in and of itself.
3. RESEARCH PROBLEM STATEMENT

So how do we deal with the confusion about conflating systems levels, and the tools we use to analyse and describe them? Again, this is a problem of using a specification that adequately describes their requirements, and as such, should use a methodology with supporting representation and domain-specific language to define and describe them. When we talk about differences and commonalities of the architecture, then they have already been described in the CMC Meta-model, but not yet formalized according to a standard. If cognitive architectures and their models have a language, then so should meta-models of cognition! Paradigms such as Model Driven Design provide the language for cognitive meta-models and model definitions, transformations, mappings between models, and a whole (domain specific) language; I argue MDD should also be used to describe current and future CMC Meta-Models to support formal validation, instead of battling for a winner-take-all programming language, compiler, or tool chain.

I argue that if the community wants to abstract away from specific details, that it has to be done in a manner that supports those general features for top-down design, but still leaves room for the bottom up modeling features specific to each architecture. My suggested method of going about it allows modelers to use different languages for each level, and how each level can be used to compile down from a higher-level language, without imposing strong theoretical commitments on lower level languages.

When approaching the fundamental recurring issues in this way, we are not just bringing a specific technology (such as my compiler vs your compiler), rather, we are bringing order by using a formal method to turn the theory represented as an abstract
meta-model into something that structures the code, and accompanying Common Model theories, without imposing micro-theoretic implementation assumptions that reside at the lower levels of the cognitive model code itself.

This thesis addresses this problem by formalizing the CMC, and using this formal model to generate a system that consists of two parts: first, a cognitive architecture-agnostic interface that supports a meta-model level approach to problem solving, and second, an architecture-specific model file template generator that allows you to relegate problem solving down to the task model level (like ACT-R). Part of the methodology of this approach necessitates experimenting to see how far we can push the meta-model to accommodate these theories, and where they break down in practical application.

The following sections intend to expand on the issues identified in Chapter 2. In Section 3.1, I introduce a conceptual model to frame and position the main issues within the larger context. Section 3.2 expands on the pertinent research questions, and possible implications of answering them.

3.1 Conceptual Model

One way to approach these outstanding issues is by using a Model-Driven methodology to describe ontological categories and relationships for analysis. Consider the overlay of the Meta-Object Facility layers on to Newell’s System Levels (Figure 3.1), with each of the Meta-Object Facility layers representing a model Vocabulary, Specification, Architecture, and Instantiating for each of Newell’s Social, Rational, Cognitive, and Biological (Physical) bands. The Common Model of Cognition sits at the intersection of M2-Cognitive, and deals with exactly those specifications that are common to all cognitive architectures.
<table>
<thead>
<tr>
<th>MOF / System</th>
<th>M3 (Vocabulary)</th>
<th>M2 (Specification)</th>
<th>M1 (Architecture)</th>
<th>M0 (Instantiation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Social</td>
<td>Meta-Model of Macro-Cognition</td>
<td>Common Model of Macro-Cognition</td>
<td>Multi-Agent Systems</td>
<td>Real-time Interaction</td>
</tr>
<tr>
<td></td>
<td>(Social Philosophy)</td>
<td>(Sociology)</td>
<td>(Socio-Technical GOMS Theory)</td>
<td>(This Zoom Meeting)</td>
</tr>
<tr>
<td>Rational</td>
<td>Meta-Model of Rationality (Logic)</td>
<td>Common Model of Rationality</td>
<td>Problem Solving</td>
<td>Interruptible Sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Predicate Calculus)</td>
<td>(GOMS, PSCM, Multispace CM)</td>
<td>(Goal Stack, Sub-Goal)</td>
</tr>
<tr>
<td>Implementation</td>
<td>Meta-Model of Implementation</td>
<td>Common Model of Processing</td>
<td>Hard-/Soft-/Wet-ware (x86, LISP, Brains)</td>
<td>My Computer (Your Brain)</td>
</tr>
<tr>
<td></td>
<td>(Metaphysics)</td>
<td>(Turing, Von Neumann)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3.1.: Meta-Object Facility vs Newell’s System Levels

The Meta-Object Levels provide graduating levels of abstraction and specificity, in this example dividing up the vocabulary used to discuss a particular level, the specification of agreed-upon standards or rules, the design of a system based on these rules, and finally the instantiation of a design in a tangible application. Each row in the table represents one of Newell’s world theories or Bands, each encapsulating common functions described by a similar vocabulary, as introduced earlier. This thesis deals mostly with the Cognitive row, and the processes of going from a Meta-Model vocabulary, to the Common Model specification (A), to specific architectures (B), and finally to an instance of a task model that instantiates specific theories (C), that requires a specific architecture.

We can use this conceptual overlay to address the main issues and questions about how categories of modeling paradigms are represented in general top-down problem decomposition definitions, and how much must be generated using bottom-up state-specific traversal of a problem space, or catering to specifics imposed by a particular platform. Furthermore, we can use this perspective to address the issue of which
aspects of a given architectural specification should be represented at the Cognitive layer, if it should be realized at a different temporal band, or perhaps sit at a different level of specificity altogether.

### 3.2 Research Questions

The original research questions introduced in Chapter 1 are restated here, with additional sub-questions based on recurring issues:

- **Q1**: How much of the Common Model of Cognition can be defined *formally*?
  - Q1a: What does a formal model look like?
  - Q1b: How do we know that it is formal?
  - Q1c: What cannot be defined formally?

  This research question has been addressed by creating a meta-model definition that reflects the definitions of the CMC, and by generating a formal specification that can be compared and validated between evolving versions of the meta-model definition.

- **Q2**: How much of a task model file can be generated from a formal Meta-Model of Cognition?
  - Q2a: Would a generated model file work for all task models?
  - Q2b: Are implementation- and task-specific model elements still required from the model developer?

  This research question deals with the generation of a model from a meta-model, and has been answered by using a candidate meta-model to specify a generic model template for two particular architectures, namely ACT-R and Soar, but only includes assumptions from the CMC; no more, no less.
• Q3: What is the effect of applying a formal Meta-Model of Cognition definition to specific components?
  
  – Q3a: Is the meta-model sufficient for design time configuration of a model?
  – Q3b: Would a formally-defined component make any detectable difference at run time?

Answering this question required generating a common interface, and mapping them to architecture components, to run test cases using commonly mapped attributes, accessing architecture-specific components and settings defined in the meta-model before it is run, and then analyzing the results during and after running the model.

• Q4: What kind of predictive power is in a Meta-Model of Cognition?

  – Q4a: Will it predict quantitative measurements of task model performance?
  – Q4b: Will it predict qualitative characteristics and attributes?

Answering this question required collecting metrics during model execution, such as the Production System clock speed, and ability to read from and write to specific buffers, and model output.

• Q5: Can a Meta-Model define constraints to validate or invalidate cognitive architectures?

  – Q5a: What are the constraints?
  – Q5b: How do we validate them?

Answering this question required defining a specification based on formal requirements definitions, comparing qualitative and quantitative results inferred
from a meta-model specification against outputs, and testing whether or not results can be validated against the CMC meta-model definition during or after model file execution.

- Q6: Does a formal Meta-Model of Cognition specify common environment interfaces that work on all CMC architectures?
  - Q6a: If so, are there specific environments that wouldn’t work?
  - Q6b: If not, where does it break down?

Answering this question required making some assumptions about how cognitive agents are to perceive and act upon an environment, and ensuring like-for-like interfaces are supported and used by candidate architectures. The approach used to answer this question required creating a mapping between feature space and complexity for arbitrary environments, and their architecture-specific representation.

### 3.3 Implications of Research

Distinctively defining and distinguishing between cognitive architectures has already been done by the modeling community. However, this is not the same thing as formally mapping them. Formally mapping the components and relationships of the CMC to specific architectures, and showing that this mapping works in practice, is the main contribution of this thesis.

Answering these questions informs researchers on the viability of using a formal meta-model definition to systematically generate, evaluate, and possibly validate different cognitive architectures that purport to implement some or all of the Common Model of Cognition. This is important because it will inform the Common Model community on the levels of abstraction that conform to the CMC, before a
modeler is forced to implement a test using architecture- or problem-specific imperatives.

A possible side effect of answering these questions is a corpus of additional information that can be used to inform the use (or misuse) of cognitive agents in complex domains that traditionally require a high degree of trust.

3.4 Summary

The current state of the art does not support comparison of cognitive models and architectures against a formal standard. We face an issue of architectural disparity, a lack of verifiable test metrics or environments based on a standard model, and the many difficulties experienced when trying to perform direct translations between model files.

The research addressed by this thesis is based on the goal of understanding how to apply formal model definitions to cognitive models based on what we already understand about the human mind, to generate, test and evaluate the performance of virtual agents through various stages of model-driven development and growth, and draw these inferences through time and space, both within and without each model.

The next Chapter outlines the Methodology of this approach; for defining and creating an example Meta-Model of Cognition from the statements of the CMC, a tool chain to demonstrate the theoretical and practical feasibility of using MDD to produce a meta-model definition, to formally generalize and template those features and modules inherent in and inherited by the CMC architectures, and using these model templates to systematically generate the required interfaces used for comparing these implementations in target domains.
4. METHODOLOGY

4.1 Introduction and Outline

This thesis proposes a novel methodology for cognitive modeling, based on Model Driven Design (MDD). Specifically, the claim is that placing cognitive architectures within an MDD framework will reveal common computational structures inherent in the architectures, and where they diverge. Critically, the result will be a formal theory of similarities and differences. That is, within the framework, the code can be treated as the theory. The method, as described in this section, describes the process of connecting Soar, CMU Lisp ACT-R, and CCM Python ACT-R into such a framework. These were chosen due to the common model claim in (Laird et al., 2017b) that Soar and ACT-R share common theoretical elements. Lisp ACT-R, and Python ACT-R were both included to create a case where the theoretical claims are purported to be the same, but the implementation is different. The Sigma architecture, however, was excluded from this research due to incomplete implementation, and reliance on a commercial Lisp compiler to make use of all required functionality.

Using a model-driven software engineering paradigm as a cognitive science method is somewhat unprecedented, so some comments are needed. The first is that this method is used for complex projects, such as building operating systems. Because cognitive architectures are complex systems, showing how this approach can be applied to individual architectures is arguably a benefit in and of itself (e.g., formal models would make cognitive architectures more comprehensible to the AI community). However, the point is to use this methodology to make formal comparisons between architectures for scientific purposes.
The methods and use case involves a software engineer following the principles of MDD to put different architectures into the same framework. The logic of this approach is the same as hypothesis testing in a psychology experiment. If a competent engineer can find a way to connect to the architectures using a common interface defined by the cognitive meta model then the hypothesis that formal commonalities exist is supported. However, if they cannot find a way to interface them, then the null hypothesis should not be accepted because, as in experimental design, the designer could have made poor choices that caused the failure. Likewise, even if the meta model is a success, it is possible that it could still be considered a greater success as more refined future version of itself [MacDougall et al., 2015].

Following MDD principles, the goal is always to improve through iteration. Thus, this should be considered the first iteration. Once the MDD system has been created it can be further evaluated by running different models across the architectures, within the MDD framework.

The remainder of this chapter is organized as follows: First, Section 4.2 introduces the use of Model-Driven Development as a methodology, how it relates to the CMC, and supports scientific investigation in this problem domain.

Next, in Section 4.3 we present the Common Model Platform: a conceptual design with examples of how MDD can be applied to the CMC using a mature tool chain to support a verifiable instantiation of CMC meta-model, architecture-agnostic model, and a practical example using a simple prototype in development.

Section 4.4 describes the application of MDD outside of the specifications of the CMC for the purpose of generating both environment-agnostic and environment-specific interfaces, with consideration to perception, action, and time.

A Common Meta-Model of Cognition (Section 4.5) describes an application of Model-Driven Development methodology to create an example Meta-Model of the
Common Model of Cognition, independent of any one specific cognitive architecture. This section covers the layers of the Meta-Object Facility standard (herein MOF), to formalize the mappings between abstracted and implemented functional and non-functional requirements specified by the CMC, and the different cognitive architectures that implement them. This formal design and analysis answers questions dealing with how the CMC can be considered a formal definition, such that the various implementations can be compared, measured, and validated using an industry tested and accepted standard.

In Mapping to a CMC Framework (Section 4.6), we turn our focus to each individual cognitive architecture, to explore each logical and functional area, and how they can (or possibly cannot) be mapped to a common interface defined by and generated from a formal meta-model of the Common Model of Cognition. This approach answers questions that deal with the unique interfaces provided by each cognitive architecture, and the generation of a middleware layer to support them. This section also expands on the CMC meta-model definition for Perceptual and Motor modules defined by the CMC; specifically addressing sensory interfaces to be mapped, measured, and evaluated independent of the target environment. This description also includes the MDD framework code, which can be found in (Peters, 2021).

In Section 4.7 we discuss the Experimental Design with respect to participants, instruments, procedures, and materials used in the study. Section 4.8 provides a review of the different virtual environments internal and external to the models, and how they are used to generate the data to measure and analyze. Section 4.9 presents the four perspectives used for analysis, being computational, architectural, behavioral and environmental, and how they each contribute to answering the research questions. Section 4.10 subsequently describes how the generated data and analytic perspectives
can validate or invalidate the research hypothesis for each research problem theme. And finally, in Section 4.11 exposes some of the inherent risks and limitations of this methodology, and possible threats to the validity of derived outcomes.

4.2 Model-Driven Development

Model-Driven Development (MDD) is a paradigm used in software engineering and computer science for the design of large scale and complicated systems. The purpose and philosophy behind MDD is to make use of abstraction and automation as much as possible to ensure a solution is correct by construction versus relying on test-driven specifications alone. The Model in MDD refers to the use of an abstraction away from specific implementation (Figure 4.1), allowing the modeler to focus on the problem space, much like cognitive models abstract away from the underlying implementation of architecture-specific micro-theories on a specific platform.

Fig. 4.1.: MDD provides layers of abstraction from representation to instantiation.

This approach allows designers to step away from specific details that do not add value to the intended purpose of the current system perspective.
Using MDD to define and generate CMC-based models is not going from a problem definition directly to a product design, but rather going from a problem through a theorizing paradigm to a product design; this is a slightly longer route to defining a theory of cognition through the model, however, the primary product then becomes a scientific theory represented formally as code.

4.2.1 Domain Model

The Model-Driven Development (MDD) approach begins with a user’s requirements that are not as detailed as the code that implements the solution. This design approach begins with an abstraction of the system through a model diagram (such as UML), and goes through a process of transformations and mappings to refine it for the target platform or architecture. There are different types of software models depending on the context of the application. For example, two broad types of models are domain models and conceptual models. According to Kriouile (2015), the MDD approach can be broken down into three main steps: describing the domain and vocabulary using a Computation Independent Model (CIM) that can describe the solution without assuming a computational medium; second, creating a Platform-Independent Model (PIM) view of sub-systems independent of any specific technical platform; and last, specifying the technical requirements of a Platform-Specific Model (PSM) to be used to generate the executable code for that platform (Figure 4.2).

MDD is a solution that bridges the gap between both attitudes: a process of refinement from an initial (common) model into a specific architecture using transformations. The Object Management Group (2014) defines MDD as part of a larger Model-Driven Architecture (MDA) to distinguish between the Modeling Space, and the Code Space (Figure 4.3).
The Model Space describes a successive layering from the most abstract concepts independent of any specific design, to the least abstract, typically represented as a UML-compliant description of classes and relationships for a target platform. The most abstract layers of the Model Space are Meta-models that are Platform-Independent Models (PIM); these are necessarily so for cross-platform support and for future compatibility. The lowest levels of the Model Space include Platform-Specific Models (PSM), that include specific constraints for a particular target computer architecture or environment, and yet by themselves are not the final product or implementation.

The Code Space, in comparison, represents code that has been transformed and generated for a target architecture or platform that is fully realized and capable of either interpreting or compiling the generated code. Crossing the threshold from Model Space to Code Space necessarily involves a transformation from a PSM Markup
Language transformation to a target file format. One example platform that does this very well for Java environments is the Eclipse Modeling Toolkit (Appendix G.1).

This approach is useful for cognitive modelers as it demonstrates how to go from an abstract cognitive model, to a formal design artifact, then to generate the necessary test artifact, and ultimately prove the conformity to specification such as the Common Model of Cognition. The MDD approach affords the assumption that model code on different platforms is not going to run exactly the same, yet since the code is generated from the same template, the modeler will know they will adhere to the specification.
4.2.2 Model Transformation

The MDD approach uses paradigms from Model-Driven Architecture (MDA) to achieve specification through various levels of abstraction. However, in order to extend beyond platform-independence to fully realizable software systems, our approach requires a transformation that is domain-specific for the test cases and environments. Here we use Domain-Specific Modeling (DSM) to generate the transforms required to match the cognitive model for each problem domain.

Fig. 4.4.: MDA processing cycle, adapted from Kriouile (2015).

To accomplish this, the DSM transform exposes the features that are available on the target platform in such a way they can be modeled independent of the implementation details. For example, a source Cognitive Meta-model that defines the CMC according to the MOF specification could be transformed from one representation format (like UML), to another target format (like XMI), and still
conform to the MOF standard. Subsequently, a source cognitive model instance that conforms to the UML specification could be transformed to its target XMI representation, and still conform to the target Cognitive Meta-model due to conformance of the transformation engine itself.

4.2.3 Test Specification

Another feature of MDD is that it can be used for generating tests from a specification, allowing the modeler to verify and confirm that the end result conforms to the requirements of greatest value. This approach uses a similar form of template to generate tests for the target platform, and then relies on either code generation or subject matter experts to flesh out the final test script while remaining true to the specification.

For example, consider a simple agent that needs to communicate with an external environment. The environment can be described as a state diagram of simple observation and action, where the problem space is represented as states, and the possible actions that can be taken connect each of the states. Depending on the environment, the problem space may be simple or complex, and the action space may be discrete or continuous. Regardless of the simplicity or complexity of the problem domain, each of these observations and actions are represented in the agent through an internal state diagram, or mental model, that uses constituent cognitive model components specified in the Common Model. The Processing aspect is captured through state transition; Working Memory is captured through the current state; Long Term Declarative Memory is captured through a representation of all currently known states and actions; Long Term Procedural Knowledge is represented as the mapping of known states and observations to their corresponding next actions; Perception is mapped to the current (or expected) observation of the
environment; and Motor actions are represented through the implementation of action selection and state transition. There is also a possibility of generating agents that that aren’t doing the same thing from a behavioral perspective, and yet not violating the assumptions of the Standard/Common Model, demonstrating that different models on different platforms are still valid models in the problem domain.

4.2.4 Model Validation

A UML diagram, such as the one developed for the CMC (discussed in Section 4.5), is not in and of itself enough to provide all of the requirements of a formal specification. A formal model requires additional constraints about the objects of the model; in computational cognitive modeling, this is ideally done by using an unambiguous natural language that is accessible for modelers of various skill levels without requiring a heavy background in mathematics (as opposed to other fields such as chemistry or physics). Formal languages, such as the Object Constraint Language (OCL) were developed with exactly this purpose in mind. OCL can be used for a number and variety of different purposes, including (but not limited to): specifying invariants on the structure or classes in a meta-model, specifying initial values for class attributes, deriving values, querying values, or specifying constraints and conditions on operations performed by classes.

There is also a distinction between OCL types that can be used to validate CMC models. First, there are structural constraints that limit functional choices based on a specification at design-time. These constraints must be met at all times; before, during, and after a model is run. Second, there are behavioral constraints that address emergent behavior at run-time. The OCL expressions that are applied to the Meta-Model of Cognition are explained in additional detail in section 4.5.4.
4.2.5 Relationship to the Common Model

The philosophies behind MDD and the CMC use very similar paradigms, and to some extent, are trying to accomplish the same objective. The CMC abstracts away from the underlying implementation of specific architectures, and rather describes the general components, characteristics, behavior, and to some extent the functionality, of all models that conform to CMC architectures. Thus, one could argue that both of the MDD and CMC make use of the principle of abstraction to describe the meta-model, or functionality and behavior of all models in the cognitive domain.

4.3 Application to the Common Model of Cognition

What the CMC is currently missing, as discussed at length in Chapter 3, is standardizing on the principle of (model) generation to move from an abstract meta-model, through domain specific modeling and transformation, to the ultimate generation of tangible constructs in, and for, a target environment. This approach must satisfy certain architectural and behavioral constraints, independent of the tests, through the principle of correctness by construction. The following section describes an application of this process to the CMC in detail.

4.3.1 Meta-Object Facility

The Meta-Object Facility (MOF) specification provides a systematic framework necessary to describe the relations between layers of abstraction, and enable inter-operability between implementations of architectures and supporting tools. The CMC, then, can be described according to levels described in the MOF specification as shown in Figure 4.5.
• The Meta-Meta-Model level (M3) represents the MOF standard, a necessarily recursive grammar capable of self-representation.

• The Meta-Model level (M2) represents the CMC Platform, and defines the Meta-Classes representing the common components, attributes, and relations described in the Common Model of Cognition.

• The Model level (M1), represents a model of the individual architectures (e.g., ACT-R or Soar) that are an instance of the CMC Meta-Model, and instantiate components of the CMC meta-classes in M2.
• The User Model level (M0) represents the real-world instances and cognitive task models that runs on a single M1 Architecture, as used for every day testing of intelligent cognitive agents.

It is important to reiterate here that the use of the terms “Meta-model” and “Model” in Figure 4.5 are based on MOF terminology, and may not line up perfectly with the distinction between a Cognitive architecture, and the Cognitive model files that run on them. In the context of applying this MOF standard to the CMC, the MDD can use this framework and, coupled with the right design tools and generative techniques, take a model of a system and generate or create more specific (and usually complex) components with pre-defined functionality and interfaces that can be described independent of the implementation medium.

For this purpose, I present a Common Meta-Model that can be used to describe and generate an interface that adheres to the requirements of the CMC, can be mapped to each cognitive architecture, and used to test task model templates independent of the target architecture, according to design principles of Model Driven Development. This meta-model and framework can also be used to generate statements to control and validate that the structure and behavior being observed is not in violation of the properties specified at the CMC level.

4.3.2 Use Case Design

A holistic platform that follows the MDD paradigm must adhere to and accommodate the design principles outlined in the MDD methodology: tools to create a Platform Independent Model (PIM) based on the CMC Meta-model; creation of Platform-Specific Model (PSM) for each cognitive architecture based on the PIM; generation of Model-to-Model and Model-to-Platform interfaces that allow different models and environments to communicate. In addition to these
requirements, it should support a log collection system to compare the results generated from executing specific tasks. This form of platform design is practical and can be used to bridge disparate and sometimes incompatible technologies that underpin a subset of systems of interest. My approach to integration and design uses existing architecture and environment APIs where available and necessary.

**Modeling Tools**

A Common Model platform based on a Model-Driven Architecture should support development from a top-down meta-modeling approach, and generate the required structures as far as (but no further than) the theories expressed in the Common Model as supported by the target architecture. Supporting the assumptions and
various micro-theories embedded in cognitive architectures are typically invoked as part of a bottom-up architecture-specific development processes; picking one micro-theory sometimes means the exclusion of others, and assuming otherwise only limits potential tests of competing micro-theories.

Our meta-modeling approach, therefore, requires tools to model at the highest level using a domain specification language template that can be implemented and refined for specific architectures, and instantiated as lower levels of functionality and specificity only as required. From a usability perspective (Figure 4.6-A), the Common Model community agrees upon those standards that are included in any proposed meta-model.

The CMC platform shall then use this meta-model definition to support Model Developers (Figure 4.6-B in high level design of architecture-agnostic models, with a supporting toolkit and workbench. The platform shall be able to use this model definition to generate equivalent text-based model file templates, and generate the requisite interfaces to be used for further testing and evaluation.

**Interface Generator**

The lower level interface approach necessitates a platform that can expose the common elements of both architecture and environment; a form of universal *coupler* for agent design and experimental docking. A CMC platform should facilitate a rigorous test methodology capable of supporting cognitive architecture selection, agent configuration, micro-theory selection (if applicable), and collecting agent-specific metrics used in evaluating architecture- and component-specific performance and behavior using APIs exposed by Cognitive Architectures (Figure 4.6-C).
A CMC platform should also support environment selection, test selection, test definition, and reporting on various performance and behavior metrics of the cognitive agent in those environments, using APIs exposed by Virtual Environment Developers (Figure 4.6-D), data internal to agents, external observations of those agents, and supporting metadata from the environment or problem space being tested. For systems that do not present an accessible interface, I have extended scripts where necessary, or made of third party protocol buffers or sockets for inter-process communication.

Affordances and Constraints

The Common Model both describes and prescribes the fundamental characteristics of the participating architectures, while leaving some details outside the boundary of what constitutes conformance. This level of descriptive power should remain with the meta-model, while at the same time limiting, or constraining, those features that may invalidate a model as a proper reflection of the Common Model of Cognition.

Upon closer inspection there are some features of ACT-R and Soar that, while more specific than what the Common Model dictates, is still common among all architectures. One example of this is the implementation of chunking. Chunking is mentioned as a mechanism of information storage; in ACT-R, chunks are stored as discrete slots, whereas in Soar, they are stored as attribute-value pairs in a rigid tree-like structure. Attribute value pairs, while not a mandatory feature, can also be simulated in ACT-R (and often is in practice). Both implementations of chunking are more specific than what is described by the Common Model, and yet their underlying implementations are fundamental aspects of each architecture with varying room for adjustment on behalf of the model developer.
A Common Model Platform, therefore, could also accommodate such cases where specific implementation details between Common Model architectures are implemented based on similar micro-theories and constraints, or using similar data structures, without detriment to the Meta-Model representation, so long as these are clearly documented as such.

4.3.3 Architecture Interfaces

Each cognitive architecture in this study provides a selection of interfaces to support instantiation, control, and monitoring of models running in their respective runtime environments. As an extension and necessity of the research problem in this thesis, the Common Model platform uses a middleware layer that serves as a proxy between models generated by the meta-modeling workbench, the cognitive architecture interfaces used by these model files, and the virtual environments selected for testing and evaluation. The interface layer can be generated by any Model-to-Text (M2T) software tool that implements the MOF standard. This middleware then serves as a dispatcher service between a cognitive model file running directly by the environment (e.g., using LISP syntax), or by exposed API such as Remote Procedure Calls (e.g., using Java or Python), and a target environment, depending on what the target architecture makes available to developers.

ACT-R

The ACT-R Environment can be used one of two ways: the first is through the provided Graphical User Interface (GUI), and the second is through Remote Procedure Calls (RPC). ACT-R provides command line access through the default
host system terminal, supports a GUI based on Tcl/Tk, and runs in any Windows and Linux environment as long as the necessary Lisp libraries are present.

```python
def add_key_monitor():
    global key_monitor_installed

    if key_monitor_installed == False:
        actr.add_command("key-press", respond_to_keypress,
                         "task key output monitor")

    #"output-key" is the signal produced by the keyboard device
    actr.monitor_command("output-key", "key-press")
    key_monitor_installed = True
    print("key monitor installed")
    return True

celse:
    return False
```

Fig. 4.7.: Example of ACT-R Key Monitor Binding.

The current version of the CMC middleware platform launches an ACT-R environment in Linux, and interacts with the RPC dispatch service using two interfaces. First, an interface with key bindings registers a function that is called whenever the specified binding in the ACT-R kernel is executed.

For example, Figure 4.7 registers a function to be called whenever the running model presses a key on the keyboard. This interface then captures keyboard commands to be send to an external environment (in this specific example, and OpenAI CartPole game). Second, a pre-configured model file (generated from the MTL) is then loaded by the ACT-R runtime environment, and launched as normal.

Figure 4.8 illustrates an example model file that makes use of a Motor Module to send a press-key command to the manual buffer, which is then intercepted by the dispatcher service and sent to the registered key monitor for external processing.
The final version supports both user-generated and template-based model files, interface visual and motor buffers directly through the Inter-Process Communication (IPC), and measure sub-components of ACT-R by parsing JSON-RPC output using the *dispatcher* service.

**Soar**

The Soar Environment comes in pre-compiled and source code versions for both Windows and Linux operating systems. Soar uses a custom *Soar Modeling Language* (SML), and accepts options through a separate configuration file. The environment offers a debugging GUI based on Java, and an early version Python interface.

The current version of the CMC platform launches the Soar environment in Linux, supplies a pre-configured model file based on user-defined variables, a corresponding agent options in a separate configuration file, and captures agent metrics through standard output. Interfacing with Soar is done in a similar fashion.
as ACT-R, however, a Soar agent is instantiated and registered with the Soar kernel at runtime (Figure 4.9). Instead of interfacing with the kernel through RPC, the middleware interfaces with a proxy agent acting on the kernel's behalf, and provides access to data structures representing Working Memory Elements (WME).

```python
def get_move_command(agent):
    output_command_list = { 'move': ['direction'] }

    if agent.Commands:
        (commands, mapping) = parse_output_commands(agent, output_command_list)
        move_cmd = commands['move']
        direction = move_cmd['direction']
        mapping['move'].CreateStringWME('status', 'complete')
```

Fig. 4.9.: Example of Soar Key Monitor Binding.

Model files in Soar follow a similar structure to ACT-R, and can be generated from the same PIM using a platform-specific MTL generator (Figure 4.10).

```
# Apply the movement

sp {apply*move*create-output-command
    (state <s> ^operator <o>)
    (<o> ^name move
        | ^dir <dir>)
  -->
    (<s> ^output-cmd <output-cmd>)
    (<output-cmd> ^direction <dir>)
}
```

Fig. 4.10.: Example of Soar Model File with Output Command.

The CMC platform supports both user-generated and template-based model files, interface directly through the command line or Python interface, and measure sub-components of Soar by parsing output using the terminal I/O services.
Model Components

The main functional CMC modules are measured across the computational, architectural, and behavioral perspectives; and to do so we need to interface through an equivalent mechanism.

Structure and Processing is verified through a computational analysis of the environment timing cycle, as well as an Architectural analysis of inter-module communications.

Working and Long Term Memory are verified through an analysis of Memory-specific module measurements for storage and recall.

Perception and Motor modules will be verified through an analysis of State observations presented by the environment, mapping to an internal representation, and by comparing expected model-derived actions with those generated by agent outputs in each of the test scenarios.

The interfaces generated from a formally defined meta-model can therefore be used to inspect, verify, and validate each major component defined by the CMC. To do so, we need to capture these results not just within the models, but also from the perspective of the environments in which they are situated.

4.4 Capturing Experimental Results

The MDD approach can go beyond capturing cognitive models, to capture the environment and the experiments, and thus provide a seamless integrated framework for the modeller. This is easier said than done, since the most common practice in Cognitive modeling is to assume a transduction hand-off; that is, once the Production System issues a command to a module (as is done in functional programming), there is an expected result with no concern as to how that result is generated — the command
is based purely on an instructional format. My approach is going beyond the CMC, because with MDD we can describe the transformation that we need in order to make a connection to different environments, while maintaining traceability, regardless of which level generated the specification.

4.4.1 Perception

There are two fundamental ways to view interaction between an agent and an environment. One is from the first-person perspective of the agent, as it experiences the environment. The other is from the perspective of the environment as the agent operates within it. A Common Model platform that serves to interface between them necessitates an understanding of, and transformation between the perceptions and actions of the agent within the environment. Perception of environment-generated events that occur independent of the agent should be represented in a format that can be processed by any agent, and accept feedback from the agent in a format such that agent actions effect change in the environment.

The Common Model prescribes the ability to translate raw signals into a symbolic and sub-symbolic representations with relationships and associated metadata, and that a cognitive agent may support multiple modalities of perception through visual, auditory, and haptic interfaces; however, the CMC does not make any commitment as to how this information is represented, only that it is. Further, the CMC assumes the agent will have an attentional bottleneck that limits the amount of information that can be represented and processed at any given time, and as discussed, is an established characteristic of Working Memory Elements.

A Common Model platform, therefore, should support the instantiation of multiple perceptual components capable of accepting, processing, and storing representations of the environment, through the presentation of this information in a manner and
format that *caters* to the perceptual interfaces exposed by each Common Model architecture.

### 4.4.2 Action

The Common Model also prescribes the storage of internally represented symbol structures and metadata in motor buffers capable of transforming this information into control-specific signals for each effector; the CMC, however, does not make any assumptions about the number or variety of effectors that represent limbs or locomotive apparatus, as this is largely environment- and context-specific. A Common Model platform, therefore, should support the instantiation of multiple motor components capable of accepting, processing, and producing external representations of the agent’s embodiment, through the presentation of this information in a manner and format that *supports* the motor interfaces exposed by each Common Model architecture.

