HÁSKÓLINN Í REYKJAVÍK
REYKJAVÍK UNIVERSITY

MSc in Computer Science

A Foundation for Autonomous Conceptual Engineering Design

Chloe Schaff

January, 2024

TÖLVUNARFRÆÐIDEILD
DEPARTMENT OF COMPUTER SCIENCE

A Foundation for Autonomous Conceptual Engineering Design

Chloe Schaff

Thesis of 60 ECTS credits submitted to the Department of Computer Science at Reykjavík University in partial fulfillment of the requirements for the degree of Master of Science

January 29, 2024

To the inventor in all of us

Contents

List of Figures

List of Abbreviations

AI	Artificial Intelligence
AGI	Artificial General Intelligence
ACED	Autonomous Conceptual Engineering Design
CPM	Critical Path Model
CRM	Causal-Relational Model
EDA	Engineering Design Agent
EDL	Engineering Description Language
HLD	High-Level Designer
IRD	Initial Requirement Description
KBE	Knowledge-Based Engineering
LLO	Low-Level Optimizer
LoD	Level of Detail
O&O	Operations and Objects
OR	Output Renderer
OTCE	Optimizer-Tester Combined Environment
RDIS	Restricted Domain Intelligent System

A Foundation for Autonomous Conceptual Engineering Design

Chloe Schaff

January 25, 2024

Abstract

The engineering method is one of the most powerful problem-solving tools humanity has ever developed. It allows a designer to begin with a high-level conceptual understanding of a problem and then develop solutions at decreasingly low levels of detail until they are left with actionable sets of schematics and instructions. This, however, has always been the domain of human engineers as the engineering method is designed for — and assumes the presence of — human-level intelligence. Indeed, even recent advances in automation have only been able to recreate part of the conceptual design stage of the engineering process. The current state of the art involves knowledge-based engineering systems that can improve a human engineer's efficiency and generative design systems that can completely solve low-level part design problems; both of which require close human supervision. Neither of these kinds of systems, however, have the high-level design reasoning or agency of a human engineer and so they are unable to approach unfamiliar or more complex design problems. Additionally, trusting their designs requires extra verification and testing on the human engineers' part. The objective of this work was to take a step towards developing these high-level design capabilities by introducing methods and systems from the field of artificial general intelligence and implementing them in a small technology demonstrator. The resulting prototype, *EDA-0*, is capable of reasoning about basic conceptual design problems, identifying viable solutions, and outputting a human-readable trace of its decisions. It can also be further modified to include cooperation with a human engineer. Though limited in its current capabilities, this prototype lays important groundwork in autonomous problem decomposition, state representation, and human-computer cooperation. The work performed here is intended to form the foundation for a new generation of artificially intelligent engineers and designers.

Acknowledgments

I would like to thank Kristinn R. Thórisson and the AERA team for their assistance and feedback during the course of this project, I would not have made it this far without their help.

I would also like to thank my family who kindly supported me while I fled to Iceland to study artificial intelligence.

Chapter 1

Introduction

Engineering design is the process by which humans develop new solutions to problems. The solutions need not be perfect or even fully understood, what matters is that they provide a workable solution in an acceptable amount of time using available resources. Doing this requires a high level of problem-solving capability and the ability to reason about problems, environments, prior knowledge, and any other relevant factors. Though engineering is often associated with dense math, complex equations, and mind-numbing spreadsheets, the process of design is much more intuitive and abstract. In other words:

> "[W]e assume the secret of engineering lies in the mastery of arcane realms of knowledge — sophisticated calculus and powerful computing science implemented by a dispassionate, almost mechanical person — but the power of engineers lies in their *method*, a method used long before sophisticated mathematics and computers" - Hammack (2023)

All of this makes engineering design a difficult process to automate because so much of it is conceptual. That is to say that much of the process requires representing knowledge at a highly abstract and causal level for easy manipulation. For example, imagine a bicycle. Before you even start the design process, your mental picture most likely consists of a vehicle with two wheels, handlebars, and a triangular frame holding it all together. You probably also understand that pushing down on the pedals causes the gears to turn, which then causes the rear wheel to turn. At no point do you possess knowledge of this bike's exact geometry and you certainly could not transform this picture directly into instructions for building this bike in real life; you simply have a rough idea of what it looks like and a simple causal model of how it works. If desired, you could even choose to forget about the bike design and decide to design something entirely different. Human minds excel at this kind of abstract and general reasoning, but this still presents significant difficulty to computers.