### 4.4.3 Time

Agents operating in any virtual environment would necessitate a stimulus-action cycle, regardless of internal representation. From the Common Model’s perspective, the internal processing of pattern-based invocation can be any combination of internal-based execution (i.e. meta-cognition for awareness, or phonological loop for language), as well as external-based execution (e.g., gross or fine motor movements) to facilitate action, so long as the internal clock of the agent runs at approximately 50ms (corresponding to Newell’s deliberate-act cycle), independent of how long they take to perform in the environment (whether real or simulated). The Common Model Platform, therefore, should support interfaces that present environmental
stimulus, and accept agent actions, according to the assumption that both cognitive
agent and environment must run at the same clock speed relative to one another.

4.5 A Common Meta-Model of Cognition

This section describes the first main contribution: an application of the MDD
methodology to create a candidate Meta-Model of Cognition, and a formal mapping
of these components to their respective cognitive architecture interfaces. I have
created this mapping to produce a tangible Meta-Model that can be used as a
template for different applications; including (but not limited to) viewing the CMC
from the perspective of a platform-independent software architecture, and a formal
e numeration of functional and non-functional interface requirements. These answers
are important as, taken together or separate, leads to a better understanding of
Common Model Architectures.

The question of Task Models deals with an aspect of cognitive modeling that, while
conceptually sitting at the Knowledge (vs Cognitive) Level, forms an integral piece of
any cognitive modeling experiment. The model code relies on, defines, and executes
the very micro-theories that lie dormant in any cognitive architecture. Model files deal
with both levels simultaneously, and exploring where to “draw the line” between them
using different code generation techniques will help us to fundamentally understand
how much of a Task Model file is genuinely unique to the knowledge-level task. An
immediate (and perhaps obvious) benefit would be dynamically-generated templates
for an instantiated architecture, or task environment integrators for task models; a
more elusive (and less obvious) potential benefit is an understanding of how to define
the boundaries of theory testing for Cognitive level validation, versus Knowledge level
validation (if at all possible, or desirable).
4.5.1 Epistemological Perspectives

Now, before describing each step of the Formal Modeling process, it is important to understand that we are dealing with two epistemological perspectives.

First, we are dealing with the common architecture-agnostic components and functions as described in the Common Model of Cognition, and fundamentally the same for ACT-R and Soar; these functions implement Macro-level aspects of cognition that implement general functions within and across modules.

Second, we are dealing with the different Micro-level, architecture-specific methods of implementing identical functions, such that there is a different way of doing it in ACT-R, a different way of doing it in Soar, etc. Each architecture allows you to create different models that essentially accomplish the same task, albeit via different strategies, skill levels, and using different assumptions about the agent’s cognitive makeup.

It is important to distinguish between these two perspectives, since making assumptions about Micro-level implementations at the Macro-level of cognition forces a modeler to make strong assumptions about the underlying theories of cognition that have not yet been agreed upon by the CMC community. Thus, we deliberately separate these two perspectives in the Common Model Platform.

4.5.2 Specification and Process

The meta-modeling standard described by the Essential MOF (EMOF) standard describes a high level process that can be followed regardless of the domain, for the purpose of producing a flexible and extensible Domain Model consisting of objects and relations representative of the concepts pertinent to that domain. This meta-model was originally defined using the Eclipse Modeling Framework (EMF) definition of the
MOF, and represented as an *Ecore* file that is structured according to the Essential MOF specification, however any tool capable of representing EMOF models in a standard format (i.e. XMI) can be used for this purpose. The process is intended to provide all of the components required to successively generate each layer of the MOF, to produce a Platform Independent Model (PIM) that can later be translated into a Platform-Specific Model (PSM), as described earlier in section ref methodology.

- **M2**: Meta-Model Definition
- **M1**: Architecture Interface Generation
- **M0**: User Model Generation

Meta-Model Definition (Section 4.5.3) deals with the M2 layer, or Meta-Model Definition represented as a class diagram. This process involves gathering and refining the core concepts in the CMC. Concepts at this stage are represented using the Universal Modeling Language (UML) to graphically represent the classes, attributes, and relationships between concepts.

Abstract Interface Generation (Section 4.5.5) describes generating the M1 layer as a process of CMC Model Transformations. This transformation turns the UML classes defined in M2 into a series of Architecture Model interfaces better suited to high level programming languages, to interact with the run-time components in each cognitive architecture.

The last section, User Model Generation (Section 4.5.6) describes the process of the M0 layer as an Implementation (Generation) of User Model templates based on these components and relationships, to take advantage of the common (standard) task-independent entities present in all user model files.
4.5.3 Meta-Model Definition

The method of establishing a Meta-Modeling Definition in this thesis is largely inspired by the process used by (Beydoun et al., 2009) for modeling Multi-Agent Systems, with consideration for the Meta-Cognitive Modeling processes used by (Caro et al., 2014) for classification and identification of concepts and relationships most applicable to the theoretical and practical applications of Cognitive Modeling.

![Meta-Modeling Process Diagram]

Fig. 4.11.: Meta-Modeling Process

The process of creating the meta-model, depicted in figure 4.11, begins with a collection of terms extracted from the original Standard Model of the Mind, and other relevant terms found in cited literature, to establish a vocabulary of candidate keywords. The terms are then successively grouped and categorized, resulting in the components and relations that establish the primary meta-model. My application of this process was necessarily cyclical, as a practical application of the model produced from a first pass revealed various inconsistencies between my initial interpretation as described, and the practical application of the model to interface design, as described in Section 5.2.
The primary sources of information are derived from the architectures defined in the current CMC, specifically, the ACT-R and Soar cognitive architectures. The general steps of meta-model derivation are iterative in nature. The current CMC defines a mixture of components, relationships, functions, and specifications. Extending the applicable definitions to a standardized meta-model must then be performed through some interpretation; only the functional ontological relationships can be mapped at this level.

**Concept Extraction and Classification**

The cognitive architectures discussed as part of the CMC use a number of different concepts to describe the underlying structure and processing of their cognitive models. Concept extraction is a refinement of the terms that were collected and pooled. The concepts were collected directly from the CMC publication, the graphical diagrams that represented the main functional areas, and extant sources associated with the original primary architectures mentioned therein.

The first grouping deals with general concepts presented as part of a model, or cognition in general, and references to structural components or relationships. These groupings were only used to help cluster the terms in context or subject areas in which they were most frequently encountered. Key words were clustered into discussions about Models, Modules, Processing, Memory, Learning, Perception, Action, and the Environment.

**Invariant Concept Identification**

Next, the concepts were refined into unified representations based on their shared meaning or definition. This is the short list of candidates that have clear definitions or
consistent usage throughout the CMC paper and accompanying manuals and reference guides. Concepts are then revised based on shared words, or definition of concepts based on context.

Differences between the definitions were reconciled before putting them into a short list with definitions. For example, working memory, global working memory, temporary global space all refer to a form of Working Memory. This process was followed through terms based on general areas of cognition consolidated by the original CMC functional regions.

**Classification of Groups**

Clusters of multiple overlapping terms were then grouped into general concepts; for example memory buffer, retrieval buffer, visual buffer were all considered belonging to a general concept of buffer. In this case, a buffer would be considered a new, general term that all forms of buffers can be considered a specialized instance. All of the original, as well as the new terms, form a concept represented as a Class.

Last, the components and their relationships were categorized into logical packages with system inter-dependencies. This is where concepts were sorted into one of two groups: design-time, and run-time.

Design-time concepts are mostly static within each architecture, where modelers do not have much ability to define or reconfigure certain aspects. For example, the architecture kernel, production system algorithm, memory retrieval interface, are all fixed or immutable constructs that can only be pre-determined when the architecture is compiled before any user model is run. These are still required in the meta-model to represent the interfaces that allow some degree of custom configuration for theory testing, or example changing the level of noise in chunk activation, or the amount of time each cognitive cycle runs independent of outside interference.
• Processes: Production System, Activation, Utilities

• Components: Declarative, Working Memory, Perception, Motor

The second category, “run-time”, consists of components that are defined by the modeler in the form of a Task Model that runs on the architecture. These components and their aggregates are for the most part malleable and constitute the definition and instantiation of specific modules, or task knowledge required to fire productions.

• Components: Rules, Actions, Conditionals, Chunks, Slots

• Processes: Perceptual, Motor

Identifying Relationships between Elements

Classes relations are can be visualized using standard UML symbols; for example association (line), generalization (arrow with line), or aggregation (diamond with line). If an element is only comprised of a simple value (Integer, Boolean, etc.), that element is then considered an attribute of a class. In a UML class diagram, attributes are depicted inside of the class.

Generalization defines inheritance, or an “is-a” relationship, in which a specific concept captures an attribute that other concepts have in common. For example, both a memory buffer and a perceptual buffer are both buffers, therefore they would share a common parent in a hierarchy of buffers. Examples of inheritance found in the CMC include; declarative is-a memory (B1), procedural is-a memory (B1), etc. These relationships were used to generate the initial hierarchies:

Aggregation denotes a composition, or “has-a” relationship, where one component contains or is comprised of a collection of other components. For example, a rule is composed of both a condition and an action, as both components are “owned” by
the rule and are involved in its practical use. Aggregate relations are also defined by their multiplicity; a range defining the minimum and maximum amounts of that aggregate.

Association is a functional relationship, usually denoting some task that one concept performs on another to some effect. For example, a cognitive cycle depends on the process of a rule fire based on some condition; therefore, a condition and an action have a functional relationship since a condition “causes” an “action”.

- working (memory) accesses declarative (memory), procedural (memory), perceptual (buffer), and motor (buffer)
- learning (reinforcement) modifies rules
- motor (module) produces actions

For example, every instance of a CMC architecture is derived from a root class called Model. From the parent Model class it defines five major functional areas,
or Modules, being: Working Memory (WM), Declarative Memory (DM), Procedural Memory (PM), Visual, and Motor. Thus, all five of these functions are represented as Modules (or Components) within the root Model. As they are all considered functional Modules (with varied representation), they are tied together using only the attributes common to all of them through an abstract parent class named Module.

![Diagram of CMC Meta-model Modules](image)

The basic unit of information that is stored or processed by any CMC model is by chunks of collective information, and represented by the Chunks class. These chunks store one or more discrete units of information, however the CMC architectures use different terminology to describe and represent these units. All modules in the CMC contain short- and long-term stores of data, and in most cases, can only be accessed through a buffer, represented as a Buffer class in our meta-model. The buffers store chunks, and thus have a relationship represented as bufferchunk.
Working Memory in the CMC constitutes the current goal of the agent, and is a point-in-time snapshot of that agent’s focus and attention in the present moment. WM is a logical component of the CMC; however, in most architectures (such as Soar and ACT-R), WM is implicitly instantiated as soon as a model is created.

Declarative Memory in the CMC contains the “what” information that is known by the agent; this information is stored in Chunks. DM can contain an indeterminate number of chunks, and is represented by the $dmchunks$ relationship in the meta-model.

Procedural Memory in the CMC contains the “if-then” clauses, typically represented as Left-Hand Side (LHS) and Right-Hand Side (RHS). If all of the statements in the LHS match the contents of WM, the rule will “fire”, making changes to WM as defined in the RHS of the rule. A procedural rule is represented as a Rule class, each having a Condition (LHS) and Action (RHS), represented by their respective classes. Since each condition and action is made up of rules acting on the contents of module buffers, there is a relationship between them defined by $LHSbuffers$ and $RHSbuffers$, respectively. Note, however, that since rules can effect
multiple modules, a rule can be associated with as many buffers are there are modules.

Fig. 4.15.: CMC Meta-model Productions.

Motor, Visual and Auditory components are not well defined by the CMC, other than the assumption an architecture makes use of sub-symbolic processing to transform raw signals into a feature space that are supported by symbolic operations.

**Candidate Meta-Model Definition**

The first full Meta-Model can be used to formally describe and define the Common Model of Cognition using a standard class diagram. In Figure 4.16 we see such a class diagram of the main components presented in the CMC earlier. The
class diagram captures the relationship between parent and child classes through inheritance and abstraction, the constitution of classes using encapsulated attributes, and the relationship between classes through references and composition.

4.5.4 Metamodel Validation

Meta-model and cognitive architecture interface validation can be performed by using the Object Constraint Language (OCL). The most commonly encountered OCL expressions used during Model to Text transformations are Invariant Expressions, Initialization Expressions, Derived Elements, Query Operations, and Operation Contracts.

Invariant Expressions represent the integrity constraints of a model, requiring a context type (or namespace), and a body that results in a Boolean value. All instances that are members of the context type must always satisfy the conditions of the body. These are the most common of all expressions. Example invariants found in the CMC include the value of specific variables (clock rate, recall time, learning rate, etc.), or the lower and upper bounds of composition attributes such as the number of slots in a chunk.

Initialization Expressions check to see if the starting value of component attributes conform upon object creation. The restricted attribute can, however, change after object creation. An example initialization could be a flag indicating the use of learning applied to different modules, or perhaps a default buffer state that may be updated by a task model.

Derivation Rules, and their associated elements, specify the value an element takes based on the value and state of other elements in the model. A derived element may change values, but the value must always conform to a specific rule. These values are different from initialized expressions, as they must be valid through the life of the
Fig. 4.16.: First candidate EMF class diagram for CMC Metamodel.
object they represent. An example of a derived element could be the total number of cognitive cycles since model implementation, or perhaps the state of a module between operations.

*Query Operations* are a way to wrap functions designed to return some information to a calling user or program. These functions fulfill a similar role to traditional functions, and can be called by other validation rules. An example query in the CMC could be a list of sub-classes that implement an abstract interface, or a request for all buffers that contain a specific pattern.

*Operation Contracts* take on two different forms; Imperative, or Declarative. Imperative operations specify the format of the callable operation itself, such as the parameters and return types that are sent to, or received from a function. An example Imperative contract would be a list of variables and types that form part of the function definition. A Declarative operation is less specific about the operation itself, but holds conditions on the state of the model; pre-conditions must hold when the operation is first called, whereas post-conditions must hold when the operation is finished. An example Declarative pre-condition could specify that a motor buffer must state a motor module is free before motor action can begin, whereas the operation post-condition could be that the motor module action is in fact finished before the buffer flag is updated.

With these OCL statement types in mind, we can now review the list of CMC Architectural Assumptions (Appendix C), redefine them as either a hard or soft requirement, and attempt to specify them according to a formal statement using the Meta-Model graph. Viewing the Assumptions from this perspective, however, also forces us to divide them into two broad categories: those line items that describe the idealistic, philosophical, or qualitative characteristics of that aspect of the architecture (commonly known as a non-functional requirements), as opposed to the
operational, relational, or quantitative inputs, functions, and outputs of that aspect of the architecture (commonly known as functional requirements). Functional requirements are those that have an immediate effect on the system, and any changes will drastically impact or prohibit the proper function of the system. Non-functional requirements define the quality or attributes that define the standards of usability and effectiveness by which the system must perform.

**Structure and Processing**

The Structure and Processing assumptions describe the intended purpose and behavior of the entire architecture, independent of specific modules.

- A1: The purpose of architectural processing is to support bounded rationality, not optimality.

  This assumption states the Production System produces bounded rationality, and can be rephrased as requirement:

  - R1: The Architecture is not designed for optimality.

  Bounded rationality is a non-optimal characteristic that depends on generalized performance across task environments, and is therefore non-functional.

- A2: Processing is based on a small number of task-independent modules.

  This assumption, much like A1, states the Production System requires task-independent modules, and can be rephrased as:

  - R2: Modules are not customized for specific tasks.
This requirement, much like R1, describes a non-optimal characteristic at the module level, and is therefore non-functional.

Assumption A3 states “there is significant parallelism in architectural processing”, and has two sub-assumptions:

- A3a: Processing is parallel across modules

This assumption states Module processes communicate with other Modules, independent of a shared clock, and can be rephrased as requirement:

- R3: Modules communicate with one or more other modules.

Assumption A3a, sets a general default permission that all modules may communicate with one or more other modules without conditionals set by a shared clock. There are two ways to deal with communications in Role-Based Access Control (RBAC), either deny-by-default, or allow-by-default. We will soon see that a deny-by-default policy makes the most sense, and therefore want to set a default conditional on Collaborator relationships between Modules. Figure 4.17 illustrates the accessor relationships between architectures components defined by the CMC, and can be read as the access rights of the component in each row as it acts upon other components in each column.

A simple collaboration matrix, also known as an Access Control List (ACL) between objects and subjects would describe whether an object can read from, and/or write to a subject. In this case, all Modules are members of the matrix as Collaborators, however, not all of them are Callable from all other modules. As we will see later on, there are two special relationships in the CMC: the Production System has a special communications relationship with the Working Memory component (whether a centralized module or distributed buffers) as a central hub of activity. The second special relationship is the ability for Perceptual and Motor
modules to communicate directly with each other. All other communications must go through the Working Memory system. Therefore, it makes the most sense for each module to contain “Accessor” relationships with other Classifiers, such as Modules and Buffers, as well as the type of access.

This requires two OCL statements: a general Initializer for the access relationship, and a pre-condition that any caller can in fact call the `getter` or `setter` methods.

- **A3b:** Processing is parallel within modules.

This assumptions states that each Module can perform internal processing without a serial bottleneck or consideration to a shared clock with other modules; it can be rephrased as requirement:

- **R4:** Modules can perform asynchronous processing.

This requirement is a generalized affordance for the architecture to choose how to process information within each module, and is therefore a non-functional requirement.
• A4: Behavior is driven by sequential action selection via a cognitive cycle that runs at 50 ms per cycle in human cognition.

This Assumption is somewhat loaded, and holds two statements. First, it states the Production System selects a single action per cognitive cycle. Second, it states each production cycle runs at approximately 50ms per cycle. The connection between the first and second statements is somewhat unclear; is the Behavior being driven by action selection limited to 50ms, or is the action selection limited to one selection per 50ms, or one final action per 50ms? Readers familiar with the Architectures in question may already know the answer, but a formal system makes no assumptions. Therefore, we can be rephrase the first Assumption as requirement:

• R5: The Architecture selects a single action per cognitive cycle.

This requirement is a form of gating mechanism, that holds as long as action selection is performed serially. We will see later on that the interpretation of what constitutes an action is not the same between architectures, but at this point we can suffice by validating whether an action has been selected during a cognitive cycle.

The second Assumption can now be rephrased as requirement:

• R6: A single cognitive cycle runs at 50ms per cycle.

This requirement is quite clearly a functional requirement that can be directly specified and initialized at design time, as well as validated at run time. We can use two OCL statements to capture this requirement. First, we use an Initialization statement to ensure the cycle value is set to 50 (ms).

context Model:: cycle: Integer init: 50

Second, we can use an Operation contract to specify the length of time for a single cognitive cycle to run:
context Model:: step(): Integer pre: self.cycle = 50

The last structure and processing Assumption stated by the CMC is:

- A5: Complex behavior arises from a sequence of independent cognitive cycles that operate in their local context, without a separate architectural module for global optimization (or planning).

This Assumption contains two sub-statements. The First one states that sequential cognitive cycles operate independent of one another using a context (representing a current state); therefore, cognitive cycles in and of themselves do not have direct temporal access to the states of cycles that have already occurred, or will occur at some point in the future. This Assumption can be restated as requirement:


The second sub-statement in Assumption A5 says the Architecture does not have or use a module dedicated to global optimization or planning. This requirement can be restated as:

- R8: The Architecture does not have a module for global optimization or planning.

As we will see later, this requirement is somewhat difficult to enforce at the Cognitive Level using the rest of the CMC as an outline, as state tracking is performed by Working Memory in all architectures, and complex behavior is derived from task environments. However, any architecture that operates through the functional interfaces declared in the Meta-Model implicitly satisfy this condition.
Memory and Content

The CMC Assumptions about Memory and Content deal with the structures of memory modules, their communications paths, and the effect this has on global control. The first Assumption is:

- B1: Declarative and procedural long-term memories contain symbol structures and associated quantitative metadata.

Unpacking this Assumption can result in multiple requirements. First, we have a Declarative Long-Term Memory (LTM), restated as requirement:

- R9: Declarative Long-Term Memory contains symbol structures.

Second, we have a Procedural LTM, restated as requirement:

- R10: Procedural Long-Term Memory contains symbol structures.

Both of these requirements are permissive of one format of representation; namely symbol structures.

Assumption B1a contextualizes this statement with specific examples:

- B1a: ACT-R: chunks with activations and rules with utilities; Sigma: predicates and conditionals with functions; Soar: triples with activations and rules with utilities.

Let’s unpack this. Declarative LTM is represented in ACT-R as chunks, Sigma as predicates, and Soar as triples. An ACT-R chunk is an untyped list (usually represented as a Collection). Sigma predicates are a typed variable used in first-order logic, and Soar triples are typed variables connected (by default) in a graph structure. R9 can thus be restated as:
• R9: Declarative Long-Term Memory contains symbol structures consisting of one or more variables, alone or in combination.

context AbstractDeclarativeMemory inv: self.dmchunks->notEmpty()

Unpacking B1a further; Procedural LTM is represented in ACT-R as rules, in Sigma as conditionals with functions, and in Soar as rules. A rule can, for all intents and purposes, be redefined as a function (Then clause) triggered by a conditional (If clause). R10 can thus be restated as:

• R10: Procedural Long-Term Memory contains symbol structures consisting of rules.

Going back to Assumption B1, we also have associated quantitative metadata. Assumption B1a states Declarative LTM metadata in ACT-R consists of activations, and in Soar as activations. The Sigma example does not specify the metadata type. Thus, we can interpret this as requirement:

• R11: Declarative Long-Term Memory variables have a quantitative activation value.

context Chunk inv: self.activation <> 0

Going back to Assumptions B1 and B1a (yet again), we can derive Production LTM metadata in ACT-R consists of utilities, and in Soar as utilities. The Sigma example does not specify the metadata type. Thus, we can interpret this as requirement:

• R12: Procedural Long-Term Memory rules have a quantitative utility value.

context Rule inv: self.utility <> 0

Moving on to the next Assumption, we have:
• B2: Global communication is provided by a short-term working memory across all cognitive, perceptual, and motor modules.

Global communications here introduces the Working Short-Term Memory (STM) as a central hub, or gateway between modules. Also note the *cognitive* modules here is separate from *perceptual* and *motor* modules. Thus, we can deduce from this statement that Declarative LTM, Perceptual LTM, and possibly Working STM are of a *Cognitive* class type, whereas the Perceptual and Motor modules are of a *Non-Cognitive*, or perhaps *Peripheral*, or *Embodied* class type. Either way, we can restate the first Assumption as requirement:

• R13: Working Short-Term Memory provides global communications across all modules.

Going back to our discussion of RBAC and ACL definitions, Working STM here then must have both read and write access to provide this level service. We can thus add Working STM modules to our list of Accessors.

context WorkingMemory inv: accessor->exists(pm: ProceduralMemory)

Moving on to the next Assumption:

• B3: Global control is provided by procedural long-term memory.

This Assumption states that Procedural LTM contains the elements necessary for control of the next sequential action. We can introduce this as requirement

• R14: Procedural Long-Term Memory *contains* action selections.

Assumption B3 also contains two additional sub-assumptions:

• B3a: Composed of rule-like conditions and actions.
• B3b: Exerts control by altering contents of working memory.

The first statement quantifies the contents of Procedural LTM of rule-like conditions and actions. A rule, but earlier definition, is composed of a predicated conditional (IF) clause, followed by an propositional action (THEN) clause. We can, therefore, restate R14 as:

• R14: Procedural Long-Term Memory contains action selection rules.

The second statement, B3b, qualifies the Global control structure of Procedural LTM; action selection rules alter the contents of Working STM. This Assumption forms the crux of the Production System itself, as Procedural rules are the only element capable of acting upon Working STM structures. We can, therefore, assert this Assumption as requirement:

• R14: Procedural Long-Term Memory modifies Working Short-Term Memory.

The last Assumption about Memory and Content is:

• B4: Factual knowledge is provided by declarative long-term memory.

This statement, by itself can be restated as requirement:

• R15: Declarative Long-Term Memory contains facts.

This statement by itself does not say much, aside from the facts label applied to information stored in Declarative LTM structures. The CMC does provide a sub-assumption with examples:

• B4a: ACT-R: single declarative memory; Sigma: unifies with procedural memory; Soar: semantic and episodic memories.
We can derive that Declarative LTM in ACT-R is a single structure, Sigma uses the same structure as Procedural LTM, and Soar partitions Declarative LTM into two separate structures for two kinds of memory. This statement does not do much for standard assumptions, other than give examples of how the architectures are fragmented in their implementation. We can, therefore, conclude that Requirement R15 by itself is enough to satisfy Assumption B4.

context DeclarativeMemory inv: self.dmchunks -> exists(c: Chunk)

# Learning

Assumptions about Learning describe mechanisms and processes of learning in CMC architectures, as well as the data structures on which they rely. The first assumption is:

- C1: ACT-R: All forms of long-term memory content, whether symbol structures or quantitative metadata, are learnable.

This assumption mirrors that of B1; it applies to Declarative LTM facts (R9), Procedural LTM rules (R10), Declarative LTM activations (R11), and Procedural LTM utilities (R12). Here, learnable can be defined as adding new instances, or updating existing values. From this assumption, we can start with their equivalent representative requirements as:

- R16: Declarative LTM can add new facts.
- R17: Procedural LTM can add new rules.

These requirements are obviously functional, and translate to standard setter operations on their corresponding modules. The simplest OCL statement here
would be an Imperative contract describing the format of adding facts and rules, and updating their activations and utilities, respectively.

The next Learning assumption says:

- C2: Learning occurs online and incrementally, as a side effect of performance and is often based on an inversion of the flow of information from performance.

This statement contains four different assumptions. First, that learning occurs online. Online learning here means during run-time, as opposed to a-priori knowledge that can only be loaded during initialization. Second, learning is incremental. Incremental here suggests an incomplete representation of the required facts or rules required to achieve the intended goal state. Third, learning is a side effect of performance; this assumes there is a primary effect of performance, hinting at an external feedback mechanism. We see in the next Assumption that performance evaluation includes a reinforcement value. Last, learning is derived from an inverse flow of information from performance. We can restate this assumption as requirements:

- R18: Declarative LTM can update the activation value of existing facts.

- R19: Procedural LTM can update the utility value of existing rules.

The CMC text mentions an internal representation of experience, but this is not yet stated here. At this point we can assume a functional declaration that insists the setter methods (R16-R19) derived from C1 are *callable* during run-time, and accessible from within the model.

The following statement contextualizes these assumptions with:

- C3: Procedural learning involves at least reinforcement learning and procedural composition.
We can unpack this assumption into three distinct features. First, that Procedural rules utilities can be updated based on reinforcement values. Second, that Procedural rules can be added through composition; here composition can also mean compilation. Third, that Reinforcement learning is not the only type of learning available to Procedural LTM.

The first two sub-assumption from C3 are further contextualized:

- **C3a:** Reinforcement learning yields weights over action selection.
- **C3b:** Procedural composition yields behavioral automatization.

The first assumption, C3a, is directly identifying the product or outcome of the feedback mechanism; that is, *weights*. The weights are based on the perceived utility or value of the last action performed. These weights also constitute a result provided by one of many possible algorithms, that are computed using values supplied by either internal policy, or external signal. Both of these possibilities are extensions of assumption C2, and indicate a possible source of Operations performed through Requirements R18 and R19.

The second assumption, C3b, is directly identifying the product or outcome of composition (or compilation) of new Procedural rules; that is, *behavior automatization*. Automatization here is not qualified in the CMC text, however, it would be safe to assume this implies increasing levels of sophistication according to some external policy. This statement also contains an example as:

- **C3bi:** ACT-R: rule composition; Sigma: under development; Soar: chunking.

From this example, we know that Procedural LTM rules are created in ACT-R through *rule composition*, and in Soar through *chunking*, whereas Sigma does not learn new Procedural rules. Two things to note here; first the reuse of the term
chunk, but in a very different context as seen earlier in Assumption B1a (Declarative LTM symbols). The word chunk in B1a is used as a noun to describe a discrete piece of memory. The word chunking used here is a verb, indicating a process of composing new Procedural rules. The second item of interest is the lack of implementation in Sigma; the CMC text does not expand on whether this applies to the theory, or code, or both.

We can conclude that Assumption C3b describes a requirement that cannot be directly validated in the architecture itself, and therefore can not be restated as a functional requirement. We can also conclude that there are some explicit conflicts in terminology, and lack of completeness in the representative architectures.

The next assumption about Learning is:

- C4: Declarative learning involves the acquisition of facts and tuning of their metadata.

This statement is, for all intents and purposes, a restatement of Assumption C1, albeit more specific to Declarative LTM. The corresponding requirements (R16-R17) therefore, do not need to be restated based on this Assumption.

The final assumption about Learning is:

- C5: More complex forms of learning involve combinations of the fixed set of simpler forms of learning.

This assumption takes a similar form of C3b; it implies a gradual increase in sophistication for the learning mechanism without indicating specifics of how or where this happens. This assumption is, therefore does not require restatement.
Perception and Motor

The last group of Assumptions describes the *non-cognitive*, or peripheral modules, that sit between the core cognitive components and the external world. These assumptions, while not clearly defined by the CMC, are probably they most important as the CMC papers identified them as the most neglected. The first Assumption is:

- **D1**: Perception yields symbol structures with associated metadata in specific working memory buffers.

This statement can be unpacked. First, we see that Perception (modules) *yields* (or produces) symbol structures. This statement is very similar to that of C3a and C3b, however differs in source; the symbols are a product of the processing in the Perception Module(s), vs an indeterminate algorithm. Second, the Perception module(s) produce(s) metadata. Last, the Perceptual module(s) writes symbols and metadata values into a buffer in Working STM. We can restate this top level Assumption as the following requirements:

- **R20**: Perception creates symbolic structures.
- **R21**: Perception creates metadata variables.
- **R22**: Perception writes to the Perception buffer in Working STM

This Assumption also provides additional qualifying statements:

- **D1a**: There can be many different such perception modules, each with input from a different modality and its own buffer.
- **D1b**: Perceptual learning acquires new patterns and tunes existing ones.
• D1c: An attentional bottleneck constrains the amount of information that becomes available in working memory.

• D1d: Perception can be influenced by top-down information provided from working memory.

Statement D1a expands on the original Perception classifier, and includes three important assumptions: First, is the idea of plurality. Multiple Perceptual Modules can be instantiated. Second, that each Perceptual module can receive input from different sources. Readers familiar with the CMC architectures may recall examples including Vision, Aural, Tactile, etc. These examples, however, are not stated here, and are therefore not assumed. Last, each Perceptual module contains its corresponding buffer in Working STM. Recall from Assumption B2 that Working STM controls global communication, not the Modules that are represented. We can, therefore, assume from Assumption D1a (as stated) that Perceptual modalities cannot directly influence each other’s buffers.

Assumption D1b is stated in a similar format to Assumption C4, and hence can be broken into two statements. First, that Perceptual modules can learn (and hence store) patterns. Second, that Perceptual modules can tune (or update) these patterns. Both of these statements are an expansion on the symbol structures and metadata assumptions from D1. We can, therefore update Requirements R20 and R21:

• R20: Perception creates symbolic pattern structures of the environment.

• R21: Perceptual learning updates pattern metadata variables.

Assumption D1c introduces the idea of an attentional bottleneck. This bottleneck restricts the amount of information in Working STM. Considering initial direction of information flow (from input modality to buffer), we can assume the bottleneck
acts as a filter, and selects the most pertinent information and representation to be written to the buffer. This Assumption does not say where the bottleneck is situated, so we can assume it is in the Perceptual module, and can, therefore, introduce a new requirement:

- **R23:** Perception filters information to be placed in the Working STM buffer.

Assumption D1d updates our understanding of Working STM Control; the information in the Perceptual buffer can influence (write to) the input modality. This informs us of a unique communications path not seen in the cognitive modules. Assumption D1d does not, however, state where in the module this influence is directed, only that it is. We can state this Assumption as another requirement:

- **R23:** Working STM can write to the Perceptual module.

Requirement R23 is another RBAC rule that creates an accessor relationship between the Working STM and the Perceptual module itself (vs the buffer). This can be shown as an OCL statement such:

```
Context WorkingMemory
inv: self.writes->exists(pm: ProceduralModule)
```

The last Assumption is specific to the Motor module:

- **D2:** Motor control converts symbolic relational structures in its buffers into external actions.

This statement informs us of two pieces of information: first, that Motor modules read from the motor buffer, and second, that the Motor module writes to (or acts upon) the external environment. This statement does not specify how a Motor module affects the environment, only that it does. The CMC qualifies this Assumption with the addition of the following:
• D2a: As with perception, there can be multiple such motor modules.

• D2b: Motor learning acquires new action patterns and tunes existing ones.

Assumption D2a is a repeat of D1a, namely the indication of plurality. Assumption D2b is a repeat of D1b, namely, that Motor modules can create and update patterns of information. We can, therefore restate all of D2 with some confidence as the following requirements:

• R24: Motor modules read from the Working STM buffer.

• R25: Motor control creates symbolic pattern structures of the environment.

• R26: Motor learning updates pattern metadata variables.

• R27: Motor control converts symbolic pattern structures into external actions.

These requirements define the interfaces, content, and access control of the Motor Modules. From R22 we can create an accessor relationship between the Motor module and its corresponding buffer in Working STM, as well as the corresponding Operational contract.

Context Motor inv: self.reads->exists(m: WorkingMemory)

From R23 and R24 we can define a setter Operation acting on the contents of the Motor module.

Context Motor inv: self.nextaction->exists()

Context Motor::getNextAction(): String pre: self.nextaction->exists()

From R25 we can derive a similar Operational context between the Motor module and the external Environment.

The final list of requirements and their associated assumptions are now completed:
<table>
<thead>
<tr>
<th>Req</th>
<th>From</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>A1</td>
<td>The Architecture is not designed for optimality.</td>
</tr>
<tr>
<td>R2</td>
<td>A2</td>
<td>Modules are not customized for specific tasks.</td>
</tr>
<tr>
<td>R3</td>
<td>A3(a)</td>
<td>Modules communicate with one or more other modules.</td>
</tr>
<tr>
<td>R4</td>
<td>A3(b)</td>
<td>Modules can perform asynchronous processing.</td>
</tr>
<tr>
<td>R5</td>
<td>A4</td>
<td>The Architecture selects a single action per cognitive cycle.</td>
</tr>
<tr>
<td>R6</td>
<td>A4</td>
<td>A single cognitive cycle runs at ( \sim 50 \text{ms} ) per cycle.</td>
</tr>
<tr>
<td>R7</td>
<td>A5</td>
<td>Cognitive cycles operate in a local [temporal] context independent of other cognitive cycles.</td>
</tr>
<tr>
<td>R8</td>
<td>A5</td>
<td>The Architecture does not have a module for global optimization or planning.</td>
</tr>
<tr>
<td>R9</td>
<td>B1</td>
<td>Declarative Long-Term Memory contains symbol structures consisting of one or more variables, alone or in combination.</td>
</tr>
<tr>
<td>R10</td>
<td>B1</td>
<td>Procedural Long-Term Memory contains symbol structures consisting of rules.</td>
</tr>
<tr>
<td>R11</td>
<td>B1(a)</td>
<td>Declarative Long-Term Memory variables have a quantitative activation value.</td>
</tr>
<tr>
<td>R12</td>
<td>B1(a)</td>
<td>Procedural Long-Term Memory rules have a quantitative utility value.</td>
</tr>
<tr>
<td>R13</td>
<td>B2</td>
<td>Working Short-Term Memory provides global communications across all modules.</td>
</tr>
<tr>
<td>R14</td>
<td>B3(a,b)</td>
<td>Procedural Long-Term Memory Modifies Working Short-Term Memory.</td>
</tr>
<tr>
<td>R15</td>
<td>B4</td>
<td>Declarative Long-Term Memory contains facts.</td>
</tr>
<tr>
<td>R16</td>
<td>C1, C4</td>
<td>Declarative LTM can add new facts.</td>
</tr>
<tr>
<td>R17</td>
<td>C1, C3</td>
<td>Procedural LTM can add new rules.</td>
</tr>
<tr>
<td>R18</td>
<td>C1, C3, C4</td>
<td>Declarative LTM can update the activation value of existing facts.</td>
</tr>
<tr>
<td>R19</td>
<td>C2, C3b</td>
<td>Procedural LTM can update the utility value of existing rules.</td>
</tr>
<tr>
<td>R20</td>
<td>D1(a)</td>
<td>Perception creates symbolic pattern structures.</td>
</tr>
<tr>
<td>R21</td>
<td>D1(b)</td>
<td>Perceptual learning updates pattern metadata variables.</td>
</tr>
<tr>
<td>R22</td>
<td>D1(a)</td>
<td>Perception writes to the Perception buffer in Working STM.</td>
</tr>
<tr>
<td>R23</td>
<td>D1(d)</td>
<td>Working STM can write to the Perceptual module.</td>
</tr>
<tr>
<td>R24</td>
<td>D2</td>
<td>Motor modules read from the Working STM buffer.</td>
</tr>
<tr>
<td>R25</td>
<td>D2(a)</td>
<td>Motor control creates symbolic pattern structures.</td>
</tr>
<tr>
<td>R26</td>
<td>D2(b)</td>
<td>Motor learning updates pattern metadata variables.</td>
</tr>
<tr>
<td>R27</td>
<td>D2</td>
<td>Motor control converts symbolic pattern structures into external actions.</td>
</tr>
</tbody>
</table>

Table 4.1.: Functional Validation Requirements from CMC Assumptions.

The list of requirements in Table 4.1 provide guidance on the requirements and subsequent specifications captured in MOF diagrams and OCL expressions for formal generation and validation of the interfaces.

### 4.5.5 Architecture Interface

If we wish to use multiple architectures in an experimental framework, we will need to understand how to write code that interfaces with each of these architectures. Further, if we wish to conduct modeling experiments designed to test one or more of
the functional areas or regions of interest, we will need to write custom code for each API, with varying levels of support. This is a lot of (potentially exponential) work!!

|        | Procedural Memory | Working Memory | Declarative Memory | Perception | Motor | ...
|--------|-------------------|----------------|--------------------|------------|-------|-----
| CMU ACT-R | CMU-PM            | CMU-WM         | CMU-DM             | CMU-Perc   | CMU-Motor | ...
| CCMSuite3 | CCM-PM            | CCM-WM         | CCM-DM             | CCM-Perc   | CCM-Motor | ...
| Soar    | Soar-PM           | Soar-WM        | Soar-DM            | Soar-Perc  | Soar-Motor | ...
| ...     | ...               | ...            | ...                | ...        | ...     | ... |

Table 4.2.: Mapping Measurement Methodology to Research Questions.

Even if we abstract away from the specifics of each API, we still need a way to create individual architecture module objects so that they can collaborate with the other modules representing different functions in the same architecture. This abstraction necessitates an implementation that doesn’t require updating existing client code (that interfaces with each API) when adding or changing specific modules or whole architectures. Doing otherwise soon becomes a logistical nightmare with any change to the Common Model.

To deal with this type of abstraction we can rely on a common Creational Design Pattern known as the Abstract Factory. The abstract factory pattern is typically used when a program or process is dealing with multiple families of classes that deal with a common set of objects and interfaces; however, that program doesn’t (or shouldn’t) need to know which family of classes (with corresponding objects) are being used at any specific moment.

In the case of Common Models, the abstract factory pattern assumes we will begin by declaring a common interface for each distinct Module found in a Common Model architecture (i.e. Working Memory, Declarative Memory, etc.). Then, once we have this interface defined, we create (or implement) concrete variants of this interface for each of the unique architectures that claim to support it.
Defining Abstract Interfaces

The first step is to define the interfaces used by common modules. For example, all common model architectures store procedures in some form of Procedural Memory. We know that regardless of target architecture, we should be able to add new procedures, and specify the properties of the conditions and actions that make them work at run time. The details of how they are stored and processed by each architecture is below the scope of this interface, so as modelers all we want to know is how to perform these basic operations. Therefore, all common model interfaces can implement the abstract Procedural Memory interface (Figure 4.18). This process is then repeated for each of the modules that form a part of the Common Model.

As another example, say a modeler wants a Common Model factory to produce a Declarative Memory (DM) interface. The modeler shouldn’t have to understand how
ACT-R implements DM, or how Soar implements DM, or even which architecture they are dealing with at that time (although they certainly should) only that they are working with a Common DM module, and that it implements and supports a common function, such as “addDM()”, in some fashion (Figure 4.19).

Fig. 4.19.: Abstract Interface for Declarative Memory Factory.

Generating Interfaces for Abstracting Modules

The next step is to create an Abstract Factory, an interface that provides a list of creation methods for each of the modules that make up any one cognitive architecture. These methods return an abstract module represented by the abstract interface defined earlier, one for each module defined in the Common Model. In practice, these concrete (or implemented) factories are used to generate the common Module interfaces used by a program working with a Common Model. The client
program knows it can request and interact with these modules in the same way, regardless of the architecture that generated it.

![Diagram](image)

**Fig. 4.20.:** Creation Methods for Abstract Factory Modules.

**Factory Structure**

So how does this work behind the scenes? To the modeler, they all appear as abstract interfaces, however, the architecture-specific factories have to return an interface for the module type requested, and handle the respective underlying logic required to work with it, regardless of how it is implemented or managed “under the hood”, and without forcing the modeler to deal with architecture-specific APIs. The choice of architecture factory is typically done at run-time either by the modeler, a configuration file, or the development environment.

Note that while concrete factories return instantiated versions of their respective modules, the creation method called by the client must match the abstract factory version; so for example, an ACT-R perceptual module is only expected to collaborate
with other ACT-R modules from the same factory, and is not expected to collaborate with a Soar motor module (even if synchronized hybrids are possible).

The abstract creator methods in Figure 4.20 then become the resulting “front end” for users to generate new modules using any cognitive architecture factory that implements them, and is important so that client code that relies on the common model interface doesn’t have to deal with architecture-specific variants of each module interface.

4.5.6 Model-to-Text Generation

The last stage of the model tool chain is to take each model file and run it through a transformation so that a target interpreter (or cognitive architecture) can make use
of it. Generating this code requires three essential components: an input model representing the instantiated model components and values; a code template that serves as an outline of the destination file format; and a transformation engine that iterates through the model file, generating output according to the specification in the template (Figure 4.22).

Fig. 4.22.: Model to Text Generation

Model files to be transformed into another format are typically exported in the XML MetaData Interchange (XMI) format. XMI, an OMG standard built on the existing XML standard, is used for exchanging UML modeling data and objects between different tools. In our case, we can store the CMC meta-models in XMI, and import them into any other compatible tool. Figure 4.23 for example, is a raw text version of an exported CMC meta-model. Readers already familiar with XML will notice that every component in a class hierarchy is nested in opening and closing tag brackets. In addition to the standard XML structure, these tags also include some important information that is specific to MetaData Interchange, such as the definition of structural features like Attributes and References, and structure-specific meta-data such as the upper bounds of the composition relationships covered in Section 4.5.3.
The OMG also defines a standard for writing templates to turn XMI meta-models into another format, allowing any transformation tool that conforms to the MOF Model Transformation Language (MTL) specification to parse and transform model files according to the supplied MTL template. MTL generators are typically used in practice to transform model files into source code for a specific language (C++, Java, Python), or documentation such as technical manuals. In this case, I am using MTL to create templates to transform a CMC meta-model (exported as XMI) into source code and documentation for the CMC frameworks.

It is important to reiterate the fact that the (current) purpose behind these templates is not for using the CMC models to generate or compete with existing architectures that have already be designed and compiled. Rather, we are using this process to generate the Python interfaces that will allow us to supply a standard configuration across all these architectures, as well as generate user model templates for each of the target Python ACT-R, LISP ACT-R, and Soar architecture interpreters, depending on where and how test specifications are defined by each architecture (i.e. subject to availability).
Abstract Factory Code Generation

The MTL templates used to generate the factory interfaces are structured to generate a Python module with corresponding classes and function definitions. There are two components to this template that need to be explicitly mentioned, as they are part and parcel with validating a generated interface: namely the use of abstract function decorators, and overriding Python’s built-in class types with new metaclasses that conform to the MOF standard.

First, those readers familiar with Python will notice the inclusion of Abstract Base Class (ABC) libraries, associated function decorators (@abstract), and pre-generated function definitions that, when interpreted at run-time by the Python interpreter, will ensure that only implemented (vs abstract) modules of the same factory type will work with one another, and only if they match the meta-model specification.

```python
@abstractmethod
def createModel(self) -> AbstractModel:
    pass
```

In this example, the `createModel()` method, just like the factory class, can be used by any calling client expecting a corresponding instance of that module regardless of the type of factory the client is dealing with (figure 4.20). The generated Abstract Factory interface includes one creator method for each of the CMC components specified in the meta-model, being: Model, Working Memory, Declarative Memory, Procedural Memory, Perception (specifically Vision), and Motor Action.

The abstract Model serves as the container of all other components, and is the main point of entry when interacting with sub-components. Some high level languages encourage the use of public `getter` and `setter` methods (to request or define attribute values, respectively), making everything else private (or callable)
from other libraries and programs. I decided against a hard line approach in the initial design as it was too early to know whether every cognitive architecture could in fact interface through specific classes or methods. In the end, it turns out they can, however to do so required some creative restructuring of their corresponding APIs, as explained in the next few sections.

In the next example, the AbstractWorkingMemory(ABC) is the interface of the Working Memory module. All other modules can interact with each other, but proper interaction is possible between other modules returned from this architecture factory. The abstract method addWME() is the function interface to be implemented by supporting architectures in their own concrete implementation of this factory.

```python
class AbstractWorkingMemory(ABC):

    @abstractmethod
def addWME(self, collaborator: AbstractModel) -> None:
        pass

    @abstractmethod
def removeWME(self, collaborator: AbstractModel) -> None:
        pass
```

Also, in this example, both functions also indicates that implementations can collaborate (or work with) any implemented Models from the same factory, without worrying about exactly how that class has been implemented, so long as it conforms to (and makes use of) the functions and attributes defined in this interface. Each of the modules returned by the factory creator methods follows a similar pattern to Working Memory, with the exception of the Model class, which includes extra functions required for creating, loading, configuring, and cycling model experiments.
PyEcore Metaclass

The MTL templates used to generate the factory interfaces also make use of the “PyEcore” library, allowing modelers to create and manipulate model instances using classes, attributes, and relationships that reflect the imported meta-model, all in the same Python environment as the interfaces. The PyEcore library provides a very important feature that is worth mentioning here: it uses a new (as of Python 3.6) Python meta-classes feature to redefine (and override) default type values, in order to implement a language-agnostic typing system known as “EClass”, to conform to the EMOF typing system.

The MTL template for generating these interfaces also makes use of the Object Constraint Language (OCL) so that every class instance created by this generator only uses EClass attributes specified in the model, ensuring instantiations are maintained and validated against attribute types declared by the XMI definition. In a nutshell, our theoretical meta-models implemented as code are valid end-to-end (M3 to M1), underpinning the MDD principle of correct by construction.

User Model Code Generation

User Model generation (at MOF layer M0) involves a mixture of elements involving both of Newell’s Rational and Cognitive Bands. Task models, by their nature, require knowledge specification, and are by definition out of scope for a Common Meta-Model of Cognition, as task-specific knowledge resides at Newell’s Rational Band. That being said, we can still use our meta-model as a guide to ensure model files that specify settings for components at the Cognitive level are included in their respective templates.

Take for example, a Lisp ACT-R model file header:
Fig. 4.24.: PyEcore Classes by MOF level (Aranega, 2017).

(clear-all)
(sgp: esc t: 1f .05 : trace-detail high)
(define-model test-agent)
This header file instructs ACT-R to perform some preliminary clean up, names the model, and sets the general parameters for sub-symbolic computations (:esc) and retrieval time latency (:lf). We can do a similar setup in Soar as well.

```
agent−name = test−agent
watch 5
stats −t
```

We can set a watch value and turn on per-cycle tracing. So far so good, but we want it to do something useful. The next part involve setting the goal (or focus) so the model kicks off in an initialized state. For example, consider a simple task of counting from 1, 2, ..., n. represented here as a diagram using CMC meta-model rules. Figure 4.25 illustrates a tree view of a production rule that begins a counting sequence. Each node in the tree references an implemented EClass specified earlier in the CMC Meta-Model, but as an instance of a model with instructions.

```
(goal−focus first−goal)
(p init−agent
 =goal>
  start =numl count nil
⇒⇒
 =goal>
  count =numl
  +retrieval>
  first =numl)
```

In ACT-R, we start the counting process by setting our focus (goal) buffer to initialize, read in the starting numbers, and queue up the next number in the sequence.

In Soar, the propose keyword is used to set up top (first) state (not shown here for brevity), create a new operator, and assuming this rule is selected for firing, set our working memory elements to increase on the next cycle.

```
sp {topstate*apply*init−agent
```
Fig. 4.25.: Architecture-Agnostic Production Tree.

\[
\begin{aligned}
\text{(state } <s> & \, ^\text{superstate nil} \\
& \, ^\text{operator.name init-agent}) \\
\rightarrow \ \\
( <s> & \, ^\text{name test-agent} \\
& \, ^\text{current-increase 1} \\
& \, ^\text{current-number 0} )
\end{aligned}
\]

So far so good. What about adding memory? In ACT-R, we define a memory module and add the memories by chunk:

\[
\begin{aligned}
\text{(add-dm} \\
(a & \text{first 0 second 1}) \\
(b & \text{first 1 second 2}) \\
\ldots \\
(f & \text{first-goal start 1 end 10})
\end{aligned}
\]

In Soar we can do something similar, using an attribute-value structure.
Both ACT-R and Soar model files can be broken into sections representing instantiated components of CMC modules, such as: global parameters for rule firing, activation, and learning rate; declarative memory chunks and relationship; and procedural actions to modify the contents of working memory. These are the meta-model features that we want to abstract away from the model files into a Meta-Model File that can describe (at a high level) the general layout of the model, and yet still remain architecture-agnostic.

We use an MTL file with sections corresponding to each of these functional areas, and map them to their respective definitions, accordingly. This task, as all others, requires some level of general configuration, followed by more specific values depending on the modules employed.

Defining productions for a specific task works for very simple task models, however, differences in knowledge representation and processing for each architecture makes a like-for-like generation difficult without some modeler intervention. Consider a user model of a tree structure as depicted in Figure 4.26 containing various elements defined by the meta-model.

Where the model files differ is in relation to the underlying implementation details unique to each architecture. For example, memory in ACT-R is represented by chunks with activation and rules with utilities; when used in production rules, there is a direct mapping between the Left-Hand Side (LHS) conditional, and Right-Hand Side (RHS) activation that is considered for both operation evaluation and selection.
In contrast to this, the Soar architecture represents memory as triplets with associated activation, rules, and their corresponding utilities for LHS activation, with two corresponding RHS rules; one is for a distinct evaluation phase, and the other is for a distinct selection/action phase. This implementation-specific representation of the LHS conditional and RHS evaluation-selection-action dichotomy would require an ontological mapping and transformation, such as the one implemented in Herbal (Paik et al., 2009).
An MTL template for the ACT-R and Soar architectures can define simple initialization routines using the same structured model definition, however, there are discrepancies between how chunk contents, or slots, are ordered according to Soar-specific triplets of Object-Attribute-Value, covered in greater detail in Section 5.2.

4.5.7 Summary of the Meta-Model

In conclusion, this section described the first main contribution, a Conceptual framework based on MDD to define a Meta-Model of Cognition, the constraints for validation, architecture interface definitions, and the generator for turning meta-model definitions into usable code. The next step, is to map each cognitive architecture to the interface specification, and interaction with external virtual environments.

4.6 Mapping to a CMC Framework

This section describes the second main contribution, a mapping and implementation of Soar, and two versions of ACT-R, to the CMC interfaces generated by the meta-model.

4.6.1 Processing Cycle

The cognitive model processing cycle provided by Metaverse makes a strong assumption that all agents are interfacing with some kind of (virtual) environment, regardless of task complexity. Each step of the cognitive cycle requires an exchange of information between the agent and the environment in the form of intentional
actions and sensory perceptions, while synchronizing the clock cycle between agents and environments.

Fig. 4.27.: Interaction between Agent and Environment.

The processing cycle can be broken down to six general steps:

1. Get state from environment.
2. Present state to agent perception.
3. Trigger cognitive cycle in agent.
5. Present motor action to environment.
6. Trigger cycle in environment.

Agent actions and environment observations are each inspected for a done condition before passing it along for the next cycle. If the environment is done, or the experiment has reached the step limit (whether default or specified), the trial counter will increment, the environment will be reset, and initial observations prepared for the next trial (if applicable).
Environment State

When a virtual environment is initialized or reset, it will generate an initial observation or starting state. This must happen first, so that agents can (eventually) generate a corresponding action from the observation.

Agent Perception

The observation is retrieved from the virtual environment, and stored in the model in whatever format is presented by the environment, which can be compared to the raw signals presented to our perception before any form of pre-processing takes place.

Cognitive Cycle

The next step involves triggering a cognitive cycle in the agent. If the agent is in an initial state, the first cycle will likely only involve some combination of transducing the input signal into a symbolic representation capable of triggering a production or decision in a subsequent cycle.

Agent Action

Once the cognitive cycle has completed, the experiment will request the next action from the agent’s Motor module. The default agents provided in the examples have default actions corresponding to each environment, depending on what is allowed.
Environment Action

The action returned by the agent is presented to the test environment. The experimental framework does not perform any assertions on the action to see if it is valid, as this is the responsibility of the environment.

Environment Cycle

The last step is to trigger an action cycle in the environment. Each environment is tuned to advance one frame. Considering the virtual environment frames are synchronized with the cognitive cycle (50ms), the environment will advance at approximately 20 frames per second of simulation time. It is important to distinguish between simulation and real time, since both agent and environment will advance together as fast as the underlying medium will allow. The default experiment cycle includes a \texttt{sleep()} statement to pause the thread for 50ms of real time, so it appears as if both agent and environment are actually running in real time. The modeler can increase or decrease this speed as they see fit.

4.6.2 Integrating CMU Lisp ACT-R

In order to present a unified interface to a modeler, each of the target architectures had to be mapped to the logical components of the generated abstract factory interfaces. CMU ACT-R presents most, but not all, of the supported functionality through a single Python module (aptly called \texttt{actr.py}). The module includes some functionality that is not directly related to specific cognitive functions, including a connection manager to keep track of connection between the client system and the dispatch service, a generic message passing service for simple commands and parameters, and a command monitor interface for registering
callbacks based on specific conditions. Most of these services either run in the background, or are called as part of setup and tear down of a session with the dispatcher, and are not directly used by the client.

The largest portion of the Python interface module is devoted to functions that have been mapped to equivalent functions supported by the Lisp interpreter as articulated in the ACT-R Reference Manual (v7.13). The Python interface published at the time of this writing was somewhat mixed up as function calls appeared in no particular order indicating they were implemented in a somewhat haphazard manner over time based on some other conversion requirements. I went through the reference manual, and organized the function calls by reference manual section so that future users and developers would have an easier time lookup up corresponding function calls.

CMU ACT-R supports the creation of external experimentation windows including drawing buttons, lines, shapes, as well as emulated peripheral devices like a mouse and keyboard. All of these supplementary features are called directly through the Python interface, so the Model class in the CMC framework is the most appropriate place to hold references to these types of extra features. The Model class initialization currently has placeholders for these features, and are callable from the Perception and Motor classes depending on the type of experiment.

Loading Models

The CMU ACT-R dispatch service must be running before any remotely managed model can be loaded and executed. The dispatch service can be launched manually via command line (run-act-r.command), and by default loads the Tcl/Tk GUI interface as well. I created a utility for the CMC framework to check for running instances of the dispatch service, and to create a new process if it isn’t. The CMU factory exposes
a `load()` function that takes a path to the ACT-R user model file, and passes this on to the dispatch service to instantiate a new model from a Lisp model file.

![Diagram of Lisp ACT-R Model Initialization](image)

**Fig. 4.28.:** Lisp ACT-R Model Initialization.

When a new Common Model is created by the factory, it makes a call to `__init__()` to initialize the ACT-R model instance. The Common Model then calls `reset()` and `reload()` to purge any old data associated with the instantiated model instance (in the case of a series of trials or runs), and reloads the original settings specified in the Lisp model file. The Common Model then creates a local instance of each of the concrete modules defined by the CMC (being Perception, Working Memory, Declarative Memory, Procedural Memory, and Motor).

**Production System**

While most of the function calls necessary to initialize a new model are exposed to the client some of the more useful features were missing or unsupported. For example, the Lisp interpreter allows a modeler to define a new model, add memory,
specify productions, and launch the model as expected. Clients can define meta-model options such as global variable settings, and interact with pieces of memory, however the core components required to define a new model and production memories were undocumented. I searched through the Lisp dispatcher, begun implementing my own rudimentary parser for model definitions, and contacted the lead developer of ACT-R at CMU; indeed the feature has not yet been implemented in the dispatch service, and writing my own parser simply wouldn’t work for anything more than the simplest (i.e. useless) productions, requiring all external clients to pre-load the Production System rules through model definitions in a traditional ACT-R model file.

The CMC Model interface presents a `step()` and `run()` method to support batch runs of pre-defined experiments, or stepping through experiments one cycle at a time. CMU ACT-R does have a `run-step` command in the Lisp interpreter, however, this functionality has not been exposed through the dispatcher. This forces the model client to use other interrupts, such as specific event types, or pre-determined execution times, to ensure the model stays in step with an external environment. The CMC Model `step()` function calls `actr.run()` with a default of 0.05 seconds (50ms) of model time to simulate one cognitive cycle per the CMC specification. Note that invoking a model run multiple times does not initialize a new model with every run (as originally anticipated); the sample output indicates the model halts when it hits the supplied cycle time (0.05), and does keep increasing between steps:

```
CLEAR–BUFFER RETRIEVAL
0.050 DECLARATIVE start–retrieval
0.050 PROCEDURAL CONFLICT–RESOLUTION
0.050
Stopped because time limit reached
step: 1
0.100 DECLARATIVE RETRIEVED–CHUNK B
0.100 DECLARATIVE SET–BUFFER–CHUNK RETRIEVAL B
0.100 PROCEDURAL CONFLICT–RESOLUTION
0.100 PROCEDURAL
```
ACT-R output follows a standard format of [clock] [buffer] [command] [options], where clock is the current model time, buffer is the module being evaluated or acted upon, command is the action being carried out, and sometimes optional fields indicating the attributes or values being acted upon. Readers will notice that many steps occur during the same clock cycle, indicating a mixture of near-instantaneous and parallel processing within and between modules, depending on the activity in question. Working with buffers in Lisp ACT-R uses the same syntax as specifying chunks for Declarative Memory, and are quite flexible in order and presentation; this saves the modeler from having to understand the underlying data structure before actively working with it, either directly through the model file, or indirectly through the Lisp CLI or Python API calls. Comparing chunk slots in WM uses similar predicates, however, in-line modification of WMEs requires a separate eval or bind function that may not execute as intended. This requires basic manipulation of primitives to be done in DM.

The CMU Model production system permits interruption of the cycle by specifying a condition to halt or pause. Supported conditions include: a specific clock cycle (in model time), a registered event, or a function call. The current version of the Common Model that I created takes advantage of the Lisp ACT-R mp-queue-count command that returns a count of the number events that are left to process, to keep track of runaway models that are still being engaged by the environment or experiment monitor despite a lack of matching production rules.

Each time the Common Model step() function is called, it makes a call to Lisp ACT-R run() for 0.05 seconds, requests the number of outstanding events, and sets the done flag if events goes to 0.
Fig. 4.29.: Lisp ACT-R Model Cycle.

**Declarative Memory**

CMU ACT-R loads declarative memory elements during the model load phase. These can be specified in either the model configuration file, or by calling `add_dm()` directly through the Python RPC interface. The Common Declarative Memory interface specifies an equivalent `addDM()` function and acts as a wrapper for the ACT-R interface.

Unlike Soar, the ability to add to DM is turned on by default in Lisp ACT-R, and we do not have to couple it with a learning setting; both aspects are something of a convenience, and have little impact on overall functionality.

The DM module in Lisp ACT-R has a number and variety of settings that can be used to alter the underlying performance and operation; some settings are available as callable commands through the API, however, the vast majority are only accessible during model initialization (via user model file), or directly through the Lisp CLI. The ACT-R reference manual divides the parameters into four
categories: basic functionality, activation equation functionality, and activation equation parameters. For example, the :esc parameter enables sub-symbolic computing generally, and spreading activation can be altered by the :mas parameter specific to DM requests.

**Perceptual Interface**

Perception in ACT-R can be modeled in different way depending on the level of fidelity required. The CMU Lisp version uses a visicon module capable of performing rudimentary graphical functions and calculations, with capabilities specific to modeling attention, focus, eye tracking, among others. These features are not defined in the CMC, as they are specific to the kinds of behavior experiments the original ACT-R was designed for. Visual stimuli, and corresponding representation, is still articulated by the modeler assuming the representations have already been defined or extrapolated elsewhere. This is, for lack of a better definition, where things get a little murky.

![Fig. 4.30.: Lisp ACT-R Perception Interface.](image)

The preferred method of transduction between raw external signals, visual pre-processing, and internal representation is not clearly explained by the CMC. This makes sense, as the finer details of how visual pre-processing in V1 to V4 cortices
of the occipital lobe is performed, or how dorsal and ventral tracking (what and where) of external objects influences representation, are mostly situated at the neural or implementation layer below the cognitive layer. The problem of transduction, then, is left to the end user (or modeler) and necessitates some familiarity with the problem domain and perceptual tools made available to the model architecture. The Perception module provides a setObservationSpace() method that accepts a map of key-value pairs, one for each feature dimension. This map is subsequently used to name and identify all chunks that deal with observations. If the modeler does not provide an observation space mapping, one is automatically generated using generic incremental format \( obs_n \).

During presentation of the observation to the Perception module, the experimenter calls an update() to the Common Perception module with the value of the observation. The first part of this call invokes a transduction of the supplied
observation, to deal with environment-specific feature spaces. A chunk list is prepared using either a default map, or one provided by the modeler, and appended as a name-value pair. The resulting chunk string is then sent to the ACT-R dispatcher to initialize or update the goal (focus) buffer, assuming that the supplied production rules will use the same chunk names.

**Motor Interface**

The motor interface in ACT-R is based on the assumption the model will install one or more devices that act as an virtual peripheral between the model and the environment. ACT-R requires registration of an event that can catch and handle specified motor responses such as key presses, cursor movements and mouse clicks. The CMC framework Motor interface registers a call back function with the Lisp dispatcher so any time a key is pressed by the ACT-R model, the registered python function is called by the dispatch service with the expectation the registered function will handle external events. The CMC Motor function is configured to watch for specific key presses and applies minor assertions depending on the target environment, and then stores the action returned by the dispatcher as the *next action* to be retrieved by the Metaverse experiment.

The Motor module accepts a definition of the action space through `setActionSpace()`, being a vector of `Space` objects that conform to the Gym specification of valid feature spaces. Unlike the perceptual Spaces, motor action spaces are fairly simple, usually a Discrete list of integers between 0, 1, ..., n.

Upon initialization, the Motor interface registers with the dispatcher through `add_key_monitor()`, calling `monitor_command()` to register `respond_to_keypress()` as the target function to be called whenever the ACT-R motor module issues a corresponding key press command.
When the dispatcher receives a key press event, it call the registered `respond_to_keypress()` function, which accepts the name of the model and the
corresponding key. In the absence of a viable mouse click event handler, it will also check for a pre-defined unused key to initiate a click event in the environment.

**Issues and Opportunities**

The process of interfacing with the Lisp ACT-R dispatch service is relatively straightforward. The Python module supplied for remote procedure calls, while somewhat disorganized, does provide a comprehensive selection of system calls for the majority of operations. Many of the commands exposed through this API are simply wrappers for the text equivalent, and pushed through a generic `call_command()` function to be processed by the dispatch handler.

I was hoping this generic call could be used to take advantage of functionality not exposed through wrapper functions, however, there are some limitations to the parsing and processing abilities of the `call_command()`. For example, many of the commands used in task models loaded from a file have equivalent commands in the Python API, such as general parameters, module-specific parameters, and calls to add and modify chunks in the goal and declarative memory buffers. However, there is no option to add or modify production rules using this API. I attempted to overcome this issue by passing an entire production rule as a serialized string, with little success. An inspection of the command processor used by the dispatcher suggested it is not capable of tokenizing or parsing complex nested command structures; a conversation with D. Bothell at CMU confirmed this type of functionality is not supported, and Procedural Memory contents must be loaded from a pre-generated model file.
4.6.3 Integrating CCM Python ACT-R

The Python ACT-R system created at Carleton Cognitive Modeling Lab is a reconstruction of the original ACT-R theory in Python. This version of ACT-R is offered as a Python package called \texttt{ccm}, and contains the core ACT-R theory and functionality sorted into logical modules.

![Python ACT-R Class Structures](image)

**Fig. 4.34.:** Python ACT-R Class Structures.

### Loading Models

The Python package contains a core library that is used to instantiate each part of a task model. The \texttt{ccm} package is structured such that every module or component inherits from a generic \texttt{ccm.Model} parent class (Figure 4.34). This generic Model class serves as the vehicle for converting every object and function into a traceable object for statistics and reporting. When a new user model is instantiated, all function definitions that match the pattern of a production rule are registered as potential productions for later use during run time. These functions are also modified with custom attributes to support metric gathering utilities for event scheduling and logging. This approach makes the suite flexible to use as a
self-contained framework for testing models, but does add some complexity and
unintuitive overhead when trying to debug production rules, or couple it with
external or unsupported environments.