The current state of the art in automated design comes in two forms: *generative design* and *knowledge-based engineering* (KBE). Generative design is the use of numerical optimization systems that use a model of the environment and a set of user-provided constraints to generate an optimal design for a specific part (Emdanat et al., 1999). These systems often take the form of genetic algorithms or other generative machine learning programs. For example, a generative designer could be instructed to design a mountain bike capable of handling the significant forces involved in the sport while also keeping the frame as light as possible and making sure there is still room for the wheels, seat, and so on. A similar exercise can be performed with value-driven design architectures, among others (Vidner et al., 2021; Castagne et al., 2009). This kind of design excels at finding beautiful optimized structures but becomes computationally challenging when the problem space becomes bigger; a bicycle is one thing but an airliner is very much another.

As an answer to this problem, KBE offers a way to perform low-level tasks more efficiently by using models of engineering knowledge encoded into databases. Systems like these can capture knowledge efficiently, identify what is relevant to the problem, and transfer knowledge across problems (La Rocca, 2012). This means that human engineers can use a KBE system to handle low-level design work and design revisions while they focus on the more stimulating creative work. In the bike example, one could design a KBE system that encodes an abstract model of a bike and can be provided with design specifications to produce a detailed design of a custom bike. The challenge with these systems is that they often cannot acquire this knowledge on their own and are instead entirely reliant on human engineers either to encode their knowledge explicitly or as sources of training data for their machine learning models (Kügler et al., 2023).

Another shortcoming worth mentioning is that not all automated engineering software can be completely trusted. In the case of generative designers, their numerical operation leaves them with no way to justify their decisions or explain their reasoning — their designs are simply guided by patterns and fitness functions. This is also true for pre-trained systems such as artificial neural networks, where there is inadequate semantic mapping between the network's internal representation and the corresponding knowledge (Thórisson et al., 2016b). This makes it difficult to ascertain the limitations of a system's knowledge, outside of which it may fail suddenly and in unexpected ways (Bieger and Thórisson (2018), cf. Arkoudas (2023)). This can be made to work in low-level problem-solving, where parts can simply be tested to validate the system's solutions, but this cannot be scaled up to larger problems.

KBE systems are a start in this direction as they are able to produce reasoning traces as a way of explaining their decisions. In this way, there is at least a basic understanding of the causal relations of the different elements of the design. This is an improvement in reliability but only within the predefined areas of knowledge that these systems are programmed to work within; a KBE system for aerospace design will probably not work for mechanical engineering, for instance. For this, we need a system capable of understanding a problem, recognizing its own shortcomings, and taking the initiative to work through or around gaps in its own knowledge. A system like this would possess a much higher level of problem-solving ability and could even be considered a design partner rather than a design tool.

At this point, it is worth considering what it means for a machine to understand a problem. According to Thórisson et al. (2016b), the process of understanding concerns an agent's[1] ability to form explicit composite models about its own experience. An agent's *level of understanding* can be gauged by how well it can apply its models of phenomena to predict patterns, achieve goals, generate explanations, and recreate the phenomenon from scratch. The more of these an agent is able to do with its models and the more complete and accurate the models are (with respect to the phenomenon's behavior), the better the agent's understanding of the phenomenon. Based on this approach, we can compare three different agents' understanding of the bike problem: a generative designer, a KBE system, and a human engineer. In this case, the physical phenomena we consider will be the forces the bike must resist, the materials it is made out of, the behavior of the controls, and so on. A generative designer can predict what shapes will yield the best performance and it can use physical phenomena to produce an optimized bike design. This, however, requires that the problem be scaffolded by a human, after which the system must be tested and fine-tuned until it produces the desired results; the outcome is a design system that can produce results but is less than transparent. A KBE system can improve on this by also being able to produce an explanation of physical phenomena and its reasoning for the design with regard to the problem statement; after all, this is what is encoded in its knowledge base. KBE systems also require some assistance from humans as they are often unable to learn on their own; their ability to explain and reason is only as good as the knowledge base they are provided with. Finally, a human engineer can outperform both kinds of systems with their ability to recreate the physical phenomena in question; they could even run experiments to develop new models of bike design or invent a bike from scratch given enough knowledge of the phenomena. In short, a human engineer's grasp of the problem gives them the ability to handle a high degree of novelty. To this author's knowledge, no artificial agent has yet been developed that is capable of demonstrating all of these abilities to the level of a human engineer. In order to achieve this, it is assumed that a new system must be developed to add an additional level of understanding on top of the existing technology.