Creating a new agent with Python ACT-R involves creating a class that inherits
from the core `ccm.ACTR` class, itself being an extension of a generic production
system. If the user wishes to make use of certain logging features, or provide
callback support for environment-specific event handlers (i.e. motor actions), the
agent needs to be attached to a user-defined environment wrapper that inherits
from `ccm.Model`, the wrapper class required by all components of a CCM model.
Loading configuration data and initializing sub-components in Python ACT-R is
relatively straightforward, as this is done through the same Python instance as the
rest of the agent and environment objects that form a part of the CCM model. The
Metaverse framework takes advantage of this by instantiating any necessary
modules directly in the CCM model, either through the CCM environment or the
CCM agent as required.

**Production System**

The Python ACT-R production system is initialized and loaded through a form of
function wrapping and conversion. When a CCM model is first initialized, it parses
through all member functions belonging to child classes of `ccm.Model` (the agent
being one of them), and provided they match the format of a production function,
are converted and added to a dictionary of production functions. Stepping through
each cognitive cycle is done by calling `run()` on the `ccm.Model` environment. The
Metaverse example agent defaults to a cycle value 0.05 (50ms), corresponding to the
standard CMC cognitive cycle. Each time the Common Model `step()` function is
called, the Metaverse model will \texttt{update()} the Common Motor module as explained in the corresponding Motor Interface section.

\textbf{Working Memory}

Working Memory in Python ACT-R is represented by a goal (or focus) buffer, and updated with a chunk string to allow the agent to keep track of the current state. Chunks in Python ACT-R are simply passed as a string, usually in the form of \texttt{[name];[value]} pairs. The sample agent, in this case, only has to provide a wrapper to pass the string supplied to \texttt{addWME()} along to the Python ACT-R agent’s goal buffer.

\textbf{Declarative Memory}

The example Metaverse agent loads working memory elements by directly calling \texttt{addDM()} while defining the experiment. Calls to \texttt{addDM()} work in a similar fashion to Working Memory, however instead of sending the string to the goal buffer, it is passed along to the memory buffer.

\textbf{Perceptual Interface}

The Python ACT-R perceptual interface is equipped with a visual module for task-specific tests that mirror the Lisp ACT-R Tutorial Units. The current implementation of the CMC meta-model maps the chunk names corresponding to the target test environment to an internal visual buffer representation, or pattern. This pattern then goes through a similar transduction with corresponding map to a perceptual buffer, and is updated with the new observation string at each cycle of the experiment.
Motor Interface

The provided ccm.Motor class comes with a pre-defined press() function that is callable from converted production rules. When press() is called, it yields the function (to resume in place at the next call) to simulate the the successive motor delays that form part of the core ACT-R theory (inspired by Fitts’ Law). The end of the press() function makes a call back to a key_pressed() handler that (it is assumed) has already been defined in the user-defined Environment(Model) class.

The CCM Motor module presents a key press function, and much like the CMU Lisp ACT-R, calls an external function that is in the ccm.Model environment in which the ccm.ACTR agent resides. The update() function called from the factory’s Model.step() retrieves the action value from the ccm.Environment, stores it as the last_action, and returns it as the next_action value.

Issues and Opportunities

The original ccm ACT-R package was written in Python version 2.7. At the time of this writing, many Python 2 libraries were being deprecated without ongoing support in favour of the newer Python 3 development standards. The version used in this research (forked in 2018) went through an automated migration from Python version 2 to 3, and still exhibited bugs related to changes in various supporting libraries unrelated to the ccm implementation of ACT-R. I ran the updated version through the unit test cases to confirm all core functions are working as intended before generating evaluative results.

It does appear, however, that the forked version has not been maintained or modified much since original development and publication, so there is a chance that
any recent changes to ACT-R theory (after v6) have not been incorporated into the Python ACT-R version. The models based on the ccm version did encounter an error with the syntax accepting null “name:?” pairs without a value; however, this is valid syntax in the original Lisp ACT-R.

Mandatory encapsulation of Model classes to run an agent is somewhat cumbersome and adds an extra layer of complexity when trying to couple with external environments.

The production file does not have to be in the same file as the main ACT-R agent; the Metaverse framework takes advantage of this by loading production system functions dynamically at run time using Python’s `importlib` library, indexing each function, and adding a pointer to the CCM agent before it is scanned for production functions. This approach allows future modelers to specify (or generate) model files outside of the ACT-R agent class.

Production functions are not treated as standard Python functions; once they have been processed and converted by the Model wrapper, calling standard library functions can cause unexpected behavior, or crash the agent entirely through unhandled exceptions.

The `ccm` vision system — Simple Operating System (SOS) — follows the same design pattern as the one in Lisp ACT-R, and allows the user to create representations of environment objects that are processed internal to the agent using built-in utilities. This functionality is a recreation of the original CMU ACT-R’s `visicon` module, however, is not used by the example agent as it is outside of the scope of the current CMC definition.

Motor is like the CMU Lisp ACT-R, much simpler, but adds the hand and finger movements as static delays through sleep timer. This level of fidelity isn’t strictly necessary. The individual values aren’t based on a computed formula, rather they
hard coded. This is a very straightforward method of implementation, however the lack of context might break down if testing other theories of motor control.

### 4.6.4 Integrating Soar

The Soar cognitive architecture is accessed through the Soar Markup Language (SML), an interface that processes commands in XML format. The SML interface is originally written in C++, and exported to other languages (Java, Python) via SWIG (swig.org). The Metaverse Soar factory interfaces with the Python SML library generated by SWIG.

Creating a Soar agents involves creating an instance of the Soar Kernel, and then asking the kernel for an agent.

![Soar SML Interface](image)

Fig. 4.35.: Soar SML Interface.

Creating a new model in Soar begins with starting the Soar kernel, which runs as a separate thread. The Soar kernel serves a similar purpose as the Lisp ACT-R dispatcher, allowing a user to create and coordinate between multiple agents and task models running in different threads. Agents created through the Soar kernel also
require an external print handler to answer and display command results, and for registered callback functions to handle agent or task model commands.

There are two kinds of connectors depending on level of granularity and control required for working with the Soar kernel; using an AgentConnect wrapper, or directly calling the SML Python wrapper. The AgentConnector class does most of the work for you but makes some assumptions about the tree structure in working memory, limiting the specificity of the elements and state representation. The AgentConnect class acts as a wrapper to handle some of the subtle (and potentially troublesome) nuances of setting up, configuring, and shutting down kernel and agent instantiations. The Metaverse examples implement a SimpleConnector class, to facilitate communications between the example tasks and the SML interface compiled for Python. Alternatively, a client can make direct calls through the SML interface, which supports more specificity but requires additional checks against data types being supplied, updated, or requested from working memory so as not to cause an unhandled exception.

The Metaverse examples demonstrate both forms of invocation. The psychometric examples use a custom SimpleConnector class to handle basic agent connections, model initialization, and working memory updates for each clock cycle, whereas the Gym examples make direct calls to the SML interface.

**Loading Models**

Loading a Soar model can be performed one of two ways. If using the AgentConnector class, a new agent is instantiated using a configuration file that contains the global parameter settings, and the name of the .soar file (with productions) to be sourced. Alternatively, the client can make individual calls
directly to `agent.ExecuteCommandLine()` for each of the global parameters, and source one or more `.soar` file(s) directly.

Soar model files assume the client has already specified any environment-specific mapping between perceptual values and working memory elements that represent them. If using a simple connector (Psych), the agent loader adds an agent to the connector, removes any existing WMEs of the same name (a form of reset), and then populates the WME graph with new entries. The complex connector (for OpenAI Gym and StarCraft) requires a user-specified map, and calls `create_input_wmes` to initialize WMEs using the provided map.

**Production System**

Soar production rules are typically loaded from a model file. The SML interface does not currently provide a method of defining these from the calling client; as with the ACT-R architecture, researchers have to define these productions ahead of run time to use Soar.

The decision-action cycle in Soar, unlike ACT-R, is explicitly broken into three distinct *phases* that centers around the use of *operators* as a vehicle for problem solving; an intrinsic part of Problem Space Computation Theory on which the Soar production system is based. The three phases are operator *proposals*, then operator *selection*, and finally operator *application*. Every production rule, regardless if used for state elaboration, operator proposal, or operator application, is formed as an *If-Then* statement. What classifies a rule as proposal or application (aside from following naming conventions) would be the inclusion of an *operator* parameter. For example, a typical *proposal* rule would match state conditions in the *If* clause, and create a named operator in the *Then* clause. Conversely, a typical *application* rule
would match the named operator in the *If* clause, and make changes to various state or input/output link variables in the *Then* clause (more on this later).

Soar also supports and makes use of state *elaboration* rules to perform calculations on internal variables to assist in state tracking and future operator selection. At each state of the decision cycle, every production rule is matched and executed in parallel, regardless of type. Once a wave of decision and elaboration rules have fired, their updates to conditional WMEs may cause additional rules to match or retract. Soar will continue to fire or retract additional waves of rules until no more rules match; a condition labeled *quiessence* in Soar literature.

At first impression, it appeared that each successive wave of operator proposal (and elaboration) constituted a single cognitive cycle, as each wave could update WMEs that impact the firing and retraction of other production rules. However, from the perspective of an outside environment, a single Soar *decision cycle* (including all phases) controls the rate of updates to perceptual (input) and motor (output) link variables used to observe and effect change beyond internal deliberation. Therefore, linking a decision cycle to the Common Model cognitive cycle requires a theoretical assumption that a single decision cycle could, in fact, represent the smallest unit of measure equal to Newell’s *deliberate-act* cycle. Soar does allow the user to specify

---

Fig. 4.36.: Model trace of Soar counting task productions.
the maximum amount of processing time per decision cycle, so the example module calls `max-dc-time 0.05` during initialization, where 0.05 (50ms) is the upper limit per cycle.

The Soar kernel offers a selection of `run` command parameters, allowing the modeler to run the agent for a specific number of elaborations (-e), or phases (-p), or decision cycles (-d). The Metaverse example agent, based upon the above assumption, implements the `step` interface function by executing `run -d 1`, where 1 is the number of decision cycles, to represent a single Common Model cognitive cycle.

**Working Memory**

In Soar, everything in Working Memory is global, and organized into a graph structure using attribute-value pairs. The root node of all Working Memory Elements (WMEs) is the initial state, with all subsequent sub-states (used for problem solving) linking back to the parent state. Collection of WM elements with same first ID is an object, and WM elements that make up an object are its *augmentations*. Therefore, the first item of a WME is the ID, followed by attribute/value pairs for each augmentation of that WME.

Soar syntax also makes the distinction between persistence of WM elements based on structure of the production rule. If a state is tested (*If* clause) and the selected operator modifies the state (*Then* clause) it has operator-support (aka *o-support* in the literature), since operator applications create persistent WMEs that survive until the following decision cycle. If a state is merely tested and no persistent changes are made, it has instantiation-support (aka *i-support* in the literature). When no rule fits the current situation, the cycle is said to have reached an *impasse*, and Soar (by default) automatically creates and enters a new sub-state for resolution.
The Metaverse examples map the \textit{SoarWorkingMemory} module through designated calls to create, update, and destroy WMEs through a pointer returned by calling \texttt{GetInputLink()} on the agent object. Creating a new WME is done by first calling \texttt{CreateIdWME('name')}, where \textit{name} is the human-readable designator assigned to that element, and then assigning attribute-value pairs through that element ID. The compiled SML interface is strongly typed, so there are separate create methods for each of Integer, Float, and String; for example, creating a new float element is done by calling \texttt{elemID.CreateFloatWME(attribute, value)}, where \textit{attribute} is the name of the float attribute, and \textit{value} is the actual numerical value of the float element. This explicit pairing of attribute names and values is reminiscent of ACT-R buffers, in which slot names and values can take on a similar structure and role, if so desired.

The Metaverse example retains a dictionary of ID pointers to the agent’s input link (as \texttt{input\_link\_wmes}), that are later used to update recurring perceptual inputs from the environment.

\section*{Declarative Memory}

Soar distinguishes between two types of declarative memory – Semantic Memory (SMem), and Episodic Memory (EpMem). Semantic memory in Soar is used to store long-term context-independent facts, whereas Episodic memory is used to store context- and/or time-sensitive experiences.

Both forms of memory use identifiers (as pointers) to memory elements, and makes the distinction between Short-Term Identifiers (STI) for temporary semantic memory currently in Working Memory, and Long-Term Identifiers (LTI) for memories stored in Declarative Memory. Semantic memory attribute and value structures are stored and reTreived using the global \texttt{smem.command} attribute of the current state. New memories
are added on the command’s `store` link, and recalled memories are accessed on the command’s `retrieve` link.

Modelers can store memories using the user-initiated API command `smem --add`, which can be specified at the beginning of a `.soar` file and sourced during agent initialization.

The SoarDeclarativeMemory module example implements the `addDM()` function in the by passing a structured memory chunk to the agent’s `executeCommand()` function. The example `addDM()` function expects a name and chunk string, tokenizes the string, and generates attribute-pairs for each chunk in the string. The current implementation creates a linked list of serialized slots to support the procedural nature of most tasks. Soar supports more complimented (and arbitrary) memory structures, however, at the time of this writing, arbitrary knowledge-level representations are not currently implemented in the examples.

**Perceptual Interface**

Soar uses the concept of an input/output (I/O) link in the WM graph structure to facilitate the transfer of data for sensory and motor commands. The `input-link` is an augmentation of the global `io` object, which in turn is an augmentation of the agent’s parent state object. Sensory information is added to the input-link attribute in the form of one or more nested values. Any updates made to the input link from an external source are synced with the agent just after the output phase, and just in time to affect operator proposal during the proposal phase. The input link does not discriminate against the type or structure of the supplied attributes or values, assuming this structure is also reflected in the elaboration and production rules loaded into production memory. The Metaverse example initializes the SoarPerception module through the `create_input_wmes()`
function, and accepts a mapping of the default observation spec conforming to the expected observation space of the target environment. For example, in a CartPole environment, this would be a vector of four (4) float values, each being one dimension of the feature space. The initialization routine, then, iterates through the map, creates a new WME attribute-value pair for each of the feature dimensions supplied by the map, and stores the element pointers in a local input_wmes data structure. Environmental observations are presented to SoarPerception by calling update_input_wmes() and passing the observation vector returned by the environment. The perception module then iterates through the observation space, and calls Update() on each feature element pointer, with the corresponding value of the environment observation feature. The resulting updated WMEs are then synced and ready for operator proposal and elaboration during the next proposal phase of the decision cycle.

Motor Interface

In Soar, all commands destined for an external environment are placed on ^output-link, an augmentation of the global io object, which in turn is an augmentation of the agent’s parent state object. Output commands are created during the operator application phase are added to the output-link attribute in the form of one or more nested values. Output-link WMEs are persistent and need to be explicitly removed after a successful operator application command. This can be done from within the production rule using a reject preference ("-"), or through the SML interface between decision cycles, and are removed during the next operator application phase, in parallel with other operator applications.

The SoarMotor module exposes a get_move_command() function to derive the list of commands and associated parameters through the soar agent object.
Accessing and interpreting output commands can be performed a number of ways, depending on the level of fidelity required by the output handler. The output-link can be queried and accessed for new command values, however, the soar agent also exposes a `GetNumberCommands()` function that returns the total number non-stale attribute IDs that have been placed on the output-link since the last decision cycle. The Metaverse example uses this function, and iterates through the list of commands by calling `GetCommand()` on the agent, storing each command and mapping of attribute IDs in a local data structure. Finally, the SoarMotor module parses through the command mapping structure, and depending on the environment adapter type, will parse out the appropriate command type and return it to the calling function (in this example the experiment step).

**Issues and Opportunities**

The original Soar SML interface was compiled for a Python 2.x environment, however, as most packages no longer support a 2.x environment, I had to recompile the SML library using SWIG, and generate the interface for working in the Python 3.x environment; this is now included as part of the Metaverse package as `Python_sml_ClientInterface`.

There are three aspects of this decision cycle that are in stark contrast to the ACT-R production system. First, an explicit separation of production rules into operator proposal and application, with the operator data type connecting them, forces a modeler to separate where decisions are made based on available knowledge, and where actions are being performed on state variables and input/output links to interact with the environment. This methodology has advantages and disadvantages, as it makes rules much more verbose, but also grows quite quickly and is far more
to manage without the assistance of dedicated knowledge management tools (like VisualSoar).

The second major distinction is the presence of parallel rule expansion and retraction in both operator proposal and application phases, with multiple rounds of updates to WMEs in both phases. While this may be a proper reflection and application of the efficiency of the OPS5/Rete algorithms (on which it is based), it also calls into question the functional equivalence of one decision cycle as a single cognitive cycle, and presents a challenge for integrators looking to compare different agents in lock-step fashion with each other, and/or an external virtual environment.

Third, Soar doesn’t automatically pick a random operator if there is a tie impasse, unless the modeler explicitly marks them as indifferent (”=” tag in SML), indicating it doesn’t matter. This is, for all intents and purposes, a way to give priority to production rules; it may make a model immediately more efficient by assigning a-priori utility, however, it may also stifle learning or application of lower-priority rules in states that have not yet been observed by the domain expert. If any one of the impasse conditions occurs, Soar creates a sub-goal, where the goal is to find the knowledge (i.e. update WMEs) necessary to resolve the impasse.

The creation of a new sub-state for problem solving is a design choice that closely couples the knowledge and cognitive layers, and makes it difficult to assess and describe the Soar production system using only one perspective or another; this reliance on Problem Space Computational Theory, and later developments of the Goal Dependency Set (Wray and Laird, 1998) as the de-facto standard for impasse resolution, is one example of a strong theoretical commitment that extends across system levels. This commitment is, arguably, appealing more to the use of Soar for creating agents that approximate the knowledge level, as opposed to using Soar for
the sole purpose of approximating human psychology and behavior at the cognitive level.

4.6.5 Mapping to Virtual Environments

Every cognitive architecture used in this research provides a variety of self-contained tests that use a representation provided by that suite of tools. Environment support in the Metaverse framework is built around two scenarios; either self-contained tests using tools internal to the architecture, and external tests that rely on a 3rd party environment. The moderation performed by the experimental framework between agent and environment is the same regardless of the scenario chosen, however, 3rd party environments require addition configuration depending on the chosen test.

Metaverse is configured to take advantage of the Application Programming Interface (API) provided by OpenAI’s Gym platform. The Gym API defines a standard set of methods that must be supported by any environment that is registered as available for Gym agents. Gym was designed around the principle that every agent and environment goes through a standard cycle of observations and actions, with only minor variances in the level of extension or wrapper support offered by each environment. The Metaverse Environment framework takes advantage of this assumption, and is designed to provide a layer of abstraction to negotiate the perceptual and observation transduction function at run time, with consideration to any pre-defined mapping provided by the experiment handler.
OpenAI Gym Requirements

The Metaverse environment framework supports a subset of the standard Gym environments, as well as a custom Gym environment to interact with the StarCraft RTS game. The core Gym class specification requires all environments to support initialization, stepping, resetting, and to return an observation vector including a representation of the current state. The core Gym API also provides extended environment wrappers that provide utility functions for compressing or transforming various environment representations, however, I did not find any environments that make use of these utility wrappers.

There are currently three distinct types of Environment objects: a SimpleEnvironment to support internalized (psychometric) models typically included with cognitive architectures, a GymEnvironment with extended support for the Gym API, and a StarCraftEnvironment with extra flags, action codes, and a pre-processor necessary to interact with the feature layers returned by Google Deepmind’s PySC2 wrapper.

Perceptual Interface

Environment objects allow interrogation for both the observation space, and the last observation recorded from the virtual environment. The observation space, per the Gym specification, includes one or more Spaces that an agent can expect to see at each step, representing the most salient features required to satisfy each task, or reach an end state or goal. The Metaverse examples each have a unique observation space that, after transformation, should match the dimensionality expected by the agent.
Motor Interface

Gym environments also specify an Action Space, that defines the discrete or continuous inputs that are considered valid for that environment. Simple environments, for example, may take limited discrete inputs representing cardinal directions, whereas more complex inputs may require a combination of continuous or analog inputs combined with discrete commands or buttons. Some environments do not have a “no operation” (NOOP) or equivalent, and always require some input; in these cases we can just use the last operation submitted to the environment until the observed state warrants a different input. Last, most cognitive architectures, unlike embedded systems or agents with direct access to command APIs, usually include some form of delay to simulate a human user. In this case there will be a slight delay between observations and the fastest intentional response.

Issues and Opportunities

The OpenAI Gym library is a self-contained Python package that can be installed via PyPi, and includes a sampling of basic environments. Some environments require additional libraries that must be installed separately, and are platform-specific. For example, the Box2D library used for some of the physics only provides native Linux support, but deployment on a Windows host has proven troublesome in previous versions. At the time of this writing the Gym libraries are written for Linux test environments running at least Python 3.6+.

The StarCraft API provides native support, or with a Python wrapper authored by DeepMind. The perception and action spaces of StarCraft are quite complex compared to the Gym environment API, and may be considered overly complex depending on the objective of the experiment. The Metaverse framework makes use
of a 3rd party wrapper that presents the StarCraft 2 Python API in the form of a registered Gym environment. This configuration has the benefit of simplifying the requirements for interaction to a manageable feature set for mini-games, and enables code re-use – anything written for general interacting with existing Gym environments (not including environment-specific code) should also work with the StarCraft mini-games using this method.

4.7 Experimental Design

The approach to testing cognitive architectures is based on the assumption that they can be compared for general and specific equivalency. This thesis proposes a design approach that allows a modeler to specify the general requirements and boundaries for an agent, and interface with an arbitrary cognitive architecture so long as it conforms to the Common Model of Cognition discussed in Chapter 2. This approach also assumes that there are implementation details inherent in each cognitive architecture that may preclude even the simplest tests from achieving equivalency when using a pure bottom-up approach to model coding, or transforming model code in one architecture to the model code of another architecture.

This approach to evaluation is essentially Popperian, and later extends into a posture of a Lakatosian research programme; the use of novel Psychological AI may still be a contributing factor. This is important because the simpler tests are easily repeatable and verifiable against human data, and yet, some of the more complex tasks may draw on modeling features or exhibit capabilities that are not easily falsifiable in a laboratory setting. For additional background on the use of Psychological AI, see Appendix A.2.

The proposed design and evaluation methodology discussed in this Chapter are inspired by, and may constitute an attempt to answer the calls and challenges of,
various AGI Roadmaps presented in (Adams et al., 2012), (Davies and Francis, 2013), and (Laird et al., 2017b). The comparison of models for equivalency draws on modular intra- and global inter-model measurements (respectively) as discussed in (Stewart, 2007), with allusions to Newell’s time scales (Newell, 1973) to differentiate between Micro- and Macro-level behaviors, and implemented according to the MetaObject Facility (MOF) specification, as formally (and successfully) demonstrated in (Caro et al., 2018).

The experimental design makes use of both human and machine research participants, virtual environments to conduct predetermined capability testing, procedures to replicate the study, and measurement instruments to assess models for equivalency and adherence to design principles.

### 4.7.1 Participants

The Research participants involve different virtual environment replay files that were generated for and released by the authors of each environment; participants released their rights to the replay files. Other agents include representatives of the cognitive architectures and perhaps other bots; in short, no ethical sanctions are applied or were necessary for the completion of this research.

### 4.7.2 Instruments

The instruments (for testing and measurement) are the three cognitive architectures (i.e. CMU Lisp ACT-R, CCM Python ACT-R, and Soar), a variety of virtual environments including the OpenAI library and StarCraft II game, a platform to access and measure the internal and external evaluation of cognitive
agents, and a variety of statistical algorithms associated with computational performance.

### 4.7.3 Procedures

Experimental procedures include setting up the environment, defining the state-action space in terms of subject matter expertise and time-varying phenomenon according to Newell’s world view. Test procedures that are specific to each environment are described in section 4.8.

### 4.7.4 Materials

The virtual environment output materials include the replays generated by previous human and machine agents, as well as the output and replays generated by the cognitive architectures being tested. I evaluated the formal model, architectural interfaces, and instantiated agents, from different analytic perspectives, being computational, architectural, and behavioral.

A computational, or algorithmic perspective considers the underlying processes to assesses if the agent uses the same theories of cognition, and considers if there are implementation differences in how the problem or task is achieved internal to the agent, and external to the supporting architecture.

The architectural, or structural perspective is in-side out, and considers the degree of similarity between components specified by the CMC, or implementation-specific assumptions in design a pattern that is unique to the target architecture, as compared to the formal requirements.

A behavioral perspective is outside-in and considers if two agents behave in a similar fashion when conducting the same task in the same environment, how close
they are to human performance as previously recorded, or if there are certain inputs
or states that cause an abnormal failures.

An *environmental* perspective considers the target environment, and how well the
environment supports interaction with intelligent agents attempting to observe and
act through a predefined interface.

### 4.8 Test Environments

Many environments have been used for the comparison of intelligence agents
within the fields of AI and Machine Learning, however, there are few publications
describing a common platform for agents based on Cognitive Architectures. In this
section we provide an overview of the proposed simulation environments that will be
used to evaluate the fitness/utility of each architecture, and adherence to the
principles stated in the Standard Model Assumptions.

When selecting a suitable environment, or series of environments, we must consider
the measurements by which any agent will be assessed. Requirements for cognitive
architectures, specifically, can be paired with each of these sub tests through a battery
of model-centric environments that make affordances for lack of exterior sampling,
and instead generate a model based around parameters specific to that test.

#### 4.8.1 Psychometric Tests

The majority of programmers and developers begin their path of exploration
using a set of well-known and well-documented toy problems; cognitive modelers are
no exception and typically begin with a “Hello World” version of simple
representation. I found numerous examples of simple psychometric tests embedded
through the available tutorials. These tests are good place to start for good reason:
the models to implement them are already available, and their results have been validated against human data.

All of the cognitive architectures explored in this thesis have a set of tutorial problems that require no external representation, meaning there is no 3rd party system aside from the cognitive architecture that must be installed and configured before a model can be tested. These tests fit into a relatively small model file, have well-defined parameters, and pre-generated output samples that can be used to verify correctness and debug models while learning the subtle nuances of each particular architecture.

The author has successfully reproduced an example counting test designed to demonstrate the core capabilities of Procedural and Declarative Memory systems, by using the MDD approach to EMF generation, through to model file generation using a template, and finally execution using a rudimentary middleware that can interface with both of ACT-R and Soar environments.

4.8.2 Classical Games

Classical games provide a simple and repeatable problem space for testing and evaluating intelligent agents. Many early arcade games from the 80’s and 90’s can be easily accessed through standard platforms free of charge. Two recent platforms, being OpenAI and the General Video Game (GVG) Competition were established for the purpose of testing Reinforcement Learning agents, and present a standard API for agent interfaces. This thesis makes use of simple games from the OpenAI platform for testing single agents against a simple representation.

The OpenAI research company has developed the open-source Gym toolkit (Brockman et al., 2016) for community-driven research in Reinforcement Learning, and provides a standard interface for researchers to measure an agents ability to
learn how to navigate a variety of environments. Gym environments range from the extremely simple to highly complex with graduating degrees of difficulty, and supports both local and remote training and testing configurations.

![Gym Environments](image)

Fig. 4.37.: OpenAI Gym Virtual Environments [Brockman et al. 2016].

The Gym platform divides the environments into problems sub-types, depending on a number of factors such as the complexity of the representation, the possible feature-action space, and the increasing degree of overall difficulty to put the agent into a solved state. Example problem classes include *Search & Optimization* for text-based games, *Classic Control* using a joystick or control pad, and *Box2D* environments for more complex physical representations.

Gym test environments are divided into a number of standard classes depending on problem type, such as *Algorithmic* text processing problems, *Classic Control* of agents in a one-dimension plane, and *Box2D* to control agents in a two-dimension plane. Every environment in the Gym toolkit provides a standard interface for observation and action by the agent. At the time of this writing the Gym API supports Python 2.7 or 3.5+, and allows researchers to record and upload results to compare agent performance.
Many virtual environments designed for testing intelligent agents make certain assumptions around how the agent will interface with that environment. The OpenAI framework, in comparison, makes no assumptions about the agent interface, and instead dictates a standard API by which agents can interrogate, observe, and act on the environment; the actual definition of the environment is left to each designer, and does not assume the agent will read sprites from a region of the screen.

The Gym API also defines the concept of a space, that allows the calling agent to briefly interrogate the allowable actions for that environment, as well as the expected range for each feature in an observation. For example, the Lunar Lander environment will report a total of four allowable actions for each thruster, and the expected number range for features that describe the position, angle, and velocity of various dimensions. Feature spaces can be a standard unit vector represented as $[-1, +1]$, or an infinite boundary represented as $[-\infty, +\infty]$.

The Gym test cases included problems from the Classic Control and Box2D categories as previously described. The Classical Control environments were Cartpole-v0, and MountainCar-v0, while the Box2D environment was LunarLander-v2. This thesis explored the interaction of cognitive models in these environments, and uses results generated from the LunarLander-v2 environment for comparison.

### 4.8.3 Real-Time Strategy

StarCraft II is a suitable environment for a variety of reasons: AI researchers from various industries are already taking it seriously; the SC2 environment has the support of a large community of players and researchers; and the environment has been around long enough to generate an extremely large sample of human-generated replays for analysis.
The StarCraft game series is set in a fictional universe featuring three primary races vying survival and control of resources represented through a series of maps taking place across a variety of terrain. The three primary races (or factions) available to the player are known for specializing in a certain style of warfare, each with corresponding strengths and weaknesses that may appeal to different play styles in a rock-paper-scissors fashion. Each race also features a mix of units that can be created, upgraded, and controlled either independently or in pre-assigned groups, adding to the variety of options available to a player to counter an adversarys moves, while also adding to the potential complexity of evolving gameplay.

Observations in StarCraft II are represented to the player agent in one of two methods; a raw API that can be called by an agent class, or a more traditional screen well as a rendered screen representation by using an array of RGB values. The CMC
platform interfaces with the StarCraft API through the use of the Sc2Py libraries supplied by Deepmind.

The StarCraft environment, and associate replay data, has already been used to compare the strengths and limitations between human and machine cognition. For example, a recent tournament held between World-class players and Google Deepmind’s Alpha Star agent (Vinyals et al., 2019) resulted in a world-first victory for AlphaStar over time- and battle-tested Grand Master player.

Later analysis in (Young, 2019), however, showed the AlphaStar agent demonstrating behavior that is not consistent with that of a human adversary, including full visibility of the entire playing field (in most tournaments this is considered cheating), as well as non-human like interactions through the game API measured in Actions Per Minute (APM). A typical professional league player will show short bursts of 350-400 APM; AlphaStar, on the other hand, produced bursts as high as 1400 APM, about three times faster than the best human beings. These forms of measurements, while simple, can be used to assess the realism and “human-ness” of intelligent agents in a complex virtual environment.

In (Peters et al., 2019) we find a preliminary investigation into the corpus of data generated by interaction between human and machine, and those aspects of micro- and macro-level interaction indicative of agent proficiency, sophistication, and nature. This thesis implements CMC models akin to those inspired by SGOMS (Peters et al., 2018b) by using the mini-games developed by Google Deepmind as a standard of comparison for discussion.

4.9 Measurements and Analysis

The purpose of these measurements is to verify and demonstrate the parts that bring common models together, identifying those that don’t conform, and
demonstrate how much of the CMC can be used to generate model templates before low-level task-specific heuristics must be encoded by hand. This approach is also supported by using the modular and global measurements (such as clock rate, motor control, etc.) that demonstrate conformity to the Common Model of Cognition by using Meta-model transformations and model generation that can be formally validated.

4.9.1 Evaluation Requirements

In order to evaluate the cognitive models and compare them to each other, we use measurements that conform to four general perspectives.

First, *Architectural* measurements assess the underlying assumptions and constraints that are made available and used in the design of each model, with tests specific to the algorithmic implementation of each functional module. For the Common Model, this means measuring the information flow between components (through buffers), and the overall utilization of each component. This perspective is also important for the overall evaluation of, and comparison between, cognitive agents using various micro-theories, and agents implemented by different architectures, respectively.

Second, we look at generally observable *Behavior* of each agent with respect to each virtual environment; this behavior is analyzed from both quantitative performance specific to each domain, as well as qualitative characteristics with respect to other human and machine agents, where available. For the sake of this thesis, behavior metrics take on different forms, such as utility and fitness accomplishing a given task, interaction via input peripherals, or deviation from the same performance metrics collected from humans performing the same task. In
short, this perspective is used to determine how well the agent serves the environment, and how well the agent exhibits human-like characteristics.

Last, we look at the Environment and its impact on the performance on cognitive agents, and the system as a whole. Some environments are designed explicitly for Reinforcement Learning agents, or at least support a retro-fit API developed by a member of that particular community. Other environments are difficult to interface, creating technical hurdles that may hinder data collection or misconstrue measured results. This thesis describes each environment in terms of the selection and suitability of agent interface, how the environment is presented to the agent, and how the agent can control actions through embodiment. In short, this perspective is used to determine how well the environment serves the agent.

These perspectives, when used together, form a spectrum that we believe is both necessary and sufficient for model comparison, and to confidently establish level of equivalency at each phase of the Model-Driven Development tool chain.

4.9.2 Architectural Measurements

These measurements are used to assess the characteristics specific to each model in the CMC, as applied to the components or interfaces that correspond to each of them. The CMC architectures evaluated in this thesis all present model trace information through a choice of Command Line Interface; this is typically done via standard output (stdout), however there are alternative options using the provided GUI or debuggers. The Common Model Platform interface generator will query the appropriate mechanism for each component, depending on the target architecture. The interface will collect metrics on the following Common Model components: the production system; working memory; declarative memory (semantic and episodic);
procedural memory; perceptual input modules (visual, audio, and tactile); and motor output modules.

The Production System, as the heart of the cognitive cycle, acts as the hub of all information flows within and across the other functional sub-systems. Test measurements will focus on capturing event triggers and their corresponding load or utilization by type of an frequency. If multiple production rules are being compared, we also capture the conflict resolution process by way of serial or parallel rule comparison. Some architectures make use of Working Memory buffers or equivalent data structures for pattern matching; for example, querying and modifying the chunks in these buffers are used as part of the production cycle, therefore this activity is captured as part of production system evaluation.

The primary method of measuring cognitive models in ACT-R and Soar is through a Model Trace. ACT-R 7.0+ provides a default model-trace command applied to each instantiated module and sent to stdout. These commands are available to model files generated through the Model to Text transformation, as well as through external interface calls using RPC over Python, as shown in Figure 4.39.

Soar provides five different levels of output granularity to include Working Memory Elements, state chart, warnings, information, and full traces of WME changes at each elaboration and action stages. These trace options can be set in the Soar configuration.
file generated through Model to Text transformation, and is also available using the pre-compiled SML Python interface, as shown in Figure 4.40.

![Model trace](image)

Fig. 4.40.: Model trace of Soar counting task.

Memory systems include Long Term declarative and procedural systems. Evaluation of these functions, whether logically together or separate, includes chunk storage and activation, the use of noise or salience during activation, and reading activation rates. Measurements also make consideration for other characteristics such as the use of associative memory, inhibition, memory fade rate, partial activation, and blending.

The Perceptual system encapsulates sensor information arriving from the external environment, and can include various forms of feedback including visual, tactile, etc. Measurement of perceptual signals can arrive as a raw or pre-processed format and temporarily stored in a visual or imaginal buffer. Consideration is made to the use of image refresh and fade, and the assumption of FINST time delay between image processing and object recognition (Pylyshyn, 1980).

The Motor system generally defines the feedback provided to the external environment in the form of actions. These actions may be defined as simple as direct input or with a simulation of mechanical motion. Measurements include
feedback type, rate, and categorical frequency. Consideration is made to scenarios in which feedback simulates Fitts’ Law or GOMS theory.

4.9.3 Behavioral Measurements

The behavior of cognitive agents is represented and evaluated by an external observation, from the perspective of another agent in the same environment, and according to some ideal policy or standard. Behavior-related evaluation, therefore, includes patterns of state representation used by all test environments. This research measures behavior from multiple perspectives, such as external observation of how well the agent performs on a task in the target environment, and determining the human-likeness of the agent as compared to human-generated data and metrics, where possible.

We are interested in establishing measurements to describe how close a cognitive agent performs to humans in a similar setting. Some environments, such as StarCraft II RTS, make this information readily available. Our previous work in [Peters et al., 2020] demonstrated how baseline metrics, such as raw Actions-Per-Minute (APM) and state transition frequency, can be used to distinguish between human and machine game play, regardless of player or agent skill level. This research continues this work with a focus on drawing the relationships between cognitive architecture and the level of human-likeness, both measured and perceived by the casual observer.

4.9.4 Environmental Measurements

Some environments are purpose-built for the evaluation of intelligent agents, while others require additional effort to ensure there is an appropriate interface for agent interaction. The environments described in Section 4.8 vary in complexity,
representation, standardization, and extensibility of interface. Our previous work in (Peters et al., 2018a) demonstrated how raw perceptual interfaces stored as sub-symbolic ensembles of Spiking Neural Nets, while flexible enough for any dynamical system, still have to be tuned to the expected dimensionality of both input signals and output motor controls; a biologically-plausible limitation similar to our own visual and motor pathways. Further, in (Peters et al., 2019) we described the requirements for normalization of input features so that a learning system, regardless of type, can converge on an optimal policy gradient. This is important as some environments do not supply this distribution as meta-data, and requires a measure of sampling ahead of time.

Environmental Measurements, therefore, address the feature space presented for observation, the action space present for feedback to the environment, feature complexity and distribution, whether the interface provides raw access (via memory) or peripherals engineering for humans (i.e. keyboard, mouse, joystick, monitor), and the effects these interfaces may have on agent performance and evaluation in the other perspectives.

Once measurement criteria has been defined, we also need to define how models are validated against the requirements of the CMC, and to answer whether or not different comments can be assessed for design- and run-time conformance.

4.10 Validation of Research Questions

In order to evaluate the answers to the research questions, we need to understand how each of the perspectives and constituent measurements from 4.9 map to each of the research questions introduced in Section 3.2. The first group of questions aim to address the lack of common, repeatable empirical measurement criteria for quantitative evaluation of Common Models, and how we can trace validity of all
Common Models through a formal Meta-Model. The second group of questions deal with validation of the internal components of the Common Model. The last group of questions deal with how specific the CMC needs to be to integrate with different virtual environments.

4.10.1 Meta-Model Definition

The Architecture and Behavioral perspectives were used to evaluate the level of fidelity for models represented through a formal specification. First, we used the meta-model design toolchain to verify conformance of the Common Meta-model against the Meta-Object Facility standard, and to ensure model files are well structured. This is accomplished using the Eclipse Core Model validator to compare XMI instances against the original CMC Meta-Model UML. Second, model templates are used to evaluate if each of the CMC components can be used without further developer support for specific architectures. This involves applying each template to specific tasks outlined in Section 4.8. If a model can be executed, I compare outputs in terms of the observed behaviors; otherwise, I describe which aspects of the model cannot be generated through a Model to Text template, and why.

4.10.2 Internal Model Validation

The second group of questions deals with the specific components supported by all CMC architectures. Data measurements dealing with component utilization, algorithmic approach, and adherence to micro-theory were measured through architecture-supplied monitoring interfaces such as dispatch service or mirroring output to stdout.
Validation of candidate production cycle is established through monitoring the observe-act production cycles to ensure that they adhere to a 50ms clock cycle, and that the agent stays lock-step with the environment’s clock cycle. Validation of serial bottleneck is measured by buffer contents during both the deliberation and action phases to establish the upper bound of “chunks” available during these phases. Human likeness of cognitive agents is established by comparing agent output in terms of raw action frequency and sequencing.

4.10.3 Environment Integration

The next group of questions deal with the use of virtual environments that are not supported by any one independent architecture development group. This is important to avoid a perceived conflict of interest arising from the cognitive architecture labs designing their own test environments and tasks, and therefore establishes greater credibility for both the architecture, and selection of task environment. Establishing an ideal environment may be considered a technical panacea of sorts, however, we can use a combination of computational and environmental metrics to demonstrate the level of support offered by each environment for future cognitive architecture testing. First, I evaluate the generation of capture data for humans and machine, the format and specification, feature space complexity, and coupling observation and action cycles. Capture data is measured by format, specification, and transmission protocol used. Feature complexity is described by range, deterministic or continuous inputs, and if the number of features is fixed or variable. Observation and action space coupling is measured according to direct raw interface, proxy service, or use of virtual peripherals. Environment test execution is measured according to level of repeatability through a seed value or deterministic replay, and if customization to task difficulty can be specified by the modeler.
4.11 Current Limitations and Threats to Validity

All research designs are subject to limitations, and this thesis is no exception. The test scenarios, experimental values, measures, and assumptions may be subject to errors in execution or collection, may not contribute to the study, or possibly invalidate my research findings.

4.11.1 Ad-hoc Design Decisions

Using a Meta-model to generate model templates for different architectures only works so far as all test architectures use the same constraints without making a commitment to a specific theory of implementation. For example, both ACT-R and Soar use elements of buffers and chunks, however they rely on very different data structures to represent their relationship with control state. A meta-model could encapsulate (and mandate) the use of chunks and slots, details outside of the scope of the CMC, however choosing to do so is a design decision that may have unintended consequences during template generation or how commands are being front-loaded to a different architecture.

Cases where I am forced to make a design decision or assumptions, or refrain from doing so, have been documented throughout model generation and testing.

4.11.2 Differing Ontology

The CMC uses a standard vocabulary to describe the components and relationships between components, however upon closer inspection they may refer to these components using different (and sometimes conflicting) context-sensitive terms. For example, chunk types were created for imperatives in ACT-R, and then they disappeared from use because they became unnecessary later on; these are
essentially vestiges of ACT-R’s historical bottom-up development methodology to cater to the development language (LISP). Chunk types are strictly enforced in Soar (and Sigma by extension), forcing instantiated models into a more specific commitment in relationship. Choosing to support differing levels of declarative typing may force loss of precision or even completely change the results of a test.

4.11.3 Incorrect or Ill-defined Assumptions

Generating specific models based on templates challenges some of the assumptions that have been established in the wider cognitive modeling community, and yet may not be explicitly specified in the CMC *per-se*. For example, implementing an upper bound of the number of chunks stored in the buffer (e.g., 7 plus or minus 2) may be a common best practice, and yet not strictly enforced in any cognitive architecture. Building these assumptions into a meta-model may constrain future models that use it, without adding any value to a model’s validity or conformity. These assumptions may be ill- or overly-defined on my part, and although credible, may deviate from the Common Model community writ large. I have documented, to the best of my ability, those cases where I am trying to formulate a model transformation based on best practices, and to reference architecture documents that agree on the rational or historical evidence for a common assumption.

4.12 Summary

This Chapter presented the Model-Driven Design methodology that was used for formal validation of the requirements specified in the Common Model of Cognition. We reviewed the application of an established specification, the Meta-Object Facility, and how it can be used to design and derive the meta-model
transformations for the CMC, and their respective architectural interfaces, and possible design decisions that would have to be made for interface points at Perceptual and Motor Modules. This research methodology makes use of multiple test environments that exist internal and external to the model architecture, with scenarios designed to test specific micro-theories or human-like performance. I have used different analytic perspectives to answer the research questions, to draw on qualitative data derived from computational and virtual environments, architecture-specific interfaces, and behavioral observation. This research design does face some challenges due to model transformation uncertainty from design decisions and incorrect assumptions. The next chapter presents a review of the findings derived from applying this methodology.
5. RESULTS

5.1 Introduction

This chapter presents the results of mapping each architecture to the generated CMC framework that was described in Chapter 4. There are many features in the CMC Meta-model that can already be mapped to each architecture. Model Initialization can map the creation of module components, and loading context from an external source. Production System can map the cognitive cycle step time, decision cycle, and run time. Run time is not in the CMC, but could be added to the specification. Production Memory can map initialization and loading of production rules. Working Memory can map the creation and updates of WM buffers. Declarative Memory can map the initialization (loading), adding, and retrieving facts from DM. Perception can map creation and storage of new patterns, while Motor interfaces can map creation and storage of new actions.

These results demonstrate that all three of the Lisp ACT-R, Python ACT-R, and Soar cognitive architectures can be mapped to a Common Meta-model, and that through this mapping, task model code can serve as the implementation of a specific cognitive theory. This implementation and mapping, however, was not a seamless endeavour, and resulted in some conflicts that may warrant pause for consideration of just how “cognitive” they really are.

The Production System cognitive cycle does not operator the same between architectures, with Soar defaulting to multiple parallel processes, decisions, and sub-goaling occurring within one decision cycle. This could be overcome by
specifying maximum constraints within a cognitive cycle, but this also requires an update to the CMC.

Procedural Memory initialization in ACT-R is only supported at design-time, whereas Soar supports run-time creation outside of rule composition, restricting proper mapping to the respective *getter* and *setter* methods that can be mapped.

The CMC, in its current form, uses conflicting terminology between memory stores; *chunks* as DM facts in ACT-R, and *chunking* as Production rule compilation in Soar. This conflict makes a formal meta-model representation of storage interfaces and components difficult beyond the simplest representation at the module level, and needs to be reconciled.

The run-time results of running each architecture against the same knowledge-level task also demonstrates an adherence to the fundamental principle of the cognitive cycle, and that they are capable of comparison across different cognitive models when aligned to a central clock. For example, all architectures display the same production memory activation rate across multiple trials and conform to the CMC specification.

Although performing the same task activity, each architecture displays different behavior across modules, due to very different theoretical assumptions and/or implementation details that are difficult to reconcile through a standard framework. For example, the measured motor activation traces in the Beacon test are not uniform across architectures; both ACT-R implementations remained consistent with professional human players, whereas Soar output rates were closer to AI bots.

These results demonstrate the importance of establishing a formal criteria for measurements specific to each common functional component. The remainder of this chapter is organized as follows:

Implementing the Common Meta-Model (Section 5.2) provides a review of how well each cognitive architecture aligns with a functional interface for testing
Common Models; each meta-model component is reviewed according to the function definitions that are callable through the interface, with examples of how the calling process works. In Section 5.2.2 mapping implementation conflicts (both potential and realized) are discussed from the perspective of each meta-model component. This section demonstrates the overall accessibility of each Cognitive Architecture for integration in non-native environments, the features that can (or cannot) be mapped to the CMC using these exposed interfaces, and how to test and validate these claims.

Last, Evaluating Models with the Framework (Section 5.3) presents the quantitative and qualitative results of systematically testing each functional area of the CMC architectures using the mappings between each architecture and the interfaces generated from the formal meta-models. This section contrasts each architecture according to each CMC specification area using OpenAI virtual environments of varying levels of complexity, offering insights into the applicability of this methodology for ongoing research.

5.2 Implementing the Common Meta-Model

Integration between disparate systems is never an easy task; especially if they were never meant to work together, or side-by-side. Creating a common interface that adheres to a theoretical commitment demands a level of trust that, if one looks hard enough, one will find a representation of that theory in the system to be integrated; cognitive architectures are no exception. Dealing with micro-theories, such as a specific value for timing, activation, or delay, is very specific and narrows our search considerably. Looking for the best way to map a macro-theory such as “all agents have a working memory”, is somewhat more difficult as the identification of architecture-specific components, implementations, and function calls, can quickly
become an exercise in traversing an entire reference manual; a form of exercise that leaves the developer extremely enlightened or miserably frustrated, or (most often) somewhere in between. We begin the review on a lighter note of what went well.

### 5.2.1 Identification of Shared Functions

All of the cognitive architectures mapped to the CMC Framework had representations of every meta-model component described in the CMC Meta-model. This should come as no surprise, as these architectures (or those derived from them) were stakeholders in the original CMC definition. The Meta-Model of Cognition already agrees and successfully maps the following specification:

<table>
<thead>
<tr>
<th>Module</th>
<th>Description of Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>General model vs Specific modules.</td>
</tr>
<tr>
<td>Production System</td>
<td>Timing constraints, real vs simulation.</td>
</tr>
<tr>
<td>Production Memory</td>
<td>Loading from external representation.</td>
</tr>
<tr>
<td>Working Memory</td>
<td>Reading/writing along communications channels.</td>
</tr>
<tr>
<td>Declarative Memory</td>
<td>Storage and recall using dedicated buffer.</td>
</tr>
<tr>
<td>Perceptual and Motor</td>
<td>Input and output using interfaces.</td>
</tr>
</tbody>
</table>

Table 5.1.: Validated Mapping

Table 5.1 outlines the notable areas of agreement arising from integrating with and testing the architectures. The following sections discuss these shared functions in greater detail.

**Model Initialization**

The CMC Meta-Model maps initialization functionality to the `Module::load()` operator. OCL tests this function through a pre-condition requiring a path to load
from. Every cognitive architecture provided a method for model initialization, the key difference for reconciliation is understanding how and where this was performed.

A Lisp ACT-R model uses a separate process for the dispatch service that must be running before initialization, Python ACT-R can be instantiated right in the model class itself, and Soar can instantiate a kernel, and subsequently agent, in the same thread using the kernel interface. All three of these methods can be encapsulated as a CMC \texttt{Model()} and reside in the same process for comparison. This result demonstrates the fundamental existence of Theory as Code residing on a common medium and interpreter.

The candidate architecture also provided a method to load model parameters into the instantiated agent, either directly through an interface, or via model file. Lisp ACT-R supports custom settings for meta-data associated with enabling certain features (eg: enable sub-symbolic computing), as well as component-specific settings, through both the dispatcher function calls, and loading from a user model file. The Python ACT-R agent supports direct calls to constituent modules at object instantiation. Soar’s SML interface provides both a command interpreter and sourcing user model files. User Model settings can therefore exist in an implemented class initialization function. Loading from a model file however, is somewhat more restrictive as discussed in Section \ref{5.2.2}.

This result demonstrates a shared approach of exposing general parameters to the user, and facilitates comparisons of agents capable of enabling or disabling certain features or functions before run-time, and testing the effects of similar changes across architectures.
**Production System**

All of the candidate architectures support some way to specify timing constraints for running user models. I encountered three general categories of managing time: the cognitive cycle, the model step, and the model run.

The cognitive cycle refers to the amount of *simulated* time that an architecture permits for a change across working memory systems (and all that it entails, discussed later). The CMC Meta-Model maps cognitive cycle time in the `Model::cycle` value, and asserted by OCL through an initialization of the value to 50ms.

Lisp ACT-R assumes a 50ms clock cycle, and provides a method to override this through general settings. Python ACT-R assumes a 50ms clock cycle that is not exposed by default, however, as with all Python libraries it can be overridden in the Production class. Soar does not mandate a specific cognitive cycle, however, the decision cycle can be limited to a maximum of 50ms at initialization, effectively resulting in the same standard. This result demonstrates the *capability* of each architecture to adhere to a standard cognitive cycle; all candidate architectures are then implementations of this part of the Theory, regardless if exposed or enabled by default.

The model step refers to the amount of *simulated* time that the model will run before pausing (an possibly interrupting a full cognitive cycle). The CMC Meta-model maps this through the `Model::step()` operation, and validated through OCL via operator contract on the step value.

All of the Lisp ACT-R, Python ACT-R, and Soar architectures map this function by passing a time parameter (in seconds) to the run call. Soar makes the distinction between running a single agent, or multiple agents simultaneously, but for the purpose of this research all architectures are able to implement a single model step, and
therefore all three architectures were able to map the model step time from the `Model::step()` operation in the Meta-model.

A model run refers to the amount of real or simulated time that a model will run before it halts and shuts down. This functionality is useful for testing agents in real-time (asynchronous) environments with an indeterminant goal state, or with virtual environments that cannot be controlled through a synchronous clock cycle. The CMC Meta-model does not map this functionality based on an Assumption or Requirement, however, all candidate architecture support real-time model runs. Although not used in this research, the shared ability to specify a run time demonstrates a commitment to supporting experimental cycles which is mentioned in the CMC, and identified as an area that requires further exploration before a general consensus by the CMC is feasible. Therefore, the addition of a `Model::run()` operation is a feasible addition to the CMC specification.

**Production Memory**

The CMC Meta-model maps this ability through a `Model::load()` operator. OCL validation includes an imperative operator contract for a file path, and a post-condition on the presence of Production Memory.

Lisp ACT-R only supports loading production rules from an external file. Soar supports loading production rules from an external file, as well as by executing a command line interface at design-time or run time, making it one of (if not the) most flexible in this area.

Python ACT-R supports loading production rules defined in Python files that are imported by the instantiated agent. Python ACT-R does not support loading production rules from external Python packages or modules (by default), however I was able to mimic this functionality using Python module loader libraries, and
indexing each rule’s meta-class variables in a similar fashion as the Python ACT-R production system.

All of the candidate architecture implement a form of loading production rules, as long as that file conforms to the proper syntax defined by each architecture. All loading processes were mapped to the CMC meta-model. This result demonstrates a commitment to supporting pre-defined Long-Term Procedural Memory representing expert domain knowledge, and goes hand-in-hand with the operation of Working Memory.

Working Memory

The CMC Meta-model maps WM storage through a WorkingMemory.wmchunks relation to Chunk collection, and validated through OCL invariance on wmchunks->exists(). Adding WM elements are mapped through the WorkingMemory::addWME() operator, and validated by OCL through an operator contract consisting of imperative oclType(Chunk).

Lisp ACT-R and Python ACT-R implement the same theory of buffers and chunks, however, Soar uses an entirely different approach, and terminology. Although the terminology and representation is different between ACT-R and Soar, all of the candidate architectures expose their WM elements to the user for inspection and modification.

Lisp ACT-R and Python ACT-R use a combination of multiple buffers to represent the Short-Term Working Memory. Current context is stored in a goal module, is manipulated through the goal buffer, and exists for the sole purpose of tracking the agent’s state. Unlike the other modules in ACT-R theory, the goal buffer can be instantly modified, and acts as a sort of scratch pad. The contents of the goal buffer can be directly modified and updated by Lisp and Python ACT-R by calling their
respective functions; however, in practice, this is only supposed to be done for model initialization, with subsequent modifications coming from production rules.

Soar implements WM through a graph structure that can be read and written via command line interface at design-time, however, ongoing updates from an external source need to be handled through specific input- and output- links. Like ACT-R, the Soar WM structure is designed to be examined and modified by production rules at run time.

All of the candidate architectures have a mechanism of temporary storage that is operated on by the production system through a matching of production rules, and can be directly accessed by mapping to the addWME() operator. This result is important, as it demonstrates a theoretical commitment to relying on a temporary work space for state representation, and on external stores of knowledge that are not immediately retrievable within a single cognitive cycle.

**Declarative Memory**

Declarative Memory is mapped by the CMC Meta-model through the DeclarativeMemory addDM() and getDM() operators. Like the WM module, DM holds a dmchunks relation to a collection of Chunks, and are validated via OCL in similar fashion. The CMC requirements use the term Facts in place of Chunks here.

Lisp ACT-R and Python ACT-R use the same format form placing chunks in the DM buffer for storage, and retrieving a DM chunk by placing a pattern in the DM buffer representing a full or partial match to the desired memory.

Soar makes the distinction between two forms of DM, factual Semantic Memory (SMem), and temporal Episodic Memory (EpMem). Soar uses the same attribute-value graph syntax for both STM and LTM representation, and facilitates memory storage through a specific part of the WM graph for storage (smem.store), resembling
the DM buffer in ACT-R. Soar makes the distinction between Que-Based (QB) and Non-Queue-Based (Non-QB) memory retrieval using the Long-Term Identifie (LTIs) symbols for stored content, or Short-Term Identifies (STIs) for residual copies still in WM, supporting various scenarios for partial matching.

Despite their differences, all of the architectures’ DM stores are used to store features or instructions, recalled as necessary via Production Memory rules. This result demonstrates a theoretical commitment by all architectures for supporting some form of memory storage and retrieval using patterns of constant and variable symbols representing the information to be stored and recalled, respectively.

Successfully mapping Declarative Memory content representation poses the biggest challenge, however, it may be possible to alter the structure and ordering of either ACT-R chunks and slots, or the configuration of attribute-value pairs (also termed augmentations) in Soar, in order to represent, store, match, and recall equivalent semantically-equivalent knowledge across candidate architectures.

**Perception**

All tested architectures, including the vision systems included and available, assume some form of representation pre-processing before they are invoked. Integration with an external environment using arbitrary feature space requires definition of an observation specification, which could also be represented any number of ways.

The CMC Meta-model maps Perception through a `setObservation()` and `getObservation()` operator, stores a representation pattern as an observation attribute, and maps the observation to a feature space through an *Observation Specification* Feature Map. This attribute is not explicitly specified by the CMC
Assumptions or derived Requirements, however, this mapping was required for all test cases, and could be considered for adoption as a standard requirement.

The Lisp ACT-R Visicon assumes that chunks representing visual scene objects are bottom-up representations of the pre-processed “what” or “where” information of the object, not a top-down abstraction of its meaning. The user is responsible for this pre-processing, and has the option of mapping it to a schema based on virtual widgets that represent a virtual display. Chunk definition is largely up to the user, and these chunks must be supplied to a visual buffer if any of the virtual windows/text/button widgets can be used.

Python ACT-R takes a similar approach, but is more flexible in how the user represents the environment features to be perceived; objects can be structured and stored anywhere, as long as they are wrapped as a ccm.Model, and the production rules can access them (or equivalent buffer). This also leaves chunk definition up to the user, but is more flexible in storage and representation.

Soar only requires perceptual input be placed on an input-link that read at the beginning of a decision cycle. Soar does have a Spatial Visual System (SVS) that can store internal representations of objects, and calculate transforms as if manipulating them in 3D space.

All of the candidate architectures provide a mechanism of visual representation, however the emphasis of these systems is very different; the only common ground between them is the requirement of the user to perform a transduction between the input signal and whatever symbolic representation is presented to the input buffer or link. The implemented interfaces map the raw input signal to Perception setObservation(), uses a transduction function that is common to all architectures, and maps to the Perception obsSpec() feature map. The symbolic representation of the input signal features are mapped to chunks (for ACT-R) and
input-link WMEs (for Soar), to facilitate the transfer of external stimuli to WM buffers.

These results demonstrate that all candidate architectures make greater assumptions, and provide more functionality, than is specified in the CMC; the sticking point is lack of clarity in how external signals might be translated and stored in intermediary format before they are manipulated by an internal representation.

**Motor System**

The CMC Meta-model provides two Motor operators, `setNextAction()` for any motor handlers, `getNextAction()` for environment handlers, and stores a `next_action` attribute per CMC requirements of *one action per cycle*. These are validated via OCL operator contracts in similar fashion to the Perceptual module.

Each architecture presents command output in very different ways. Lisp ACT-R supports callback to a function that has been pre-registered during initialization, however, is very specific about how motor commands are represented. Sending keystrokes to a registered function requires the use of a virtual keyboard device, sending a mouse click event requires registration of virtual mouse, etc. The handler function is then mapped to `Motor::setNextAction()`.

Python ACT-R calls back to a function that has been defined in the Motor module implementation, however, does not have the same requirements for a virtual device as Lisp ACT-R, and hard-codes the motor delay timing into the key press function. The handler function is then mapped to `Motor::setNextAction()`.

Soar supports command recognition via output event registration or interrogating the output-link directly, depending on target application (i.e.
embedded systems), however, does not take make same level of theoretical commitments to a motor function as it does to a visual system.

In all cases, motor events are fired during the cognitive cycle step, therefore, the implemented Motor module interface stores the result of all architecture callbacks in the motor module for later retrieval by the environment. These results demonstrate a range of commitments and support by candidate architectures for Motor module functionality; from the use of virtual keyboard/mouse devices in Lisp ACT-R, hard-coded delays in Python ACT-R, to no commitments whatsoever in Soar; this lack of commitment is a topic of some discussion in the next section.

5.2.2 Identification of Conflicts

Combining cognitive architectures based on fundamentally different implementations does not come without its fair share of challenges. While many of the functional requirements could be reconciled through similar function calls or inventive manipulation of results, there are some underlying assumptions and representations that would make further reconciliation a major challenge under the current CMC Meta-model.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description of Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production System</td>
<td>Parallelization and timing: Decision Cycle timing, sub-goal impasse resolution.</td>
</tr>
<tr>
<td>Procedural Memory</td>
<td>Chunk loading: single vs multiple requests per decision cycle.</td>
</tr>
<tr>
<td>Procedural Memory</td>
<td>Support for design- vs run-time loading.</td>
</tr>
<tr>
<td>Procedural Memory</td>
<td>Terminology: memory chunks vs production rule chunking.</td>
</tr>
<tr>
<td>Working Memory</td>
<td>Structural representation: strong theoretical assumptions based on Knowledge Level theory.</td>
</tr>
<tr>
<td>Declarative Memory</td>
<td>DM retrievals: identifiers based on ontological differences between semantic and episodic.</td>
</tr>
</tbody>
</table>

Table 5.2.: Conflicts in Mapping
Table 5.2 outlines the most notable conflicts arising from integrating with and testing the architectures. The following sections discuss these conflicts in greater detail.

**Production System**

The production systems implemented by ACT-R and Soar are, in theory, based on similar principles derived from the successes of OPS5 and the more recent Rete algorithm for rapidly selecting from a large database of production rules. In practice, however, they diverge into very different methods for reconciling between impasse conditions.

Soar explicitly uses the Rete Net algorithm, and under-constrains the production matching cycle during the proposal phase, with waves of elaborations that may update WMEs and impact future operator selection. Soar productions can update WMEs multiple times per decision cycle, resulting on a delayed output. In case of impasse, Soar drops into a sub-state to figure out why there was an impasse, and maintains a goal stack until it is resolved.

A single ACT-R cycle results in a single production rule firing, and only that rule updates working memory. ACT-R handles a production impasse by applying noise and randomly selecting between productions, or using the Utility value based on previous rewards.

This disparity between the number of cycles internal to production evaluation and firing, and Soar’s distinction between operator proposal and applications, makes it difficult to match and compare WM activation between architectures. Soar’s default setting of maximum 100 elaborations per cycle, with multiple sub-goals, makes it difficult to anticipate the timing requirements per cognitive cycle.
The step function mapped through each architecture relies on the assumption that each advancement is equated with a single cognitive cycle. Soar uses 50ms per Decision Cycle (DC), but the “gating” between proposal and application is based on the operation, allowing parallel Production Rules to fire to move forward. ACT-R only fires one Production Rule per cycle, making per-cycle comparisons even more difficult.

This conflict could be ameliorated by mandating a new requirement for a maximum number or duration of decision cycles per cognition cycle. I did manage to successfully implement this by mapping the `Model.cycle` attribute to Soar’s DC time value. This is not a constraint in the CMC, and therefore just because I can map it this way does not necessarily mean that I should map Soar in this fashion until the CMC specification is more specific.

**Procedural Memory**

The structure of the CMC includes a Production Memory that is logically separate from Declarative LTM. In practice, all tested architectures require loading user-specified production rules and learning-related settings prior to running a model. In all tested architectures, the process of adding new rules during execution via production compilation or learning, is done without modeler intervention through an external interface.

ACT-R Production Rules are restricted to retrieving chunks once per cycle, whereas Soar agents can make multiple requests per cycle, as long as they are processed within the same decision cycle they are requested; ACT-R is serial, and Soar is parallel. This is a conflict as it makes comparison between production memory and rule firing difficult.
At the time of this writing, an abstract Procedural Memory representation of the architectures as tested could not specify anything more than a placeholder for design-time initialization of rule sources through `Module::load()` operator. Further, Soar is the only architecture that allows for side-loading new Production rules (at run time) without relying exclusively on loading hard-coded rules before model execution. If other architectures were able to support this feature, the Meta-model could enforce stricter requirements for `ProductionMemory::addPM()` and `getPM()` operators. Further, the CMC would need to clarify between the use of terminology associated with memory storage of facts (i.e. *chunks*) and learning new production rules (via *chunking*) before a formal model could properly and unambiguously represent these concepts.

**Working Memory**

Working Memory representation and manipulation is very different across architectures. ACT-R does not have a separate WM representation; rather, it works through the buffers of other modules (i.e. Goal/Focus context, DM Retrieval, Vision, and Motor). DM has the biggest influence on WM as strings of chunks stored from WM may not be retrieved through decay or interference (preferential relevance). Soar on the other hand does have an explicit WM graph structure that is based on a strong theoretical commitment to Problem Space Computational Theory (PSCT) at the Knowledge-level. Not only does this structure take up a huge amount of memory during (sometimes infinite) creation of sub goals, but any form of comparison with other architectures would force the assumption that WM represented in other architectures conforms to the same theoretical commitments not mandated at the Cognitive level.
ACT-R represents WM buffers as chunks as a list of arbitrary slots that may or may not have a chunk type associated. Soar represents WM as a graph structure with a rigid ontology of attribute-value pairs. Storing and manipulating WM elements between architectures cannot be done with the current implementation, however, an ontological mapping could be performed at either design- or run-time using a technical employed by Herbal Paik et al. (2009).

Both ACT-R and Soar define memory components as chunks, and agree that a chunk representation is used to justify the conditions by which a rule is fired. For example, ACT-R uses the term “chunk” to refer to the contents of WMEs stored in module buffers and referenced by Production Rules. Chunks in ACT-R serve to encapsulate a logical set of symbolic information. Soar uses the term “chunk” and “chunking” to refer to production rules that were not provided before run time, and rather derived from the process of learning new productions through impasse resolution. Chunking in Soar is the process of creating new rules (i.e. Production compilation) to avoid impasses. This difference in terminology may not be an issue in matters of writing an interface represented as standard data structures (eg: Integers, Floats, Booleans), however it does matter when establishing a standard vocabulary when comparing symbolic elements in a conceptual data structure; this must be reconciled before elementary Chunks, Slots, Attributes, Operators, Augmentations, etc. can be reconciled (and possibly mapped) between architectures.