This thesis proposes a high-level conceptual designer as a solution to the problems of understanding and reliability. Such a system is intended to be capable of manipulating designs at the "back of a napkin" level of detail; the designer will be

[1]For the purposes of this thesis, an 'agent' is defined as an intelligent system (biological or artificial) with the agency to make its own decisions and solve its own problems, usually — but not necessarily — in service of a goal or goal hierarchy.

able to solve problems with a high or variable level of abstraction and use reasoning to find and justify its solutions. This new approach is not intended to replace existing systems entirely; instead it will operate at this higher level of abstraction and coordinate the operation of lower-level generative designers. Additionally, as in KBE, such a system would represent the problem and solution with conceptual models of physical laws, engineering skills, and available parts. It would use an understanding of causality to reason both forward from the problem and backward from the goal to generate a solution in much the same way a human engineer would. It could also learn and form new models by performing experiments and acquiring experience. Because it works at such a high level, such a system would be able to avoid the combinatorial explosion involved in searching the massive solution space of complex problems and would be able to explain the reasoning for its solutions. A high-level designer would solve the problem at a conceptual level and then employ a generative designer as a low-level optimizer to test and complete the design. The abstract system would autonomously[2] decide what the parts should do and the generative system would decide what they look like. Such a system would be a new kind of artificially intelligent agent, an engineering design agent (EDA).

It is worth noting that conceptual design as described here is not based on the free association style of creativity that humans are capable of. While it is certainly possible for a solution to "pop" into one's head, that is not the kind of creativity and problem-solving explored here. Instead, the EDAs in question would approach the problem directly and develop solutions using their experience and subject matter knowledge. Creativity to an EDA is more a process of finding the model that best fits the situation and then analogizing or reinventing when that process fails. The kind of indirect creative approach that humans can take is left for future explorations into conceptual design.

If functional, such an engineering design agent (EDA) would have a similar skill level to that of a human engineer and so could assist one in much the same way that a project partner would. If desired, it could have the agency to operate on its own at its user's direction or it could be employed to check designs, solve particular elements of the problem, or engage in a collaborative design process. This level of autonomy versus collaboration could even be configurable ranging from a fully supportive to a fully independent role as per Altavilla and Blanco's five levels of design automation (2020). Its ability to explain its designs and recognize deficiencies in its understanding would also make it a trustworthy participant in any engineering workflow. While this is certainly useful in its own right, the development process leading up to an EDA would also produce a significant amount of knowledge. Such an investigation requires probing the nature of design tasks, discovering new methods of problem decomposition, determining the effect that representation has on a task, and working

[2]In this context, the term 'autonomous' is used to mean that the agent can make decisions on its own without the need for human control or intervention. It works as an independent agent, not as a tool.

out methods by which AI agents can learn from a human's feedback and their own experience. All of this could lead to advancements in task theory and design theory with the further possibility of aiding in the development of a generally intelligent agent. With all this in mind, the development of an EDA certainly appears to be an interesting vector for research.

1.1 Research Questions

To answer some of the important questions related to autonomous design, this thesis defines three key categories within which key questions have been selected to be addressed. The five research questions (RQs) below are answered in the following chapters of the thesis. They cover the three areas of *problem analysis and understanding*, *knowledge and state representation*, and *autonomy and cooperation*.

1.1.1 Problem Analysis and Understanding

Problem-solving in design often does not occur in the real world, it occurs on paper or in a designer's imagination. An EDA must be able to understand its work in such a way that, even though it works on its problems in an imagined environment, it is still able to implement functional solutions in the much more constrained real world. Additionally, a key part of design is breaking down the original problem and understanding it within the context of one's prior knowledge. It follows that any design agent should be able to take a problem and transform it into a set of related sub-problems that must be solved individually in order to build up to the original problem statement. Such an approach could be used to analyze a task and produce an objective measure of difficulty; it could also be used to compare difficulty between tasks. This would be quite a useful skill for an EDA to have.

- **RQ1:** Given that design tasks are worked in a somewhat imaginary environment, how can an agent's understanding reconcile this with the fact that the solutions to the design problems must be implemented in the real world?

- **RQ2:** How can different design tasks be analyzed and compared objectively? How could we use metrics for such comparisons to create agents optimized for different disciplines and specialties?

1.1.2 Knowledge and State Representation

Solving a problem requires a way to represent it. Natural language is likely too imprecise, ambiguous, and complicated for a design agent so we need to develop a more structured way of representing problems, solution states, and the agent's knowledge. This representation should also allow the agent to apply its prior knowledge to accelerate the process of solving a problem.

- **RQ3:** How can an agent's prior knowledge and current progress toward a solution be integrated with a given problem description? What kind of structured high-level representations are necessary and sufficient to describe a design task such that an autonomous agent can go about finding solutions?

1.1.3 Autonomy and Cooperation

Autonomy and cooperation are to some extent inversely proportional. Like a human design partner, a design agent should be able to work independently or cooperatively depending on the needs of the situation. Since an EDA cannot be a "set it and forget it" computer program, it is important to consider the features that will enable it to work alongside a human engineer to a configurable extent. Additionally, given that an EDA will need to be cooperative in some way, one must consider how exactly this will be implemented and in what way the EDA will work to incorporate feedback regardless of source.

- **RQ4:** How can a design agent be capable of both independent and cooperative workflows? In a cooperative workflow, how can it identify shortcomings in its understanding and ask for help?

- **RQ5:** A design agent may receive feedback from a human or the environment at any point in time. Could a high-level description language facilitate this? How do we account for this in the agent's architecture?

1.2 Hypotheses

Working from these research questions, it is possible to formulate a series of hypotheses based on the work already done in this area. This investigation will aim to support or discredit each of these hypotheses.

1.2.1 Problem Analysis and Understanding

- **H1:** Design tasks are a subset of a more general class of tasks. They are timeless but share their causal nature with real-world tasks. As such, existing work in task-analytic fields can be applied to quantify the complexity of a design task and use this as a metric to match agents with tasks. The causal relationships between design elements can be used to reconcile the operation of imaginary designs with their real-world prototypes; the same operational principles and relations should hold true for both. Since an agent's knowledge and experience may be unique to it, agents could specialize in specific disciplines.

  There are many ways that problem analysis and understanding could be investigated, but the first steps will require analyzing task difficulty and handling the difference between the imagined world of the design and the real world of the eventual solution. Future research should focus on the specific mechanisms involved in problem decomposition and the extent to which difficulty analysis is applicable in designing, selecting, or operating a design agent.

1.2.2 Knowledge and State Representation

- **H2:** A description language can be developed that fulfills the roles of storing intermediate states as the design progresses as well as communicating the problem and goal states. This language can be human-readable, able to be translated to and from the language of the high-level designer, and able to be filled in by the EDA's other components to fulfill all communication needs.

  The development of a description language will be a powerful first step in understanding the problem of knowledge and state representation. Such a language can form the basis for future work and can then be further refined or redeveloped to handle the greater capabilities of future design agents.

1.2.3 Autonomy and Cooperation

- **H3:** Autonomy and feedback in an EDA are largely configurable as a matter of setting a lower bound on the confidence level of the agent's models; below a certain level, it will give up and ask for help. All feedback from any source can be facilitated through the description language.

  From this initial work, it will be valuable to investigate in what ways exactly an agent and human can best work together. Perhaps there are optimal workflows that can best make use of the unique strengths of human and artificial engineers alike.

1.3 Contributions

Though there is still much to be done in EDA development, it is believed that this thesis makes some simple contributions to the field. These contributions cover engineering design, EDA architecture, and artificial general intelligence (AGI).

In terms of engineering design, Chapter 3 reframes the process in a way that can be performed by an autonomous agent. Given that many discussions of the design process presuppose human intelligence, it is important to be able to represent engineering design as something that can be performed by a system with less-than-human intelligence. Similarly, Chapter 4 provides a structure and formal language with which to perform the process of problem decomposition. Instead of breaking down a problem intuitively, this chapter provides a more rigorous causal approach that can be taken by an autonomous design agent.