Declarative Memory

Both ACT-R and Soar are historically based on modules adapted from EPIC, but they diverge in both classification and representation of the underlying structures used for LTM. ACT-R uses a single DM for both Semantic and Episodic Memory. Soar distinguishes between them, using different data structures and interfaces. Therefore,
establishing a common Meta-Model for DM is restricted to the greatest common
denominator unless both memory stores can be ontologically mapped to a pooled
storage format, such as one found in the Sigma architecture [Rosenbloom (2013)].

ACT-R restricts matching Production Rules to buffer contents only, and must
request LTM chunks from DM before matching. Soar limits each request to one per
state per decision (cycle), however, can process multiple sub-states simultaneously
in the process of a search space. This approach to memory storage and retrieval is
difficult to reconcile, as parallel storage and retrieval is embedded in WM and PS
theoretical commitments.

In order to map the `DeclarativeMemory::addDM()` and `setDM()` operators to
Soar, the class would need to be expanded to Semantic and Episodic classes with
updates to the operator contracts, with pre-conditions on the use of query-based or
non-query-based STIs and LTIs accordingly. The easier solution, however, would be
to continue mapping Soar to the CMC, with an implementation detecting temporal
elements and modifying the use of `smem` and `epmem` stores.

5.2.3 Summary of Implementing the CMC

Implementing a mapping between the CMC Meta-Model demonstrates how each
candidate architecture can be represented to an outside agent according to a standard
interface. A viable application of Common-Model Theory demonstrates the adherance
of each architecture to a common structure, facilitating the exchange of potential
collaboration between agents.

The implementation was not without challenges, revealing inconsistencies in
presentation and representation, both at the exposed interfaces as well as in the
structures on which they rely. Conflicts included a difference in cognitive cycle
implementation across decision phases, major differences in working memory representation, and production rule evaluation.

5.3 Evaluating Models through the CMC Framework

This section presents the results of running the user task models through the framework to demonstrate the differences between architectures for each functional model. There are three target task environments that demonstrate the range of coupling. The first task is a common incremental counting task that makes use of only the cognitive components of the CMC without influence from non-cognitive perceptual and motor components. The second task is the OpenAI Gym CartPole environment, to demonstrate simple motor commands in a single dimension. The third task is the StarCraft Beacon mini-game designed to test simple object recognition and movements in two dimensions.

These evaluations investigate if we can use data measurements that quantitatively describe the characteristics and attributes of user models to validate or invalidate candidate architectures, and if we can use the data generated by different user models to determine task-specific equivalency and likeness in a comparable environment.

Running experiments across agents requires a communications bus between the target task environment, and the agent to be tested. The Metaverse Python package addresses this need by structuring each generated framework interface in a Python package format, with example experiments to demonstrate how each feature can be used.
5.3.1 Initialization

The Metaverse package includes an experiment library that facilitates the setup and execution of tests using any of the CMC agents or virtual OpenAI environments supported by the package. Agents are created in a uniform manner by requesting an instance of one of the registered CMC factories, and subsequently asking the factory for an instance of a model. All factories and models use identical syntax, leaving the modeler the freedom to define the experimental parameters without concern for the specifics of a given architecture. The example experiments included in the package for demonstration follow the following sequence for experiment initialization:

1. Request and configure a test environment from the environment factory.
2. Request and configure a user model from the architecture factory.
3. Request and configure an experiment model from the experiment factory.

Examples included in the Metaverse package follow this format by initializing each of the three components in sequence, due to their inter-dependencies. An environment with a configured task or problem space is required to establish the observation and action specification (if applicable). The agent is then initialized with the appropriate mapping for the test environment. Last, the experiment is initialized with the user agent model and test environment. Configuring these three components are described in greater detail below. The first step is to select a factory and a model, and load task-relevant information and settings. This creates an instance of an agent using the generated framework, but does not immediately specify any default or environment-specific settings. The user can then load the model with a configuration and task knowledge specific to that architecture (if desired). If no model file is specified, a default generic model will be loaded, however it may not perform as expected or desired.
The task of initializing a model would be made easier if the CMC specified which global settings are common across architectures. At the time of this writing, mapping global settings to model file headers is done in their respective templates. Second, the environment is selected from one of the pre-registered options. Sample environments include one of three tiers of complexity, being psychological testing (internalized models such as counting), simple physics simulators (such as balancing a pole), or a mini-game designed to test agent performance in a real-time strategy game (StarCraft).

The task of initialization would be made easier if all architectures explicitly supported a direct mapping from observation specification to an internal representation. Conversely, the action space mapping would be easier if the architectures used a standard output handler format. At the time of this writing, these mappings had to be hand-crafted based on very different requirements and supporting features.

Last, the experiment is provided the agent, the environment, the number of trials, and the maximum number of steps per trial. An experiment requires at least an agent, and an environment, but if the trials and steps are not specified, appropriate defaults will be selected based on the environment.

5.3.2 Configuration

The purpose in each of the trials is to expose those aspects of each Common Module that have the following characteristics; namely they must:

- be linked to an underlying micro-theory of cognition implemented in the architecture;

- control some aspect of how that module functions; and
• be accessible through an implementation of the Common Model interface.

Each Common Module was assessed and evaluated in turn, using those characteristics that were successfully mapped across architectures, as described in section 5.2.

5.3.3 Data Collection

Data is collected from two primary sources: run-time calls through the CMC interfaces, and output logs generated by each agent.

The CMC interface includes collectors for each of the implemented modules. Each collector is responsible for collecting metadata associated with the call (time, delay, count, etc.), as well as the values being passed by the calling function. The example interface also injects experiment-related metadata into the output stream of the user agent for time- and sequence-dependent analyses.

Second, a log recorder is used to generate and collect a transcript of the trial run, and parse out both metadata and specific values that may not be available through the CMC interface, or can only be obtained from each architecture. User agent transcripts are collected during and after run time, and put through an architecture-specific parser that looks for keywords indicative of the metric of interest.

5.3.4 Report Generation

Reporter class contains the logic to generate and display measurements from four perspectives: environmental, architectural, behavioral, and computational.

Environmental Analysis covers metrics from the virtual environment system, test suite, and experimental suite that exist outside of each architecture, but determine the knowledge-level problem or task. Environmental metrics include: observational
192

feature space and complexity; action feature space and complexity; peripheral or
input types; and transduction requirements.

Architectural Analysis deals with both general and component-specific metrics
collected from each architecture. Architectural metrics focus on include. Production
System & Procedural Memory metrics include event triggers, load and utilization,
comparison, and conflict resolution. Working Memory metrics include events from
buffers/structures matching and updates. Declarative Memory metrics include
events from chunk storage, activation, and changes in noise and salience. Visual
Perception metrics include feature states, object recognition, and processing delays.
Motor Action metrics include peripheral delays, action types, action rates, and
action frequencies.

5.4 Architectural Comparison

The architectural comparison demonstrates how the theory of the Common Model
is the code itself, versus a purely abstract concept without grounding in tangible
representation, but only up to a certain point.

These experiments are designed to confirm and elicit the similarities and
differences in run-time operation between common model components defined by
the CMC, and more importantly, identify any requirements that are being violated.

The choice of simple counting example is intended to support a discussion about
cognitive components without interference from ill-defined perceptual or motor
configurations. The task model is designed to employ all core functions, namely the
Production System cycles, Production Memory rules, Working Memory buffers, and
Declarative Memory facts. This experiment is meant to confirm that all models
operate according to a common clock cycle, and that default settings may expose
differences in parallel vs serial processing on working memory elements.
The Meta-Model requirements to verify are:

- cognitive clock rate is 50ms
- production rule use of working memory is serial between decisions
- declarative memory recall supports sequential actions

The CartPole example is intended to support a discussion about rudimentary motor processes and action selection in an environment that does not support no-operations or pausing. The task model is designed to employ a semblance of perceptual and motor module functionality and demonstrate the inherent delays in motor response, if any exist.

The Meta-Model requirements to verify are:

- perceptual input transformation
- action selection rate
- motor output transformation
- motor output delay

The Beacons test example is designed to test agent perception and mobility in an isometric environment, with periodic delays between action selection and feedback. The task model is designed to test more complicated manipulation on two dimensional objects, and image tracking. The complexity of this test is beyond the scope of the CMC requirements, and only used to identify possible areas the CMC may be expanded based on the tested architecture capabilities.
5.4.1 Production System

The Production System results demonstrate that even though activity rates differ substantially across architectures, they stay in sync while performing the same tasks in the same environment, tracing the original CMC theory of 50ms cognitive cycles, through the generated interface, and instantiated as model code. The CMC Theory on cognitive cycles is evident in the activation trace of clock cycle steps across trials, while at the same time, the theory is not the code because of how different they are during activation at certain phases.

Production System test was conducted using a Counting test that provided each agent with a starting and target number in WM, and sequential steps for each subsequent number in DM. Each agent ran for five (5) trials, at a production cycle rate of 50ms per step, for a maximum of 10 steps per trial.

Fig. 5.1.: Production System activity rate (per cycle) in Counting Model
Production System activity rate is defined here as the firing and activation rate of WM elements associated with the proposal, evaluation, and selection of specific production rules. The frequency of elements are derived from log data recorded during execution across trials for each architecture, filtered for specific key words, and then aggregated into a time series with sub plots sharing the same time axis.

In Figure 5.1 we see a broad pattern of large activity spikes (mostly activity associated with model re-initialization), followed by smaller spikes of activity during each of the trials. The Soar architecture maintains a consistently high activation rate throughout, which is expected given the parallel nature of operator proposal and firing during decision and action phases of the decision cycle.

Lisp ACT-R is also quite verbose, however also includes an oscillation between production rules due to the task implementation.

The Python ACT-R activity rate is comparatively low, since the CCM implementation allows direct and instantaneous updates to the goal buffer without activating any specific production rule. Also, Python ACT-R activates the buffer less, and is not implemented to accommodate BOLD calculations as much as Lisp ACT-R, and thus appears inconsistent before, during, and after trials compared to Lisp ACT-R on which it is based.

The obvious oscillations between cycles demonstrates the conformance of a lockstep activity that does not drift out of phase, indicating adherence to the cognitive cycle during task execution. However, the CMC Theory is not considered part of the code once we dig into why production-related activity is firing at different rates during each cycle.
5.4.2 Working Memory

The Working Memory activity is based on the same task configuration as Production activity, by performing 5 trials of sequential counting with 10 steps per trials. WM activity is defined here as writing to a global work space represented as buffers (in ACT-R) or top-level graph (in Soar). All architectures conducted the task in $n = 55$ steps. Lisp ACT-R had 0.09 $\pm$ 0.29 per (50ms) cycle, Python ACT-R had 0.8 $\pm$ 1.08, and Soar had 37 $\pm$ 6.7 per cycle.

![Working Memory Activation](image)

Fig. 5.2.: Working Memory activity rate (per cycle) in Counting Model.

Figure 5.2 shows how writing to buffers in ACT-R can only happen at specific intervals of the production cycle, whereas Soar can write to WM multiple times during a single decision cycle until no more rules can be evaluated or fired.
5.4.3 Declarative Memory

Declarative Memory activation here is based on the same task configuration as Production and Working Memory activity, by performing 5 trials of sequential counting with 10 steps per trial (Figure 5.3). DM activity is defined here as a store or request in the DM buffer (ACT-R) or by specifying a `smem.store` (using LTIs) or `smem.queue` (using STIs) commands in Soar. All architectures performed $n = 55$ steps, with Lisp ACT-R having $1.5 + / - 0.63$ requests per (50ms) cycle, Python ACT-R making $0.98 + / - 0.4$ per cycle, and Soar making $0.9 + / - 0.29$ per cycle.

![Declarative Memory Activation](image)

Fig. 5.3.: Declarative Memory activity rate (per cycle) in Counting Model.

DM activity in both Lisp ACT-R and Python ACT-R are very consistent, with an obvious lapse in both architectures between trials. The Soar architecture does not (by default) “harvest” buffer chunks, treats all memories retrieved from SMem as persistent elements within the same state, so new results do not show up after the
first unless the graph structure is either manually purged, or queried between decision cycles.

5.4.4 Perception and Motor

Considering the lack of uniform support for visual interfaces across architectures, the Perceptual interface of each agent consisted of a custom transformation function in the perceptual module and written directly to the goal buffer in ACT-R and input-link in Soar. Also, due to the lack of uniform support for motor interfaces across architectures, the Motor interface of each agent consisted of a custom callback function to capture key press commands in ACT-R, or commands placed on the output-link in Soar.

The CMC would benefit from standard `Perception::transduction()` functions to map feature spaces (for both perceptual Observations and motor Actions) to internal component representations. The OCL validation would likely be for type `FeatureMap` with a pre-condition of novel stimulus or action updates.

OpenAI Gym

The Perceptual and Motor evaluation task made use of the CartPole virtual environment provided by the Gym OpenAI. The task was performed across 5 trials, with a maximum of 100 steps per trial. The task model loaded by each agent was based on a naive heuristic to move the cart back towards the center of the line.

Lisp ACT-R motor activation rate was $0.34 + / - 0.47$, Python ACT-R was $0.47 + / - 0.69$, and Soar was $0.61 + / - 0.49$ per cycle. The near-uniform maximum across agents is evidenced by the maximum threshold of one action per deliberate act cycle. Lisp ACT-R enforces this through the use of a virtual device associated with the
motor buffer. Python ACT-R enforces this through hard-coded delays in the motor class, and Soar enforces this by waiting until the output phase of the decision cycle to operator applications on the \texttt{io.output-link}.

**Beacon Trace**

The StarCraft Beacon virtual environment mapped through the the Gym OpenAI is designed to test more complex visual and motor stimulus response. The task is composed of a single agent placed in a random position on the screen, and a beacon that the agent must walk to, initiated by locating the beacon, and clicking on the beacon. The task was performed over a maximum of 100 steps per model. The task model loaded by each agent was based on a naive heuristic to identify the x- and y-coordinates of the beacon, and register a simple click response using a mouse peripheral (or approximation).
Lisp ACT-R (Figure 5.5) did not register any visual activity, possibly due to misconfiguration of the supplementary Vision module. Lisp ACT-R motor activation rate was 0.54 +/- 0.71 actions per cycle, with a mean Actions-Per-Minute (APM) of 648.

Python ACT-R (Figure 5.6) visual activation traces remained uniform throughout the trial. The lack of visual update response may be due to a mis-configuration of the visual Module buffer. Python ACT-R motor response was 0.49 +/- 0.75 per cycle, with a mean APM of 588.

Soar visual activation traces (Figure 5.7) did not register any visual activity, possibly due to the reliance on the io.input-link vs objects registered in the Spatial-Visual-System (SVS). Soar motor activity was 19.69 +/- 2.26 per cycle, with a mean APM of 23628.

The difference in motor output between ACT-R implementation is minimal, with mean APM values that remain at the upper bound of professional human players.
Fig. 5.6.: Python ACT-R Activity Frequency by Module in Beacon Test

Fig. 5.7.: Soar Activity Frequency by Module in Beacon Test
The Soar architecture produces an upper bound consistent with competitive AI agents discussed in Peters et al. (2020), probably due to the rate of command updates to the output-link during a decision cycle, independent of what is sent to the environment at the end of each cognitive cycle. These results confirm the ability of each architecture to represent motor commands in Working Memory buffers, and send them to actuators (or equivalent), however, the commitment to underlying theory in implementation (e.g. Fitts’ Law) is obviously not the same. The CMC, in this case, should be more specific on representation constraints to ensure the Perceptual and Motor interfaces align; for example, by specifying a $\text{max}_\text{apm}$ attribute on the Motor module, and in Soar’s case, rate limiting command updates to the output-link during action phase elaboration. OCL validation could consist of a derivation rule for a rolling average across cognitive cycles.

5.5 Summary

In conclusion, we have reviewed two different, yet equally important, approaches to implementing the Common Model of Cognition as a formal Meta-Model for cognitive agents, as well as demonstrating the conformance of each agent to the specifications described by the CMC while operating through a common point of reference measured by the generated framework.

Implementing a Common Meta-Model of Cognition for each cognitive architecture demonstrated the areas of obvious (and sometimes not so obvious) functional overlap, indicating there is room for expansion in the CMC for more specific requirements and constraints. Functional areas of consideration include a stronger specification for Visual and Motor module integration with external environments. This implementation also demonstrated some areas that were difficult to reconcile, or presented outright conflicts during implementation.
Conflicts included differences in terminology for, and standardizing across Working Memory elements, as well as differences in serial- and parallel-processing within Production System decision cycles.

Running the models in the framework demonstrated both similarities and differences in the logical structure of the original Standard Model, confirming agreements stated in the original publications, but also demonstrating the stark differences during task performance. While running a simple psychometric counting test (sans virtual environment), all agents remained in “lock-step” according to a common cycle, demonstrated consistent memory retrieval delays, and called into question the major differences in working memory utilization during a single cognitive cycle. Running the agents in the same virtual environment demonstrated the feasibility of a uniform mapping across architectures for 3rd party test environments. These tests also tangibly demonstrated that a practical application of Common Model Theory through a Meta-Model-based interface can provide the qualitative and quantitative rational for evolving the specification behind the Common Model of Cognition.
6. DISCUSSION AND CONCLUSION

The results of this thesis demonstrate the applicability of creating a formal Meta-Model of Cognition that can encapsulate, in the most basic form, the structure and requirements for cognitive agents based on Common Model architectures. With these results, we also now understand that we must include new key elements in the CMC to continue progressing towards a defendable Standard Model of Cognition capable of integrating and explaining the theoretical foundations upon which they are based; such as statement also begs the question of how I arrived at this conclusion.

6.1 Contributing to a Lakatosian Research Programme

Cognitive Psychology historically relies on a combination of empirical and Popperian falsification (1935) to measure progress. According to Popper, a scientific theory is any theory than can be falsified. Characterized by the method of induction, as well as the understanding of the limits of induction (by using finite example instances); a purely Popperian methodology relies on theories making predictions, and empirical work supporting or refuting those predictions (and the theories on which they are based). Popper later adopted the view that science proceeds through the experimental testing of “bold conjectures” (Popper, 1965). According to Newell (1990), falsification alone is “inappropriate for the development of [Unified Theories of Cognition]”, and that a more appropriate model for theory development is provided by Lakatos’ cumulative research programme (Lakatos 1970). Indeed, traditional methods may not be a good fit for Cognitive Models
implementing and testing Theory as Code, as not all developing micro theories can be systematically falsified.

Lakatos outlined the importance of a Demarcation criteria used to distinguish between endeavors that generally agreed to be scientific, versus those the majority are hesitant to endorse; those endeavors that warrant merit must be comprised of a hard core of theories that the community agrees to be the substance and standard of the work, with a periphery of theories that are being evaluated. The evolution of Cognitive Science demonstrates a strong emerging consensus of overlapping theories stemming from historically-disparate academic disciplines. Lakatos’ cumulative research programme, as embodied in the application of Cognitive Modeling, evolves the micro-theories (the hard core) upon which the Common Model of Cognition is based; these theories have been implemented across different cognitive architectures, and demonstrated against ample human evidence. Yet, all architectures also make malleable theoretical commitments through choices of development and implementation — as one must when testing a theory as code. How do we balance the two?

There are constraints on cognitive modeling that stem from functional representation vs a theoretical simplification. The functional representation is a pressure to add more features that can model human behavior, while at the same time, theoretical simplification pressure to simplify, reduce number of possible models that can reproduce the same behavior. In order to make progress in the Lakatosian sense, we have to have a theory about a function (e.g., processing, storage, learning, etc.) to be able to constrain it, and then implement an instance of this function to confirm the theory without diluting results through additional, and unnecessary, functional representation.
A formal Meta-Model of Cognition, described as a formal Model-driven system, captures the essence of the theoretical commitments, without prescribing the unnecessary functional representation. Successive generation of architectures, accompanied by user models which put new theories into effect, is a practical exercise of periphery Theories as Code, without forcing a distributed research programme to adopt the functional limitations of that implementation (as we saw in specific architectures), while providing the opportunity to evolve and promote those theories into the inner core through modification of the Meta-Model from which they are derived.

6.2 Implications and Conclusions

The results of this thesis demonstrate both consistencies, and inconsistencies, between and within the Common Model of Cognition. The successful mapping of architectures to each CMC module revealed a plethora of common touch points across architectures that could be confidently mapped to their constituent representation in the Meta-Model. This mapping process, however, also exposed many inconsistencies in terminology, ontological assumptions, and the run-time effects of catering to strong theoretical commitments outside of the Cognitive System Level. These findings provided insight into plausible updates to a newer instantiation of the Meta-Model of Cognition.

6.2.1 Interface Generation

First, the results of this thesis indicate that a Model-driven approach can be used to transform and generate a formal Meta-Model of the CMC into an integrated framework for model validation and comparison. We reviewed a perceived lack of
specificity in the current CMC, due in part to the lack of consensus outside of the core capabilities defined in most architectures (namely the Production and Memory systems), and inability to derive commonality from examples applying different architectures to the same environments.

Cognitive agents are to perceive and act upon an environment, and ensuring like-for-like interfaces are supported and used by candidate architectures. The metamodel was used to generate a framework capable of interfacing between different feature spaces and complexities for arbitrary environments. A model driven approach, then, can be used to transform an abstract model into a generated template, and from there into a platform-specific model interface. A user that has a specific environment in mind can use this interface to generate what is required to test a user model in that virtual environment. Further, an example formal specification of the mapping between perceptual and motor modules and the virtual environment provides specific and limit interface points for observation and action spaces supported by that environment.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td><code>load()</code> operator for initialization.</td>
</tr>
<tr>
<td>Model</td>
<td><code>cycle</code> value and initialization validation.</td>
</tr>
<tr>
<td>Model</td>
<td><code>step()</code> operation and validation.</td>
</tr>
<tr>
<td>DeclarativeMemory</td>
<td><code>addDM()</code> operator.</td>
</tr>
<tr>
<td>DeclarativeMemory</td>
<td><code>getDM()</code> operator.</td>
</tr>
<tr>
<td>Perception</td>
<td><code>observation</code> attribute and validation.</td>
</tr>
<tr>
<td>Perception</td>
<td><code>setObservation()</code> operator.</td>
</tr>
<tr>
<td>Perception</td>
<td><code>getObservation()</code> operator.</td>
</tr>
<tr>
<td>Motor</td>
<td><code>next_action</code> attribute and validation.</td>
</tr>
<tr>
<td>Motor</td>
<td><code>setNextAction()</code> operator.</td>
</tr>
<tr>
<td>Motor</td>
<td><code>getNextAction()</code> operator.</td>
</tr>
</tbody>
</table>

Table 6.1.: Updates to Meta-Model of Cognition
The implications of these results allows modelers to define the abstract requirements for their models, and then use a model-driven platform to generate the stuff the interfaces that are already known to work between architectures and environments.

6.2.2 Model Comparison

The results of this thesis also demonstrates that we could potentially use data measurements that quantitatively describe the characteristics and attributes of user agents to validate candidate architectures. For example, the design- and run-time environment presented by each cognitive architecture places functional constraints on the model while evaluating a specific theory. Some model files that are written (or generated) for them can be validated against the CMC meta-model specification based on the inputs for specific variables like clock speed, activation rates, etc. before model file execution. Second, the generated framework can be used to implement component-specific interface points to collect metrics during model execution, such as the Production System clock speed, outstanding event queues, or perceptual and motor requests to specific buffers.

When comparing across cognitive architectures, we demonstrated how the fundamental components can be qualitatively and quantitatively contrasted, however, there is a limit to this analysis, as functional requirements, assumptions and implementation-specific details such as underlying data structures for short- and long-term memory (also unspecified by the CMC) makes a formal analysis difficult or impossible.

The use of virtual environments, such as OpenAI, and a generated interface such as the Metaverse, for example, can be used for rapid prototyping and testing cognitive agents. The results demonstrated OpenAI virtual environments as an effective avenue,
as they avoid potential conflict of interest associated with tests developed by one modeling lab, and hold a lot of promise for evaluating human cognition without the limitations of physical interfaces.

|     | A1 | A2 | A3a | A3b | A4 | A5 | B1 | B2 | B3a | B3b | B4 | C1 | C2 | C3a | C3b | C4 | C5 | D1a | D1b | D1c | D1d | D2a | D2b |
|-----|----|----|-----|-----|----|----|----|----|-----|-----|----|----|----|-----|-----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

Fig. 6.1.: CMC Validation Results

6.2.3 Updating the Meta-Model of Cognition

Based on the results of this thesis, I would also argue that all major components of the Common Model can be modeled using a Common Meta-Model, and that the Common Model of Cognition should be more specific than it is today. Examples include the demarcation between abstract components that are common across architectures, and the reservation of certain ontological issues, such as buffers and chunks, for future reconciliation. An example Meta-Model of Cognition “CMC 2.0”, including suggested updates, is detailed accordingly in Appendix F.

The current mapping addresses the lack of formal evolution of the Common Model definitions; parts of existing cognitive architecture models should be interpreted and represented as their abstract Common Model equivalents, and upon reconciliation, form an expanded Standard Model of Cognition. This approach was demonstrated by creating a meta-model definition that reflects the specification of the CMC, and by generating a formal specification that can be compared and validated between different versions of the meta-model definition. Our candidate meta-model was used to specify a model template common across multiple architectures, and then generate
a more specific interface template that was eventually customized for Lisp ACT-R, for Python ACT-R, and then for Soar, and yet remaining faithful to the abstract specifications used to generate it.

These results demonstrated how the CMC is a good way to generalize the various cognitive models, and should continue to be used to better understand human cognition. These results, however, also suggest areas of improvement and refinement for an example Meta-Model of Cognition:

Production System

The underlying implementations of what happens during production cycles are different, however, all candidate architecture support some form upper limit. If both of the external environment and user model are properly synced, then the CMC Meta-Model could support an abstract cycle() command defaulting to 50ms.

All of the candidate architectures implement a form of loading production rules from a named file, however, also support an abstract run() command to allow a modeler to run for a specified amount of time, if the experiment is designed for it.

The CMC Meta-Model should add the follow specifications:

- Model::run() operation and imperative specification.
- ProductionMemory::addPM() operation to add new rules at run time.

This conflict could be ameliorated by mandating a new requirement for a maximum number or duration of decision cycles per cognition cycle. I did manage to successfully implement this by mapping the Model.cycle attribute to Soar’s DC time value. This is not a constraint in the CMC, and therefore just because I can map it this way does not necessarily mean that I should map Soar in this fashion until the CMC specification is updated.
The CMC Meta-Model, therefore, could add the following specification:

- `Model.decision_cycle` attribute and initialization validation (if applicable)

**Working Memory**

All of the candidate architectures expose a method to initialize and update working memory elements, both at design time, and during run time. This functionality can be wrapped by a CMC framework, such that model initialization of the starting state can be done without resorting to external sources. All tested architectures support some method of user-defined WMEs, therefore the CMC should specify an `addWM()` function for the WM module.

ACT-R represents WM buffers as chunks as a list of arbitrary slots that may or may not have a chunk type associated. Soar represents WM as a graph structure with a rigid ontology of attribute-value pairs. Storing and manipulating WM elements between architectures cannot be done with the current implementation, however, an ontological mapping could be performed at either design- or run-time using a technique employed by Herbal (Paik et al., 2009).

Therefore, the Meta-Model of Cognition could also include the following specifications:

- `WorkingMemory.wmchunks` relation for reference and validation tests.
- `WorkingMemory::map()` operator to transpose chunks and triplets.
- `WorkingMemory::addWME()` operator.
- `WorkingMemory::getWME()` operator.
Declarative Memory

All tested architectures support user-defined additions to DM storage. While the underlying structures may be different, the example implementation support a chunk-oriented instantiation for storage and recall.

Therefore, the Meta-Model of Cognition should specify and constrain the following elements:

- `DeclarativeMemory.dmchunks` relation for reference and validation tests.
- Update `DeclarativeMemory.addDM()` operator with pre-operation conditional on `.dmchunk` relation.

Perception and Motor

All tested architectures, including the vision systems included and available, assume some form of representation pre-processing before they are invoked. However, even if the vision systems are not used, the representation requires temporary storage in the visual (or equivalent) WM buffer. Therefore, a CMC Perceptual Model should support a user-defined mapping of perceptual stimuli presented by the environment, such that it can be referenced and processed by procedural rules specified by each architecture’s internal knowledge-level representation.

In all cases, motor events fired during a single cognitive cycle step, therefore, the implemented motor module stored the result of the callback for later presentation to the environment. A Common Motor Model could therefore specify an abstract representation of the action specification, and even implement a `next_action()` (or equivalent) function to retrieve this information at the end of each cognitive cycle, regardless if the last cycle resulted in an updated action.
The CMC would also benefit from standard `Perception::transduction()` functions to map feature spaces (for both perceptual Observations and motor Actions) to internal component representations. The OCL validation would likely be for type `FeatureMap` with a pre-condition of novel stimulus or action updates.

The CMC should be more specific on representation constraints to ensure the Perceptual and Motor interfaces align; for example, by specifying a `max_apm` attribute on the Motor module, and in Soar’s case, rate limiting command updates to the output-link during action phase elaboration. OCL validation could consist of a derivation rule for a rolling average across cognitive cycles.

The CMC Meta-Model should therefore add the following specification

- `Perception::obsSpec()` and related `.FeatureMap` attributes.
- `Motor::actionSpec()` and related `.FeatureMap` attributes.
- `Motor.max_apm` attribute and derivation rules.

The updates discussed in this section would make the CMC Meta-Model better reflect the interface definitions that were successfully mapped to the Requirements. The next section discusses changes that could also be applied directly to the architectures tested in this thesis.

6.2.4 Updating Common Model Architectures

A Standard Model of Cognition will not emerge by simply updating a list of requirements in a formal specification; different Architectures that commit to adopting the core theoretical tenets (and their requirements) will have to abandon some theories, and adopt common ones, over time.

Table 6.2 outlines some suggested changes that the tested architectures could implement in order to better align with a CMC standard, resulting in additional
areas of overlapping support. The following sections discuss these suggestions in greater detail.

### Production System

The CMC needs to clarify between the use of terminology associated with memory storage of facts (i.e. *chunks*) and learning new production rules (via *chunking*) before a formal model could properly an *unambiguously* represent these concepts.

Another difference in Production Systems is how they manipulate Working Memory during a single cognitive cycle, and Soar's consideration for unlimited elaboration cycles outside of production rule evaluation during parallel processing. If the CMC would standardize on single computations during production rule matching, it would break Soar's ability to arrive at quiescence. Alternatively, if the CMC would explicitly require parallel evaluation over all productions (and their associated effects on WMEs), then the ACT-R theory would require adjustments to accommodate additional inline processing.

Soar is the only architecture that allows for side-loading new production rules after model initialization (at run-time) without relying exclusively on generating new

<table>
<thead>
<tr>
<th>Module</th>
<th>Arch</th>
<th>Suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prod System</td>
<td>All</td>
<td>Standardize use of chunks vs chunking.</td>
</tr>
<tr>
<td>Prod System</td>
<td>ACT-R</td>
<td>Run-time addition of rules.</td>
</tr>
<tr>
<td>Prod System</td>
<td>Soar</td>
<td>Decouple sub-goaling (PSCM).</td>
</tr>
<tr>
<td>Working Mem</td>
<td>All</td>
<td>Standardize slots, attributes, operators, augmentation, etc.</td>
</tr>
<tr>
<td>Declarative Mem</td>
<td>Soar</td>
<td>Standard interface for STI/LTI storage/recall.</td>
</tr>
<tr>
<td>Declarative Mem</td>
<td>All</td>
<td>Ontological agreement between triplets and slots.</td>
</tr>
<tr>
<td>Perception</td>
<td>All</td>
<td>Standard mapping for intermediary object and signal transduction.</td>
</tr>
<tr>
<td>Motor</td>
<td>ACT-R</td>
<td>Decouple use of peripheral devices.</td>
</tr>
</tbody>
</table>

Table 6.2.: Suggested Updates to Architectures
rules through compilation. If other architectures were able to support this feature, the Meta-model could enforce stricter requirements for `ProductionMemory::addPM()` and `getPM()` operators.

And finally, we address the pink elephant in the room. There is a major difference in the level of commitment to a specific reasoning strategy; namely, the Problem Space Computational Theory, as implemented in Soar. This commitment is at the heart of Soar’s processing cycle, is used to resolve impasses, generate sub-goals, and ultimately results in the creation or update of production rules and utilities. This commitment is, however, based on a Knowledge-level problem solving strategy that is not shared, implemented by, or required by ACT-R theory. This begs the question — is PSCM necessary for a Standard Model of Cognition? If so, the CMC should standardize all architectures to use decision cycles and impasse resolution based on this theory. If not, then Soar should decouple functional components of the rule selection (i.e. the ReteNet) implementation from impasse resolution (i.e. sub-goal stack), and allow the modeler to select and/or implement any Knowledge Level problem solving theory of interest. This decoupling of systems levels only serves to strengthen a Standard Model of Cognition without unnecessary bleeding between levels; if PSCM is the closest approximation to human-like problem solving, then rest assured, it will emerge as a de-facto micro-theory in a Standard Meta-Model of Rationality.

**Working Memory**

All of the candidate architectures expose a method to initialize and update working memory elements, both at design time, and during run time. This functionality can be wrapped by a CMC framework, such that model initialization of the starting state can be done without resorting to external sources. All tested architectures support
some method of user-defined WMEs, therefore the CMC should specify an \texttt{addWM()} function for the WM module.

Both ACT-R and Soar define memory components as chunks, and agree that a chunk representation is used to \textit{justify the conditions} by which a rule is fired. However, this agreement quickly breaks down when writing an interface represented as standard data structures (e.g., Integers, Floats, Booleans); it does matter when establishing a standard vocabulary when comparing symbolic elements in a conceptual data structure. This issue must be reconciled before elementary Chunks, Slots, Attributes, Operators, Augmentations, etc. can be implemented (and possibly mapped) between architectures.

\textbf{Declarative Memory}

In order to map new DM operators based on Soar, the Meta-Model class would need to be inherited by Semantic and Episodic classes with updates to the operator contracts, with pre-conditions on the use of query-based or non-query-based STIs and LTIs accordingly. The easier solution, however, would be to continue mapping Soar to the CMC, with an implementation detecting temporal elements and modifying the use of Smem and EpMem stores depending on how they are called, and/or what they are passed as operator values.

Successfully mapping Declarative Memory content representation poses the biggest challenge. It may be possible to alter the structure and ordering of either ACT-R chunks and slots, or the configuration of attribute-value pairs (also termed \textit{augmentations}) in Soar, in order to represent, store, match, and recall equivalent semantically-equivalent knowledge across candidate architectures.

If the CMC Meta-Model changes to accommodate Soar, other architectures will need to explicitly accommodate sub-modules for Semantic and Episodic Memory
stores. If not, Soar would need to provide an ontological mapping for general operators.

**Perception and Motor**

All tested architectures, including the vision systems included and available, assume some form of representation pre-processing before they are invoked. However, even if the vision systems are not used, the representation requires temporary storage in the visual (or equivalent) WM buffer.

All Common Model architectures will need to agree upon a standard for intermediary object mapping during signal transduction, as well as a observation specification mapping to internal representations based on “what” and “where” information, where appropriate. Also, due to the lack uniform support for motor interfaces across architectures, the Motor interface of each agent should agree upon the command types supported vs forcing the modeler to chose a specific peripheral (i.e.: ACT-R keyboard or mouse), or having to rely on implementing their own (i.e. Soar output-link).

### 6.3 Limitations of this Research

The research presented in this thesis may have limitations on generalizability, or reproducability, due to faults in the research methods or controls, such as the design and development of model-driven frameworks, the implementation of mappings between framework and architecture, issues with the tools and toolchain used to generate the results, limitations on internal or external validity of results, limitations on controls, and limitations on the generalizability of the results.
6.3.1 Technology and Standards

The development environment used for the majority of this research was based on open source software technologies provided through different consortium, such as Eclipse, as well as open source standards bodies such as the Object Management Group. These technologies and standards, although historically robust from years of application and refinement, may contain errors in the code or methods used to describe and deploy them in practice. Any errors associated with these systems would impact the validity of results in this thesis.

Second, the practical implementation of the generated framework, the virtual test environments, or mapping between them, may not follow current standards or best practices documented by the authors of each respective technology. The research methodology and results presented in this thesis was conducted or a span of time that saw an evolution in the Python libraries used, bug fixes in the OpenAI interface, and the sophistication of modeling technologies, such as EMF and PyEcore implementations of the MOF standard, that are each responsible for various phases of meta-model and model to text transformation and generation. An improperly configured tool in this toolchain could potentially threaten claims to “correct by construction” advocated in discussions on Model-Driven Design methodology.

6.3.2 Research Methods

Potential limitations on internal validity would stem from incorrect assumptions on the relationship between between documented interfaces in each architecture, or the causal relationship between architecture and task model settings, and the expected output while running model traces. For example, architecture documents provide
a non-exhaustive list of function parameters that can be used to specify settings critical to model performance, such as cognitive cycle time, memory storage and retrieval delays, and watch levels for output. These variables, if improperly used or configured, may threaten the validity of the variable data collected and reported against task model testing.

Limitations on external validity may be an issue if the results reported in this thesis could not be generalized to a wider audience, could not apply to different architectures that fit the Common Model criteria and assumptions, or apply to test cases that rely on similar mapping to virtual environments. For example, the CMC Meta-Model definition and framework were based on a combination of personal interpretation, and consultation with other expert researchers in this domain; however, if the interpretation is not correct, the model-to-text transformations could not be generalized. Alternatively, the functional mappings that were applied to the candidate architectures may not be extended to other architecture interfaces, even if those architectures comply with the Common Model criteria; therefore, any suggested changes for a more specific Common Meta-Model of Cognition would not be generalizable to the wider cognitive modeling community, and inhibit replication of results using alternative systems or tasks.

Limitations on controls would be caused by a biased or incorrect selection of cognitive architectures, task models, or virtual environments used to demonstrate model-driven design and generated results. For example, the choice of task models and virtual environments were based on a combination of my personal and professional familiarity with these domains. To control task model bias, Psychometric task models were selected from base cases found common between candidate architectures, with comparison between sample task model code to ensure non of the candidates were given an advantage. Virtual environment task models were based on the OpenAI API
standard, with confirmation that specific virtual environments used in this research were authored and implemented by researchers or organizations with no formal ties to any of the cognitive architecture authors or labs.

6.4 Recommendations for Future Research

This thesis covers a lot of ground between different applications, methodologies, and topics. While a fully comprehensive evaluation of the theories, methodologies and applications would be desirable, it is simply not feasible due to limitations of time and space allocated to a graduate program. The work, however, does not have to stop here (nor should it), and there are many avenues for potential exploration in the future. Pertinent topics that may warrant additional review include, but are not limited to: the application of the CMC Meta-Model to other cognitive architectures; the potential integration of different Common Model architectures into hybrid configurations using a Standardized Messaging Bus inspired by the CMC Meta-Model and framework; and perhaps investigating the formal definition and model-to-model transformation between theories defined and implemented at the Knowledge or Neural Systems Levels.

The original Standard Model papers included a plethora of suggested changes, mappings, and revisions from various research labs credited for the creation of equally prolific architectures (see Appendix C.2 for details). Many of these architectures would be excellent candidates for further exploration, since they may overlap with or implement the majority of characteristics and requirements defined by the CMC; examples include, but are not limited to: an exploration of mapping Sigma, which is largely inspired by Soar; the cognitive functions implemented in SPAUN (Nengo), the CLARION architecture, the EPIC architecture, the INCOG framework, and further
mappings to fMRI data and Regions of Interest derived from the Human Connectome Project \cite{Stocco2021}.

The CMC Framework, mapped to multiple architectures, implements a standard interface for message passing between experimental models and virtual environments. We discussed some of the potential mappings between elementary symbols and tokens that, while existing at the Knowledge level, may be stored, and retrieved using a transformation to map between ontologies. Further, we also explored and discussed many of the underlying data structures that can be queried through the API provided by each architecture, and the indication that these sub-systems can work in “lock-step” according to a central clock that exists outside of all of them. If functional modules can be accessed through the framework, and symbolic representation of data transform between ontologies represented by each architecture, then a framework based on the CMC could be used as a messaging bus to communicate between these architectures, or specific sub-systems; this application would enable the use of hybrid architectures without mandating any structural changes to the underlying code within them.

And finally, the definition of a Meta-Model that exists at the Common Model of Cognition, using standards designed for transformation between meta-model levels, and across domains, invites future researchers to consider creating a similar Meta-Model definition for System Levels that sit adjacent to the Cognitive Level; namely the Knowledge Level, and the Neural Level. According to Newell \cite{Newell1990}, the power of a system abstracted by levels of complexity stems from the insulation of each level from the one below it, as well as the prescriptive power of a higher level for the one below it – indeed, according to \cite{Newell1990,pg,49}, the requirements of a Cognitive Architecture could be defined by a Model at the Knowledge level. Future research in this area could include various approaches to knowledge
representation and problem solving, such as GOMS and Socio-Technical GOMS theory, different applications of Problem Space Computational Models (beyond Soar and Sigma), Multi-space Models, etc. In addition, the process of following a Model-driven approach could be extended to the Neural level, to provide a mapping between, and definition for, CMC functional components implemented in neural networks, and mapped to various Regions of Interest. Example applications could be a Model-Driven design and standard for biologically-plausible implementations such as Neural Engineering Framework (on which SPAUN is built), and more traditional (and popular) implementations that do not follow the rigours of biology or specific dynamical systems theories. This research would contribute to, and benefit from, a formal mapping and understanding on the characteristics and requirements of each system level, provide avenues for collaboration between projects operating at various system levels of complexity, and support larger research programmes that feed requirements and results between research clusters.

6.5 Conclusion

In conclusion, we have presented a discussion on the results of this thesis, namely the application of the Model-Driven Design to the current Common-Model of Cognition. We discussed the merits of adopting and contributing to a Lakatosian research program, the implications of mapping the CMC to a Meta-Model of Cognition, the limitations of this research approach, and possible future avenues of exploration.

Contributing to a Lakatosian Research Program is necessary when dealing with theories applied as code that are not immediately falsifiable. The Common Model of Cognition is based on Assumptions derived from converging and unified applications
of micro-theories, and by virtue of this consensus supports a hard core of theories for a formal system.

Generating an abstract interface based on the CMC as published raises some interesting theoretical questions about the applicability of this approach to existing architectures. We found that comparing cognitive models using this interface demands the addition of supporting features for the most basic implementation on each architecture.

The successful mapping of common classes, attributes, and operators provided a sound foundation to establish a formal Meta-Model of Cognition, and also presented an opportunity to recommend updates to the formal Meta-Model. The conflicts in mapping could be considered an obstacle, but at the same time outlined areas ripe for continued conversation and hopefully outline opportunities for future convergence should these architectures converge in implementation.

This research is not without possible limitations, including use of technology based on current standards. Technology changes very quickly, and therefore research results based on a specific technology is less relevant just as quickly. The methods presented in this thesis were designed and implemented to transcend the limitations of a specific technology, but there will always be certain aspects that are not immune. The Standards upon which the Model-Driven Design methodology and application were based will eventually change, however, considering the historical track record of the standards committees involved, and the ongoing use of these standards by the research community, it was deemed a risk worth taking!

The results presented in this thesis may be a novel application of one methodology to a budding community, however this research barely scratches the surface of what could one day contribute towards a more formal Standard for Cognitive Modeling. There are many interesting applications that warrant further investigation, such as
the mapping and adoption of so many other worthy cognitive architectures, other virtual environments, and a more comprehensive evaluation of the capabilities they offer.
APPENDICES
A. THEORETICAL MODELS AND PERSPECTIVES

The field of Cognitive Psychology has provided us those regularities of behaviour that we can formulate and anticipate in the typical human being, derived through a multitude of testing volunteer participants in a controlled setting. In (Miller, 1956) we find evidence that despite the extreme sophistication of the human mind, processing is limited to a finite set of discrete units of information. As we will later see, these limitations are expanded and described as micro-theories that govern some (but not all) aspects of cognitive architecture implementations. Examples of these regularities include, but are not limited to: defining the expected time to interact with a computer peripheral (i.e. Fitts’ Law) (Fitts, 1954), item recognition and semantic memory (Sternberg, 1975), and the relationship between episodic memory and autonoetic consciousness (Tulving, 1985). These and other theories have become a staple of modeling and predicting intelligent behavior in humans, and to a greater understanding of the various sub-functions of mental processing. These micro-theories, although applicable to all architectures, only manifest overtly when considering these architectures from a specific epistemic or ontological perspective or level.

A.1 Levels of Analysis

Human beings have a natural habit of dividing large subjects into levels or layers of complexity to assist with learning and understanding information in smaller, manageable pieces. After all, we are limited beings, and discussing the entire corpus of Cognitive Models in one go (or any topic of merit for that matter) would likely
result in frustration. Contributions to complex fields also follows a similar pattern, with sometimes infinitesimally small steps adding to a larger corpus of knowledge, with the occasional (and usually unwanted) side effect of never overlapping.

Simon and Newell (1973), can be credited with the widespread use of production systems as detailed models of the human control structure; a formalism for describing human-like decision making and problem solving behavior. Control structures for chronometric behavior (based on response times) were not well defined at that time, and required a more generic theory of this cognitive behavior; more specifically, a general unified theory required the inclusion of a spectrum of overlapping time scales (Newell, 1980, 1990).

Alan Newell, a leading Computer and Cognitive Scientist of his time, tackled this issue by expressing serious concern about the direction, and divergence, of theorizing within cognitive psychology. In “You can’t play 20 Questions with Nature and Win”, Newell (1973) argued against the behaviorist approach to creating disparate theories of cognition through disjoint experiments with subjects generating stimulus-response models, and the community’s habit of producing what he termed micro theories, each producing what Newell thought of as small pieces of a larger tapestry, with little effort to stitch them together. This argument evolved into his work on Unified Theories of Cognition (Newell, 1990), in which Newell attempts to define the mechanisms underlying these micro-theories.

Newell described actions in terms of System levels that are inter-dependent; the higher levels require multiple lower levels to operate (as shown in Table A.1). He also argues for time constraints of geometric systems; in the human mind, the smallest order of magnitude between levels is 10 as reflected in his Time Scales of Human Action.
<table>
<thead>
<tr>
<th>Scale (sec)</th>
<th>Time Units</th>
<th>System</th>
<th>World (theory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^7$</td>
<td>months</td>
<td></td>
<td>Social Band</td>
</tr>
<tr>
<td>$10^6$</td>
<td>weeks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^5$</td>
<td>days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^4$</td>
<td>hours</td>
<td>Task</td>
<td>Rational Band</td>
</tr>
<tr>
<td>$10^3$</td>
<td>10 min</td>
<td>Task</td>
<td></td>
</tr>
<tr>
<td>$10^2$</td>
<td>minutes</td>
<td>Task</td>
<td></td>
</tr>
<tr>
<td>$10^1$</td>
<td>10 sec</td>
<td>Unit Task</td>
<td>Cognitive Band</td>
</tr>
<tr>
<td>$10^0$</td>
<td>1 sec</td>
<td>Unit Operations</td>
<td></td>
</tr>
<tr>
<td>$10^{-1}$</td>
<td>100 msec</td>
<td>Deliberate act</td>
<td></td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>10 msec</td>
<td>Neural circuit</td>
<td>Biological Band</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>1 msec</td>
<td>Neuron</td>
<td></td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>100 µs</td>
<td>Organelle</td>
<td></td>
</tr>
</tbody>
</table>

Table A.1.: Newell’s System Levels with Overlapping Time Scales.

This perspective is also reflected in other theories that attempt to define and describe levels of abstraction in dynamical systems in order to ease analysis of their constituent parts and sub-systems, together and in part. McClamrock (1991) makes an argument against dividing up systems into specific levels without understanding the underlying questions by which we are trying to understand them. The levels of investigation must reflect the answers that we seek, and by reducing components to primitives only when decontextualizing them is necessary to understand their contribution.

According to Marr (1982) there were three fundamental levels of analysis: computational, algorithmic, and implementational. For Marr, the computational theory accounts for the main goal of what the system is trying to accomplish. The representation of the algorithm questions how the system can be implemented and defined logically to accomplish this goal, while the hardware implementation concerns the underlying physical realizability of how the system manifests in the natural world. Pylyshyn (1984) presented these same layers or perspectives as semantic (sometimes intentional), syntactic ( sometime symbolic), and as physical (or sometimes biological). Contemporary psychology also defines them as content,
form, and medium (Glass 1979), without straying too far from the purpose of investigating these systems through a layered approach.

This approach to defining and describing complex systems using levels of abstraction is ubiquitous amongst the Cognitive Sciences, as both the function and form are contextualized according to how their intended design accomplishes the goal within a period of time. Marr’s mental layers, then, can be seen as corresponding to Newell’s cognitive band, as the behavior to be defined and described is realized within similar time scales, whereas the implementation layer is relegated to the biological band.

Dawson (1998) describes these layers of analysis in terms of the explanatory power attributed to each: the computational level determines the problem being solved; algorithms, through simulation and modeling, determine the steps required to solve the problem; and implementation determines the mechanistic principles and instantiation thereof. He later added a fourth, architectural level, as an appeal to the epistemology of information processing systems, and means to address the ontological interplay between the logical algorithm and physical implementation; a highly valuable method of addressing the modular constructs that determine function through implementation (Dawson 2013).

The Biologically Inspired Cognitive Architectures (BICA) society has generated a different perspective of cognitive architectures organized into a hierarchy ranked by agent sophistication (Samsonovich 2015). Sophistication here is defined by the amount of internal processing required by the agent to produce the next action; a very simple agent is purely reflexive to inputs provided by the surrounding environment. More sophisticated agents will demonstrate goal-seeking behavior, build an internal representation of their environment, and learn to optimize their actions (Table A.2).
<table>
<thead>
<tr>
<th>Architecture type</th>
<th>The agent is capable of</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta-cognitive and self-aware</td>
<td>Modeling mental states of agents, including own mental states, based on the concept of self</td>
<td>Highest</td>
</tr>
<tr>
<td>Reflective</td>
<td>Modeling internally the environment and behavior of entities in it</td>
<td>High</td>
</tr>
<tr>
<td>Proactive or deliberative</td>
<td>Reasoning, planning, exploration and decision making</td>
<td>Middle</td>
</tr>
<tr>
<td>Reactive, or adaptive</td>
<td>Sub-cognitive forms of learning and adaptation</td>
<td>Low</td>
</tr>
<tr>
<td>Reflexive</td>
<td>Pre-programmed behavioral responses</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

Table A.2.: Hierarchy of Cognitive Architecture Sophistication

This perspective of hierarchy of computational complexity draws a relationship to Newell’s temporal complexity of system levels, as we intuitively abstract and define orders of operations to change our perspective on complexity. The sophistication of action is closer to Marr’s computational layer, and the level to which the agent can realize higher goals and objectives, and peripherally related to Newell’s Social band in which interplay with the environment is of chief concern, yet without regard to temporal factors.

A.2 Research Methods

The design and analysis of Intelligent in autonomous agents involves the evaluation and comparison of agent behavior and performance to that of other humans and machines. The methods of data collection and evaluation techniques depends largely on the overall goal and intent of the researcher; the applicability to Cognitive Science here is based on two categories; the research methods (tools) employed, and subject matter (phenomena) to be studied. This section compares and contrasts competing theories, and use their merits to justify the theoretical foundation of the dissertation.
In *The Role of Artificial Intelligence Research Methods in Cognitive Science* [Davies and Francis (2013)] call into question the current gold standard for transdisciplinary research in Cognitive Science, with special attention to the differences in approach between Psychology AI and Engineering AI, and the ways that they each contribute to the field. They argue that comparison of models should not be limited to the use of statistics only, and that model comparison shouldn’t be limited to only humans or animals. Thus, current restrictions on, and perceived value of, current modeling data sources and outcomes may end up hinder longer term growth and stifle potentially informative research outcomes. Their argument is important because, as outlined in this thesis, the field can progress from the analysis of theoretical computational philosophy without mandating a direct comparison to human-generated data. In [Davies and Francis (2013)] we find further comparisons between Psychological AI and Engineering AI based on their respective results using novel functionality, and empirical quantitative models.

According to [Davies and Francis (2013)], Psychological AI is largely dominated by the opinion that the only valid modeling processes are found in nature using quantitative methods. This is a weakness, as it limits modeling to only those processes that can be quantitatively measured in natural intelligence. They argue Psychological AI should include models that aren’t limited to just quantitative comparison with psychological experiments, and include qualitative observations; examples include dreaming, planning, consciousness, reasoning, and interactions between these subsystems. Another issue here is quantitative comparison and correlation only lends weak equivalence; strong equivalence of implementation details requires separate experimentation as argued by [Pylyshyn (1980)]. Therefore, there is a significant trade off between the modeler generating the psychological test data, or using somebody else’ data. The studies referred to in the Appendix of this
thesis relied on data generated by the author, as well as other researchers, derived from both human and machine sources.

*Psych AI with Novel Functionality* involves a modeler generating data on functionality natural intelligence has to make (hypothetical/theoretical) predictions through a process of *forward engineering*, while leaving it up to others to generate psychological data for comparison. Another way to look at this is as a “computational philosophy”, used to point experimenters in future directions. If the researcher can demonstrate it has never been done before, or in the same manner, then, according to Davies et al. “the very fact that the model worked as designed is an acceptable evaluation all by itself, even without statistical testing.” (Davies and Francis, 2013). If the model, already based on a solid architecture and domain knowledge/rules, works at all, then it inherently shows that the theory *can* work. A model without corresponding data can be compared against itself by *ablating* (removing) significant sub-components to identify the source of power. Therefore, qualitative model outputs are usually (and quite successfully) subjectively assessed using structural ablation techniques, without necessarily losing the predictive power of the model.

Cooper (2007), in *The Role of Falsification in the Development of Cognitive Architectures*, argues that a lack of direct falsifiability need not undermine the scientific development of a cognitive architecture, if broadly Lakatosian principles are adopted. This is in direct disagreement with traditional quantitative approaches, is in agreement with Davies’ *Novel* approaches to both Psychology AI and Engineering AI, and is constructively used to set new direction for burgeoning research areas, and keep researchers from getting stuck in the quagmire of recycling the same problems and test data, and therefore limiting potential breakthroughs.
The approach used by this thesis, as well as the comparison of features between cognitive architectures, aligns closely with the Psych AI with Novel Functionality approach due to the novel generation of interfaces that may explore new ways of generating and comparing cognitive models. In accordance with Lakatosian principles, new functionality can still be considered a valid contribution without direct falsifiability as human-generated data may not be available, or deemed inappropriate, for direct comparison against these methods.

*Engineering AI*, in contrast to Psych AI, tries to create intelligent agents without limitation to inspiration from natural intelligence (or that found in nature); it does not model natural minds. According to Davies et al., all Engineering AI qualifies as Cognitive Science since both occur at the *information processing* level of analysis ([Davies and Francis, 2013](#)). This can be compared to Marr’s computational and algorithmic levels of analysis, stressing the *what* and *how* about a problem the agent is solving, and less about the underlying implementational (or realized) level using natural affordances and constraints found in natural systems. The value in this approach lies in the realm of the possible — AI without the same constraints as natural intelligence can inform cognitive modelers about the boundaries of what is currently possible with optimization as the main goal, and indicate the trade off between optimality and generalizability prominently found in natural intelligence. Therefore, the primary requirement of Engineering AI as a valid contribution Cognitive Science is that it *situates* itself within the other Cog Sci sub-disciplines.

The use of Psychological and Engineering approaches to AI each have merit, and can be used to inform the boundaries between what can and should happen in models of natural intelligence. The methods employed in this thesis take advantage of both, including elements of quantitative and qualitative analysis for presentation of the results. Knowing the difference...makes all the difference.
A.3 Development Paradigms

Developing cognitive models, and the architectures that support them, are approached through a combination of both imperative and declarative paradigms, depending on the level of abstraction required. The use of various language constructs has an immediate effect on how easy or difficult it may be to decompose goal-seeking behavior into specific tasks, and to compose more sophisticated behavior from primitive procedures imposed upon model developers. However, the practical application of these paradigms can also be affected by the modeler’s approach to problem solving (namely top-down vs bottom-up), and with lasting effects; creating an architecture with specific programming languages and features both supports and restricts future models.

Imperative programming is used when the exact states or problem solving steps need to be followed. This is usually found in a procedure, of “do X, then Y, then Z”. In languages such as C and Java, this is explicit since the compiler will produce machine code that also executes commands in the same order that they are written. Imperative programming is practical and make sense when articulating state-specific behaviors, or iterating through well-defined chunks for information, but quickly becomes unmanageable in larger problem spaces.

Declarative programming on the other hand, is used when we know the exactly outputs to achieve, but doesn’t really care how they are achieved. Functional programming operates on this principle, as programs are written in terms of function calls that accept input parameters, and return a result of a predefined type. What happens inside of the function is of little concern to the programmer, with the intention of abstracting away the details of problem solving or data manipulation. Declarative programming also makes sense; we naturally think this
way, and it makes larger problems (and sub-problems) manageable. Both can be used, and both are useful in their own right.

A.4 Modeling Paradigms

There’s an issue of how much of human thought is performed in a Top-down vs Bottom-up fashion, and how this should be reflected in the models we write to accomplish specific tasks, or tests a specific theory of cognition. Take for example different programming paradigms in terms of the program execution and how it is encoded. Most programming languages can include and support multiple paradigms.

There is, quite frequently, a top-down vs bottom-up orientation in any cognitive model, similar in style to how most AI applications are written for optimization, versus serving a larger (might I suggest AGI) application for generalization. If a model with specific procedural memory is written imperatively, it is specific for that task. If environmental context is loosely matched against a desired end goal state, we are doing it declaratively for reusability.

Top-down modeling is largely declarative, and the production system is driving it. Assuming the PS always sets up the state for the next action, the modeler is content to define a particular end state, leaving production compilation and impasse resolution to find the way, with little regard to the underlying theory or implementation details. People already think in a top-down way; for example simple tasks can use a combination of productions and learning utilities, or use sequential dependency detection coming out of DM, and/or blending features. There are different ways to test a model or theory, but the details of how they are performed aren’t at the forefront.

Bottom-up modeling is largely Module-driven, with specific and precise production rules and DM chunks defined and loaded ahead of time, and the assumption there
may be other unexpected emergent behavior, but the analysis is more concerned with the journey than the end state. There are elements of this in peripheral modules as well; a vision system notices something, a simulated amygdala feels unsure, a context-sensitive alarm triggers task switching. Bottom-up features in cognitive architectures are available and specified ahead of time, and usually form a part of the theory to be tested.

But what would make a modeler sacrifice one over the other, inadvertently or otherwise? I would suggest for the same reason Object Oriented Programming (OOP) became mainstream to mediate between the two; to serve as the medium to facilitate between the problem space, task decomposition, and to abstract away the repetitive details. In a cognitive architecture, however, stacking a high-level grammar on top to compile into architecture-specific model code may not compile into models that sufficiently test the features of the architecture, forcing the model to make strong assumptions about a subset of micro-theories being invoked at run-time. Even worse, top-down compiling may rule out these features, unbeknownst to the modeler.

A.5 Hierarchical Specifications

In hierarchical systems, we can use higher levels to describe lower levels. Newell (1990, p49), suggests that “Specifications for systems are often given at the knowledge level. Every level, of course, can and does serve as a specification for the level below it.”. When Newell wrote that statement, there was no practical work higher than the Rational Band. Lieto et al. (2018) expanded on this by suggesting we could describe Macro-cognition using a different language than the Cognitive level, but that doesn’t interfere with lower level paradigms or approaches to problem solving. If higher levels can be used as a specification for the one below it, then it may be fair to say, for example, the PSCM is a (Macro-level) Rational Band specification for how the rest
of the Soar architecture deals with knowledge-level tasks, rather than a core part of it.

In [West et al., 2013], we find an argument for adding Macro-cognition as an intermediary between Newell’s Social and Rational bands, and to provide a layer of specification, for example, handling interrupts and external queues outside the scope of the GOMS theory so prevalent at the time. The Socio-Technical GOMS (or SGOMS) theory can be used to describe exactly how the macro level works, by invoking macros as a specification for lower level unit tasks. The proposed solution in SGOMS is to define macros describing how humans typically respond in an environment (alone or with others), assuming the macro is how we typically think, but not absolutely how we think.

A bottom-up approach to model design is based on a tradition of writing imperative functions and procedures to produce a very specific result based on a micro-theory. These models perform equivalently to human-generated data from task-oriented psychometric tests, while still aiming for underlying functional consistency and perhaps prediction. Many Cognitive Architectures complicate what is usually deemed a straight forward approach, since Production Systems must also employ the same characteristics as declarative programming methods and execute rules in an arbitrary order, much like Expert Systems.

This approach is “intendedly parallel” (like our nervous system), yet in reality must be debugged serially as all other software systems. This dichotomy of processing can be difficult to conceptualize. When a developer begins to use these environments, they quickly learn to read the code not only in “cognitive speak” using terms such as modules, buffers, and chunks, but also to conform to the constructs imposed by that architecture’s code interpreter, such as LISP-inspired code interpreter with a modified syntax. Rules defined by condition-action sequences are even sometimes accompanied
by prefix-notation to perform inline math, since writing declarative and procedural memory elements capable of the simplest arithmetic operations can be somewhat daunting, and usually out of scope. Many of the basic ACT-R and Soar models referred to in this thesis make use this approach to define the basic operations that must be used for a specific test, while at the same time require very explicit (and somewhat repetitive) memory declarations to perform the simplest operations, such as adding simple numbers together.

The top-down approaches seek to solve this issue through a functional paradigm by exploiting the merits of code abstraction and inheritance through Object Oriented Programming (OOP). Examples include the HLSR language constructs (Jones and Wray 2006) based on relations, transforms, and activation tables, that must be compiled for the target cognitive architecture, assuming the target computational architecture is already working. Another exemplar of top-down cross-compilation is Herbal (Paik et al. 2009); an environment that can be used to generate code for Jess, ACT-R, and Soar, and features different modes to simulate what is expected from novice and expert developers in the target environment. The existence of an Herbal compiler also assumes the there is a mapping between Herbal and ACT-R, a mapping between Herbal and Soar, and by extension, that they are the same and could support a one-to-one transformation. The merits of an Herbal-like compiler are obvious, however, is only tangentially applicable to this thesis since cross-compiling imperative functions and routines typically occurs at Newell’s Knowledge-level.

The different theoretical models and perspectives each have a role to play in the research and development of cognitive architectures and models. Each level of analysis, research method, and development paradigm, forces a researcher to take a stance on what they are going to look at, how they will look at it, and what can be
concluded from the results. Cognitive architectures also span the spectrum of systems levels, from rational knowledge, through cognitive, to neurological. This thesis is most concerned with the organization of architectures and activities implemented and studied at the Cognitive Level, and is of course limited by the constraints of taking that stance.
B. TAXONOMY OF COGNITIVE ARCHITECTURES

A 40-year historical review of the Cognitive Modeling enterprise (Kotseruba and Tsotsos, 2020) concluded there may be no “one size fits all” approach to architectural taxonomy, and instead describes them according to their respective approach to implementing broad cognitive functions (i.e. perception, attention, memory and learning). A community-led initiative to establish a Common Model of Cognition (CMC) took this exact approach to define assumptions about broad functional categories found in human cognition; the CMC outlined the functional categories, relationships, and some general terminology, however there are still recurring issues with mapping specific implementations.

The idea behind abstracting human cognition into a set of design principles is certainly not a new one, as most modern architectures can trace inspiration through the works of Alan Newell and Herbert Simon on the use of production systems for Human Problem Solving (Simon and Newell, 1971), an extension to Simon’s earlier work on a hierarchic systems (Simon, 1962), and related work on the General Problem Solver (Hunt and Ernst, 1971).

B.1 Categorization

The methodology of analysis can take different forms depending on how one categorizes the subject of architectures.
B.1.1 Memory and Learning

Anderson (2003) breaks cognitive architectures down into three sub-types (symbolic, emergent, and hybrid) to describe how Memory and Learning are usually implemented (Figure B.1).

![Taxonomy of Architectures by Memory and Learning](image)

Fig. B.1.: Taxonomy of Architectures by Memory and Learning (Anderson, 2003).

B.1.2 Traditional Representation

Duch et al. (2008) expanded on this organization to include graph-based memory, localist emergent, and symbolic-connectionist hybrids (Figure B.2). This perspective of grouping by traditionalist representations is arguably the most common in the literature. Thagard (2012) later describes the same architectures in terms of how procedures using comparison operators are either defined as connectionist via neural networks, or rule-based using if-then statements; this perspective does point out the role of the production system and rule matching in taxonomy, but remains true to
the traditionalist approach of how the cognitive cycle is implemented and computed under the hood per-se.

Fig. B.2.: Taxonomy of Architectures by Traditional Representation (Duch et al., 2008)

B.1.3 Modularity of Function

Sun (2004) organized architectures according to the modularity of function, akin to Fodor’s theoretical Modularity of Mind (Fodor and Pylyshyn, 1988). In it we find the same dichotomy of symbolic vs. emergent (connectionist) in single modules, with room for multi-modal hybrids being homogeneous or heterogeneous composites of the two, and described by the cohesion or coupling of these modules Sun’s approach takes on more of a structural perspective, with inter-module communication (or lack thereof) and region of interest as the primary focus.
B.1.4 Research Methodology

Gray et al. (2008) proposed one of, if not the most comprehensive taxonomies, and is distinctive by the comprehensive ordering by research methodology [Gray 2008]. In it, they divide architectures by use for developing cognitive science theory and basic research, vs cognitive engineering disciplines including control systems, control of components, and human-computer interaction. Gray et al.’s approach is truly all-encompassing, with an emphasis on the larger research paradigms that led to the creation and use of each major architecture.
Each of these approaches to sorting and describing cognitive architectures appeals to a different perspectives, ranging from the low-level implementation and representation, structural distribution and localization of information processing, to the overall research paradigms governing their use. In (Kotseruba et al., 2016) we find a comprehensive historical review of these developments and perspectives; in it, they point out disparity of perspectives (ranging from philosophy to neurobiological), and settle with the traditional perspective of implementation detail (symbolic, emergent, hybrid) – if only because the majority of available publications already adhere to this perspective.
C. COMMON MODEL OF COGNITION

C.1 The Standard Model of the Mind

The Common Model of Cognition aims to describe, and in some cases, prescribe the fundamental characteristics and requirements envisioned by Alan Newell (Newell 1990). In (Laird et al., 2017b), we find a comparison of the three primary Common Models with respect to these requirements, with statements about how each of the four main components operate.

Table C.1 (Laird et al., 2017b) is a color coded mapping of each of the three preliminary CMC architectures to the four functional areas and characteristics.

A. Structure and Processing

The Common Model defines the following characteristics for Structure and Processing:
1. The purpose of architectural processing is to support bounded rationality, not optimality.

2. Processing is based on a small number of task-independent modules
   a. Processing is parallel across modules
   b. Processing is parallel within modules

3. There is significant parallelism in architectural processing;

4. Behavior is driven by sequential action selection via a cognitive cycle that runs at 50 ms per cycle in human cognition.

5. Complex behavior arises from a sequence of independent cognitive cycles that operate in their local context, without a separate architectural module for global optimization (or planning).

   Processing across modules (2a) is asynchronous for both ACT-R and Soar, and as they rely on an external clock to advance to the next step in processing changes. Processing within modules (2b) is conceptually different for all three: ACT-R uses rule matching of slots against buffer contents. Soar has two phases for operator selection and operating firing, and thus processing is considered actioned during the operator firing phase.

B. Memory and Content

1. Declarative and procedural long-term memories contain symbol structures and associated quantitative metadata

2. Global communication is provided by a short-term working memory across all cognitive, perceptual, and motor modules
3. Global control is provided by procedural long-term memory
   a. Composed of rule-like conditions and actions
   b. Exerts control by altering contents of working memory

4. Factual knowledge is provided by declarative long-term memory

The use of declarative and procedural symbol structures (B1) is implemented differently: ACT-R uses chunks with activation levels, and procedural rules with utilities; Soar: uses triples (object, value, attribute links) with activations, and procedural rules with utilities; Sigma, being a functional programming language, uses logical predicates and conditionals typical of procedural programming languages. Factual knowledge (B4) is stored as a single declarative memory in ACT-R, while Soar makes the distinction between semantic and episodic memories. Sigma integrates the two and unifies declarative memory with procedural memory by using a combination of typed predicates while computing additive sums over factor graphs.

C. Learning

1. All forms of long-term memory content, whether symbol structures or quantitative metadata, are learnable.

2. Learning occurs online and incrementally, as a side effect of performance and is often based on an inversion of the flow of information from performance.

3. Procedural learning involves at least reinforcement learning and procedural composition.
   a. Reinforcement learning yields weights over action selection.
b Procedural composition yields behavioral automatization.

4. Declarative learning involves the acquisition of facts and tuning of their metadata.

5. More complex forms of learning involve combinations of the fixed set of simpler forms of learning.

Learning new procedural rules (C3b) is only supported by ACT-R (through rule composition) and Soar (by chunking). At the time of this writing, Sigma is still developing this feature.

**D. Perception and Motor**

1. Perception yields symbol structures with associated metadata in specific working memory buffers.

   a There can be many different such perception modules, each with input from a different modality and its own buffer.

   b Perceptual learning acquires new patterns and tunes existing ones.

   c An attentional bottleneck constrains the amount of information that becomes available in working memory.

   d Perception can be influenced by top-down information provided from working memory.

2. Motor control converts symbolic relational structures in its buffers into external actions.

   a As with perception, there can be multiple such motor modules.

   b Motor learning acquires new action patterns and tunes existing ones.
C.2 Suggested Changes for the Common Model

Hinrichs and Forbus (2017) proposes a dozen elements of a standard model of the mind and characterize their interrelationships, as if the field may one day compare possible future metrics within a system comparable to the Standard Model of Physics. The areas suggested for comparison are goal-seeking behavior, analogy, learning, memory, inference, representation, theory of mind, attention, reflection, causal and qualitative modeling, and spatial and temporal reasoning. Of interest their insistence the Common Model is especially suited to qualitative comparison of the relationships between quantitative states without explicitly naming them. One problem with this approach that they readily identify is how model evaluation based exclusively on qualitative results, by nature, introduces personal bias, and is subject to interpretation. The second issue is one found in the majority of the work attempting to map to the Common Model; the desire to define rational and meta-cognitive capabilities that manifest at much higher time scales than what is described in the Common Model.

Rosenbloom (2017) provides some additional insight into the discrepancies that surfaced while mapping Sigma onto the CMC. They propose to add extra features for self-monitoring, evaluation of how learning is affected by composition of memory sub-systems, and to establish a consensus that symbolism can represent the full cycle of information from perception, through cognition, to motor control. They go on to suggest some metrics that could be used to establish this consensus: primitive elements based on procedural or inferential role semantics; relations between the syntactic structures of elements; and organizing these results according to groupings of compound elements, ordering vectors, and metric aspects of regions of interest. The argument for measuring regions of interest, according to Rosenbloom et al., takes a more tangible approach to comparison, may be most applicable to
corresponding architectures such as Soar (upon which Sigma is based), and serves as an inspiration for unifying measurement across all CMC architectures; however, since many of these metrics are derived from Sigma’s functional implementation, they may not be transferrable to other architectures that use different data structures.

Rolfe and Haugh (2017a) propose a framework of six key capabilities, aptly named Integrated Cognition (INCOG), using dimensions and scales of challenge (easy to hard); taken together they present a ””DARPA hard problem””. The axes of capabilities are: Multi-level Mind, Unification, Learning Strategies, Reasoning Domains, Reasoning Mechanisms, and Knowledge Abstraction.

While they do go into a great level of detail describing the composition of each dimension to justify the difficulty scales, they are unclear how much support each architecture has outside of the boundaries of the Standard model. In Rolfe and Haugh (2017b) they provide examples of how established architectures (including ACT-R and Soar) self-report their capabilities according to the INCOG framework proposed in Rolfe and Haugh (2017a). Each architecture is compared against the capabilities of their own ”INCOG Strawman” architecture previously developed for DARPA’s Information Processing Technology Office (IPTO) in 2003. Rolfe and Haugh (2017b) draws the conclusion that implementations of the most well established architectures still require relatively low-level programming to add capabilities not explicitly designed into their architectures, and that the Common Model should be more explicit about requirements for dynamic spatial reasoning. The author concurs with this argument for specific flow of information between perception-motor and the rest of cognition, as this specification sits at the Cognitive level; however, this framework defines Learning and Reasoning strategies that exist
well above the Cognitive level, and belong in the Rational band, and hence not a part of the CMC proper.

Dancy and Ritter (2017) present two aspects that have not been seen in the CMC, namely a body, and a way to interact with the world, arguing that much about human cognition is evolved from and based upon embodiment in a physical world. They propose the requirement for physiology processing, with an example “HumMod” model and simulation system to keep physiological and cognitive systems synced in time, and contains functionality to affect sub-symbolic quantities of (actuator) measurements. They also suggest the CMC should have a (sub)module for perceptual-motor interaction that can read lists of words, dialogue items and colors, icons, move the mouse and click, and to type in accordance with existing GOMS/HCI environments. Dancy and Ritter (2017) make an additional note that affective sub-modules still need to be developed, and most architectures within CMC (including their own) tend to take shortcuts when representing the environment through mental models. The author can confirm these shortcuts exist in both ACT-R and Soar model files; they are usually an artifact of task-specific imperatives that cannot be further abstracted without some difficulty. The list of suggestions, while noteworthy, mostly reside at the Biological layers, and are an example of how the CMC architecture can inform and describe the sub-symbolic qualities that perceptual and motor modules have, but not necessarily how they should be implemented at those levels.

Taatgen (2017) argue that unlike the standard model in physics, the standard model of cognition does not specify the basic units (of knowledge) that can be broken down even further for explanations, and propose that the CMC needs to add multiple levels of abstraction (complimentary to Newell’s Bands) to encourage research efforts to establish learning mechanisms that connect each level. They
suggest these mechanisms can be grouped according to clusters of neurons (or equivalent), primitive operations, operators or productions, goals, tasks, and higher levels that provide “top-down guidance”. They describe a network of low-level operations that feed “up” the hierarchy while modeling the Rapid Instructed Task Learning (RITL) task. This work appears to be very similar to the Socio-Technical GOMS approach used in (Peters et al., 2018b) to model Unit Tasks for intelligent agents in StarCraft. My present work takes a step in this direction through formal specification.

Stewart and Eliasmith (2017) are proposing the CMC should also take into account the characteristics of dynamical systems theory, as demonstrated in their earlier work on Neural Engineering Framework (Eliasmith, 2003), and the Neural Engineering Objects (Nengo) platform (Stewart and Eliasmith, 2014). Dynamical systems theory takes into account continuous state representations, continuous timing, continuous actions, continuous learning, and parallel control loops. They argue the CMC can measure neural evidence in Basal Ganglia to support the proposed modifications, and how some large-scale integrated continuous models already modelled in Nengo can still satisfy the CMC if augmented with these requirements. There are two potential issues with this suggestion: first, that (Stewart and Eliasmith, 2017), do not yet have solutions that comprehensively deal with these challenges, which would make conformity to the CMC difficult to confirm. Second, these suggestions conform to an implementation at a simulated Biological level, and therefore should be outside the scope of the CMC.

The CMC is somewhat abstract and subject to interpretation by different researchers. Stocco (2017) proposed a reinterpretation of selective attention as a form of control of working memory. They suggest a new form of attention could be based on self-generated reward signals and model-free reinforcement learning. To
accomplish these changes, they suggest a framework that accounts for spreading activation from working memory, the procedural control of working memory augmented with reinforcement learning, and continuous performance monitoring much like meta-cognition. To support this suggested modification, Stocco et al. outline two tasks that the community could use to quantitatively evaluate the utility of these changes: the first is a Probabilistic Stimulus Selection (PSS) task using comparison of choice vs avoidance of non-verbal stimulus and accompanied by a binary reward; the second is the Simon task, using an allocation of attention to different stimulus features, and measures of response to reflect the retrieval of previously instructed rule from Long Term Memory. They do admit, however, that current architectures that include a model for the Basal Ganglia (BG) still disagree on certain details, such as the distinction between BG pathway in the Sub-Thalamic nucleus and Stratium, and the overall role of the BG in action selection vs general gatekeeping.

In (Peebles and Cheng, 2017) we see a broader argument that the Standard Model of the Mind (SMM) must incorporate multiple representational formats, and meta-cognitive processes that operate on them (a similar argument to that made by Taatgen, 2017), but more specific to modeling visual representations of mental imagery. To support multiple levels of abstraction, Peebles et al. suggest a meta-cognitive model could be measured using an affine model of problem solving on the Raven’s Progressive Matrices test (Kunda et al., 2013). Despite this suggestion, Peebles et al. are not clear whether mental imagery must be supported by some form of bitmap representation of the topological structure of objects, a convolution function for edges, lines and shapes, or a spatial coordinates system that is extracted from the perceptual system for use somewhere else. This lack of
clarity may make some quantitative measurements difficult, but does address the perceptual system shortcuts pointed out in (Dancy and Ritter, 2017).

Kieras (2017) presents a plausible argument for the Common Model to modify characteristics of Working Memory. The arguments in question make reference to how the Executive Process - Interactive Control (EPIC) architecture uses functional relations between cognition, perception, and action, and that humans do not have a cognitive bottleneck during response-selection; a claim discussed in significant detail by Meyer and Kieras in (Meyer and Kieras, 1997a) and (Meyer and Kieras, 1997b). The problem yet to be addressed involves assumptions based on low-level processing evidence that contradicts some key constraints in the proposed standard model (at higher levels). While the argument and identified problems are well founded, it remains as a proposal only.

Emotional learning for the Standard Model was proposed by West and Young (2017), including a description of how emotional factors impact decision-making, and an outline of how to augment the standard model to include emotional factors. They suggest the addition of somatic markers representing desirability of objects and people, and an Emotional Module that feeds back into the initial goal (focus) based on a value parameter. Of note is the suggestion this modification can use emotional module/buffer reaction to conditions as a form of shadow production system; the new system would assess if agents change, or fail to change values when exposed to different information like somatic markers.

Lieto et al. (2018) later suggested the addition of a hierarchically ordered set of social interactions to identify which parts of the CMC are necessary for each level of the hierarchy, and ultimately support the emergence of social cognition. They suggest this could be supported through the use of a separate representation at a higher level of abstraction that can still reduced to the level of the Common Model. A higher order
representation (like the shadow system suggested in (West and Young 2017)) could tie emotional and social markers in at various levels of a hierarchy. The proposal by Lieto et al. adds a novelty to the CMC, but does not suggest any metrics to confirm the implementation, acknowledging that possible interplay between contrasting views is not yet entirely clear.

A more concrete proposal from Dudzik et al. (2018) outlines the establishment of a test platform through the development of three components (in addition to front-loaded expertise): an agent based on cognitive model; an environment to run the agent; and a test-bed to host the environment. While simple in form, Dudzik et al. identify the necessary components that satisfy the need to compare different cognitive models. Dudzik et al., suggest the use of Alan Newell’s Single Complex Task Analysis described in (Newell 1973) to measure backwards compatibility in the performance of open-ended tasks. This work, while identifying a straight forward methodology of development, lacks detail on implementation and representation for the values used to prioritize between different tasks at different times, and suggests the use of meta-goals to overcome this limitation.

Meta-cognition is also discussed by Kralik et al. (2018), suggesting the CMC needs to back up to identify and elaborate on common language that should be used to describe metacognition. They go on to suggest that only once this has been accomplished, should the community add the necessary components that, according to Kralik et al., already have empirical evidence of existence. The components for measuring metacognition include: monitoring; evaluation; planning; mental simulation, control (arbitrating, multitasking, task switching, modulate or bias); self-reflection; and self-improvement. They point to similar work in (Gamez 2018) that agrees a grounded, scientific, measurable theory can overcome the limitations
of empirical philosophy, and lend credence for the practical application of meta-cognition within the CMC.

Arora et al. (2018) suggest agents based on CMC should construct objects that are reflected in a memory architecture, with concepts represented in a unified high-dimensional geometrical space. They present an implementation of this using Vector Symbolic Architecture called “Kantian Holographic Declarative Memory” (K-HDM), and evaluate this representation by defining memory chunks as slots, values, environment vectors, and memory vectors. K-HDM uses non-commutative circular convolutions to create conjunctive relations between concepts, and vector addition to add these relations to their meanings vectors. This work is somewhat similar to the multidimensional unit vector representation of concepts in (Eliasmith, 2003) and (Eliasmith and Anderson, 2004), albeit more specific for the representation of hierarchical abstractions.

A very promising study was produced by Steine-hanson et al. (2018) using large datasets based on fMRI from the Human Connectome Project to refine the CMC and test it as a reasonable model for human cognition. Steine-Hanson et al. compared this data against variations of the CMC using direct and mixed relationships between regions of interest, a hierarchy model of cognition, and a star procedural model. Their preliminary findings indicate a Mixed (Common) Model presents the most plausible structure, and indicates additional pathways should be added to the CMC in support of their findings. This work was continued by Stocco et al. (2021) using Dynamic Causal Modeling (DCM) of the HCP data sets against alternative cognitive architecture organizations, demonstrating how the current CMC is generally superior to alternative modular arrangements found in other abstract models, indicating it is still a viable blueprint to represent the functional arrangement of the human brain.
D. ACT-R COGNITIVE ARCHITECTURE

The Adaptive Control of Thought (ACT) is a theory of cognition created by John Anderson (Anderson and Schooler, 1991) to describe the human mind, and designed to satisfy some of the original criteria set by Newell (Newell, 1973). ACT was later updated to include a rational component, not because of how it models rationality, but because of the influence of rational analysis at the time (Anderson, 1993).

D.1 Evolution of ACT-R

ACT-R in its current form is best described in Anderson’s book on “How can the Human Mind Occur in the Physical Universe” (Anderson, 2007), aptly named after Newell’s last lecture on the same topic, in which ACT-R is described as a logical representation of the various functions of the human brain. Some of the main themes of ACT-R covered in (Anderson, 2007) include the modularity of the mind, the structure and function of Human Associative Memory (HAM) theory (Anderson, 1983), and how the Adaptive Control of Thought can be represented as a production system consisting of conditions and actions, much akin to IF and THEN statements in declarative programming languages.

ACT-R is the latest version in a series of versions (ACT, ACT*), and somewhat encapsulating the Anderson’s understanding of the ever-evolving field of cognitive psychology. The original ACT, largely based on HAM theory, was later augmented in ACT* based on the optimality of human cognition, under the assumption that humans have evolved to produce the most optimal decision making process in the environments that we are situated (Anderson, 1983). The idea at the time was if we
could model this decision making process based on human constraints imposed by our biological brain and structure, it would reduce the search space and produce an optimal solution faster than alternative designs.

D.2 Layout of ACT-R

The ACT-R architecture uses the concept of *modules* to represent discrete functional areas of the brain. Modules in ACT-R cannot be accessed directly; instead, each module has a designated *buffer* that the central (production) system uses to retrieve, modify, and store small bits of information much like memory registers in standard computer architectures. The most commonly used modules in ACT-R are the intentional, declarative, procedural, visual, and motor modules (see Figure D.1 adapted from [Laird et al., 2017b]).

The *intentional* module is used to store the current goal of the agent, is representative of our anterior Pre-Frontal Cortex (aPFC), and is interfaced through the intentional buffer. The *declarative* module stores chunks of declarative memory, is modeled after the our brain’s Temporal and Hippocampus areas, and is interfaced through the retrieval buffer. The *visual* module stores observations received from an external visual system, is representative of our Occipital lobe, and is interfaced through the visual buffer (our Parietal Cortex). The *manual* module stores actions sent to an external motor system, is modeled after our brain’s motor area and cerebellum, and is interfaced through the manual buffer (our Motor Cortex).

D.3 Knowledge in ACT-R

Makes the distinction between two types of knowledge: declarative, and procedural. *Declarative* knowledge (the *what*) in ACT-R theory can be described as discrete chunks of information that represent the model’s knowledge of various facts
about the environment, as well as knowledge of the current goal, and possible actions that can be produced. *Procedural* knowledge (the *how*) in ACT-R theory is the combination of conditions and actions that change states. Procedural combinations are termed *production rules*, and are the mechanism by which an ACT-R model produces change in states, much like the if/then rules in most programming languages.

The Chunk Activation equation is composed of the base-level activation value, the spreading activation value, the partial matching value, and a noise value. Each of these values can be specified.

Spreading activation can be altered by the :mas parameter.
Strength of association between two chunks can be explicitly set using \textit{add-sji} or \textit{sji-hook} parameters.

Partial matching is set through the \textit{:mp} parameter. Maximum similarity parameter is set by \textit{:ms} Maximum difference of similarity is set by the \textit{:md}

\textit{:blc} is base level constant \textit{:bll} is base-level learning, and sets the value of the decay parameter \textit{d}

\textit{:le} is latency exponent \textit{f} in equation retrieval times. \textit{:lf} is latency factor \textit{F} in equation retrieval times.

The noise value of chunk activation is, by default, based only on a chunk-specific transient noise, and can be set using the \textit{:ans} parameter. Permanent noise that is always applied is generally not included by default (and is not a part of the original theory), however it can be turned on using \textit{:sdp} and the value set with \textit{:pas}.

Declarative finsts (fingers of instantiation) is a setting that determines if, and how many of the most recently retrieved chunks should take priority in retrieval requests. The modeler can set these parameters using \textit{:declarative-finst-span} and \textit{:declarative-num-finsts}, respectively.

\section*{D.4 Decision Making in ACT-R}

The heart of the ACT-R decision making cycle is the \textit{production system}. The production system is represented by a \textit{procedural} module, and is modeled after our brain’s Basal Ganglia. The production system cycles through a process of pattern matching (based on our Stratum), pattern selection (based on our Pallidum), and pattern execution (based on our Thalamus). The ACT-R decision making process also relies on the concept of \textit{production rules} to choose the next course of action. The decision making process assumes the agent already has all of the knowledge that it
needs to function in the target domain. This knowledge needs to be input or loaded by the programmer into a model file in the form of chunks of memory, or obtained and encoded from some external stimulus (Anderson, 2002).

![ACT-R production system with buffers](image)

The chunks of information specified in the left hand side (LHS) of a production rule is compared to those in each specified buffer. Given the possibility of multiple production rules matching, the production system will compute and select the rule with the highest estimated utility value. ACT-R utility function is designed to maximize the reward of prospectus knowledge retrieval over time.

Every production rule in ACT-R has an associated utility value:

\[ V = pG - C \]

such that \( G \) is the current goal, \( p \) is the probability of that production rule helping to reach goal \( G \), and \( C \) is the cost of retrieving that production rule from memory. The action of retrieval, just like everything happening in our own brain, has an associated
energy cost, thus the utility of fetching the rule depends on the difference between the expected value and cost of retrieval.

Each chunk has an activation level defined as: \( A_i = B_i + \epsilon \), where \( B_i \) is the base-level activation (recency and frequency of chunk \( i \), described later), and \( \epsilon \) is a noise value.

When a retrieval request is made to declarative memory, each chunk has a probability of being successfully recalled if it exceeds a retrieval threshold \( \tau \). This probability of recall is computed as:

\[
p_i = \frac{1}{1 + e^{-\frac{\tau - A_i}{s}}}
\]

where \( s \) is the noise factor computed at run time. Chunks activation also depends on the time it takes to retrieve it, a concept we intuitively understand as having to "dig deep" when asked for a fact that has not been recalled for some time. ACT-R calculates this latency as \( time = F e^{-A} \) where \( A \) is the activation of the chunk (as described earlier), and \( F \) is the latency factor.

Productions, on the other hand, are simply selected based on the cumulative utility of the chunks matching the specified criteria. This process is not explicitly coded into ACT-R; however, if there are some productions competing with an arbitrary utility value \( U_j \), then the probability of each competing production \( U_i \) can be shown as:

\[
p(i) = \frac{\frac{U_i}{e^{\sqrt{2s}}}}{\sum_j \frac{U_j}{e^{\sqrt{2s}}}}
\]

where the summation of \( j \) is the sum of all productions that have their LHS conditions satisfied. ACT-R also contains a *temporal look ahead* feature that computes the cost of a future production (possibly) having a higher expected value, which is defined as:
\[ \int_{V_i}^{G} (x - V_i) Z_t(X; V_i) \, dx \leq \tau \]

where \( V_i \) is the value of the production rule \( i \) at time \( i \), \( x \) is some constant value (undefined) \( \tau \) is the expected gain from waiting, and \( Z_t(X; V_i) \) is the probability distribution that a production rule with value \( x \) will ever match at some time in the future. Newell defines \( Z_t \) as:

\[
Z_t(x; V_i) = \frac{1}{t(G - V_i)} e^{-\frac{1}{t(G - V_i)}}
\]

where the PDF of this latency function is derived from (Anderson, 2002). Whitehill performed an analysis of this integration and describes how it could (mathematically) be compared to a form of stochastic optimal control theory (Whitehill, 2013).

The likelihood of needing a memory in the future is described as the “Odds likelihood” ratio (Anderson, 2007) defined as: \( A t^{-d} \), where \( A \) is a scaling constant, and \( d \) is the decay rate (see (Anderson and Schooler, 1991) on how these were derived from human data).

### D.5 Learning in ACT-R

There are two types of learning in ACT-R; one is applied for individual knowledge chunks stored in declarative memory, and the other is applied to the production rules that bind them together.

Chunks learn a base level learning rate through a process of frequent retrievals, that in turn increasing the chance of activation, and can be defined as:

\[
B_i = \ln \left( \sum_{j=1}^{n} t_j^{-d} \right)
\]
where $n$ is the number of previous presentation for chunk $i$, $t_j$ is the time since the $j$th presentation, and $d$ is the decay parameter (defaults to 0.5 in the latest version of ACT-R), as described in (Anderson, 2007). This value is assigned and updated for each chunk as it is first created (and entered into declarative memory), and for each retrieval thereafter.

Production rule learning in ACT-R is based on a simple integrator model (Bush and Mosteller, 1955). For example, if application $n-1$ of production rule $i$ yielded a reward of $R(n-1)$, and a current reward of $R(n)$, then the utility of production rule $i$ for the current application can be shown as:

$$U_i(n) = U_i(n-1) + \alpha[R_i(n) - U_i(n-1)]$$

where $\alpha$ is the learning rate, based on the classical Rescorla-Wagner learning rule (Rescorla and Wagner, 1972).
E. SOAR COGNITIVE ARCHITECTURE

The main inspiration for the Soar cognitive architecture can be traced through the early works of Newell (1973). The modern version of Soar is based on the organizing principle of problem spaces conceived by Newell (1980), and further refined to deal with impasses and sub-goal creation by Laird et al. (1986). Soar later evolved into Newell’s candidate architecture to satisfy the requirements outlined in Unified Theories of Cognition (hereafter UTC), (Newell 1990).

E.1 Evolution of Soar

Earlier versions of Soar were based on classical representations of production rules, objects and attributes, generating goals, and learning through information chunking (Laird et al. 1987). Later versions added new representations capable of handling various complex tasks. Example changes include: making the distinction between types of long term memory, being semantic memory, episodic memory, and preference memory, as well as adding new learning mechanisms such as semantic learning, episodic learning, and reinforcement learning (Laird, 2012). Later versions of Soar also make use of a Goal Dependency Set (GDS), that allows a modeler to customize the “psychology” of Soar, all depending if the main intent is to model humans or to develop agents for large-scale knowledge systems (Wray and Laird, 2003). This approach to managing large databases of native LTM was also an attempt at overcoming some of the limitations demonstrated in the ACT-R architecture at that time (Douglass et al., 2009).
Fig. E.1.: Soar Cognitive Architecture (Laird et al., 2017b).

E.2 Layout of Soar

The Soar Cognitive Architecture (Figure E.1) is divided into a few main components that are used to support the central concept of behaviour as movement through a problem space (Nilsson, 1971). Soar accomplishes this by representing the current goal state and possible operators as a graph of objects with interconnected relationships and values; all knowledge in Soar is therefore mapped as either goals, problem space, states or operators. The current state (frequently referenced as s0 in Soar literature) represents the current goal context, made up of an instance of each type of object residing in Working Memory. Soar also uses the concept of domain knowledge mapped to the problem space, with the intent of reducing the total state space required to search for applicable operators.
E.3 Knowledge in Soar

Knowledge in Soar is divided into two main types of memory; Long-Term Memory (LTM) that holds the knowledge independent of the goal context, and Working Memory (WM) that contains an instance of memories that represent the current situation. The Working Memory construct is made up of discrete Working Memory Elements (WME), and holds one of each of the four components (or slots), being the goal, problem space, state, and operator. This knowledge is retrieved from, acted upon, and stored in Long-Term Memory through the use of the decision cycle (akin to a production system). The relationship between these slots is called a chunk, and represents the association between slots given the current context.

Long-Term Memory in Soar is used to store all of the procedural, semantic, and episodic associations that are used to move from one context to another. These associations, stored as chunks, are also the mechanism by which Soar learns how to move to a previously unknown goal state after an impasse occurs.

Procedural memory in Soar stores production rules as a set of conditions and actions. Conditions are those identifiers that are compared to a given context (the IF statement), and actions are those changes to be performed if the rule is matched and fires (the THEN statement). Soar also makes a distinction between different types of productions, depending on their role, being: operator proposal, operator comparison, operator application, or state elaboration.

E.4 Decision Making in Soar

The decision making (or execution) cycle in Soar is effected through five main phases, being input, proposal, decision, application, and output. The input phase is used to read in sensory data that will reside in working memory. The proposal phase involves pre-emptively firing all matching production rules (without affecting any
outcome) to explore the state space and identify the ideal operator for the current situation; this sub-cycle of matching and firing happens in parallel and continues until either there are no more rules to consider, or an impasse occurs (explained in additional detail below). The decision phase either selects the operator identified during the proposal phase, or in the case of an impasse, will generate a new state that can be used to learn a new behavior if encountering the exact same context again. The application phase effects those changes specified in RHS of the production rule, to either create or modify a WME.

E.5 Learning in Soar

Learning new productions in Soar is performed through a chunking mechanism, for the purpose of dealing with an impasse situation. Soar accomplishes this by relying on the relationship between super- and sub-states, and by linking contextual knowledge in which all available options have been exhausted. Soar deals with an impasse by creating a new sub-states.

Soar uses Q-values to perform Reinforcement Learning (RL), and to assign a utility value to operators that were a part of the selected production. Q-learning has been shown to be a close approximation to human learning [cite Walsh and Anderson 2010]. RL rules are specified by the modeler in the same syntax as the LHS of productions, and can be as simple as one rule per state-operator pair (akin to a table-based algorithm), and interpreted as binary features in a linear approximation:

\[ Q(s, a) = w_1\phi_1(s, a) + w_2\phi_2(s, a) + \ldots + w_n\phi_n(s, a) \]

where the RL rules are numbered 1..n, Q(s,a) is the state-operator (or state-action) pair, \( w_i \) is the preference value for the RL rule \( i \), and \( \phi_i(s, a) \) is an exclusive binary filter and equals 1 if rule \( i \) matches (s,a), otherwise 0 [Laird 2012].
### Explanation-Based Chunking Algorithm

<table>
<thead>
<tr>
<th>Relevant OSK Tracking</th>
<th>Identity Analysis</th>
<th>Rule Formation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator Decision</td>
<td>Identity Assignment and Propagation</td>
<td>Identity Graph Manipulation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>During Problem-Solving</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Constraint Tracking</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operationality Analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Backtracing</td>
<td>Rule Formation</td>
</tr>
</tbody>
</table>

Fig. E.2.: Explanation-Based Chunking in the Soar Cognitive Architecture (Laird 2012)
F. CMC META-MODEL DIAGRAM

Fig. F.1.: EMF class diagram for CMC Metamodel.
G. META-MODELING TOOLCHAIN

The OMG specification for MOF is not a tool in and of itself, it is only a standard. The utility of this approach is found in commercial and open source projects that commit to its implementation. This section describes various tools that were used in the production and testing of meta-model designs, templates, graphs, and other related artifacts.

G.1 Eclipse Modeling Framework

The Eclipse Foundation is an example open source Integrated Development Environment (IDE) with a community devoted to implementing and maintaining packages that are compliant with Essential MOF (EMOF), a subset of the MOF standard considered applicable to the vast majority of modeling tasks. This community effort and product is collectively termed the Eclipse Modeling Framework, and consists of a suite of tools and Eclipse plugins that, when used collectively, form an end-to-end tool chain to define, model, and generate code for any user-specified domain (Figure G.1).

The three primary tools that form this tool chain are: Eclipse Model Framework to create the domain meta-models; Sirius, a plugin used to design Graphical User Interfaces for end users; and Acceleo, a code transformation and generation plugin compatible with many existing high-level languages.
G.2 Graphical Modeling Framework

Modeling can be performed by manipulating the data elements and relationships using different viewpoint representations, such as list, tree, or graph. This approach works well for smaller models, but quickly becomes confusing, complex, and difficult to maintain.

Sirius is an Eclipse plugin designed to provide a Viewpoint specification for working with a precise view of a pre-defined EMF meta-model. Sirius allows a domain expert to define the views, visual elements, and tool palette that will be available for end users that are working with a domain model. An ideal use case built on a Sirius graphical editor facilitates the custom interface and objects for cognitive modeling in the CMC domain. The current approach is based on the Sirius graphical modeling framework to generate viewpoints that adhere to the CMC meta-model, and to specify the relationships between entities that are built on it.

The Common Model User Interface can use a graphical representations of CMC objects, their relationships, and user-defined customization to aid in architecture-agnostic model design that can still adhere to the CMC specification (Figure G.2). The graphical modeling interface also provides the modeler a choice of environments
Fig. G.2.: Sirius graphical modeling example.

in which to run the agent, selection of tests to perform, and option of environment-specification customization, where applicable.
REFERENCES