Following this work, Chapter 5 suggests an architecture for such a design agent. The EDAs proposed use a novel hybrid architecture in which problems are first approached conceptually and then iteratively solved with the help of a generative designer. Chapter 6 then builds a small prototype of this architecture and validates it experimentally.

More generally, the process of developing an EDA can help guide the way toward more intelligent and capable AGI-aspiring systems. This thesis' EDA architecture uses an AGI-aspiring system at its core but augments it with a variety of knowledge bases, a description language, and other components. Should an AGI-aspiring system attempt to wholly take on the process of engineering design, it can use this architecture as a roadmap for the extra features and functionality it will need to integrate.

Finally, this thesis conducts a brief exploration into the safety of a fully functional EDA and the implications it might have. Appendix A considers how powerful it can be to have easy access to engineered solutions and how to prevent this from "falling into the wrong hands", so to speak. With a system as potentially powerful as an EDA, it is believed that this discussion is best started as early as possible.

1.4 The Path to an Engineering Design Agent

The nature of engineering design means that any agent capable of autonomously solving a design problem is going to have to be fairly advanced. The EDAs proposed in this thesis are no exception, they are complicated systems that require the development and integration of a number of prerequisite components before they can achieve full functionality, many of which do not yet exist or do not exist in the required capacity. This exploration will not attempt to solve all of these prerequisite problems but it will address enough of them to be able to create a functioning technology demonstrator. The goal is not just to develop a framework for an EDA, but to properly prove the concept with a system capable of very basic design.

As such, this thesis takes a breadth-over-depth approach to these problems. Many different issues from problem description to experiential learning are discussed in sufficient detail to allow for the creation of the demonstrator system but not as much detail as would require excessive amounts of time. An attempt is made to draw a throughline across these problems in the hopes of demonstrating where they fit into EDA development and why they are key parts of the process. In keeping with these limitations, a large part of the conclusion chapter will be dedicated to discussing areas for future work that are the intended focus of this author's doctoral studies. This thesis represents the crystallization of all of this author's efforts to develop an EDA and all of the sub-projects and incremental problems that were explored in that process. This has been no small challenge and it is hoped that the work described here is useful in the development of some of the first practical engineering design agents.

Chapter 2

Related Work

The process of solving an engineering problem requires skills in, among other areas, problem-solving, knowledge management, autonomous cumulative learning, and cooperation. Though these have been the subject of much research into AI and engineering, it appears that limited work has been done in applying them to the challenge of autonomous conceptual engineering design (ACED). Most prior work appears to be focused on creating a more general AI agent, better understanding the engineering design process (cf. Venkataraman and Chakrabarti, 2010), or streamlining a human engineer's workflow through automation (cf. McComb et al., 2017; La Rocca, 2012; Plappert et al., 2022). Of this work, the most closely related field is knowledge-based engineering (KBE), which seeks to automate repetitive tasks to allow human engineers to focus on the more interesting and important parts of the design. To this author's knowledge, however, this field has yet to attempt the creation of a design system with *agency* and a *skill level* on par with that of a human engineer. This is what this thesis aims to work towards.

This chapter is broken up into three main sections, one for each of the categories of research questions outlined in Section 1.1. First, we consider the challenge of *problem analysis and understanding*: what it takes to understand a problem within a framework of existing knowledge, decompose it into its component parts, and refine it into a solution. Much work has been done in these areas but so far little has been applied to autonomous engineering. Next, the focus is on *how knowledge and solution states are represented* within an engineering system. The goal is to determine the best way to encode knowledge such that an agent can apply it across a variety of problems and still be able to expand its understanding through autonomous learning. Again, this has been researched significantly but the focus is rarely on both agency and engineering. Finally, we look at factors influencing *autonomy and cooperation* in which the goal is to create a design agent capable of working on its own or cooperating with a human engineer. While this has been explored significantly, only a small amount of work appears to attempt to fuse this research into a single unified design agent.

Altogether, the intent is to take the previous work done in these fields and synthesize it into an approach to the challenge of creating a design agent capable of engineering design at a near-human skill level. An attempt is made to fill the gaps of this research and to begin to apply it to the development of an EDA.

2.1 Problem Analysis and Understanding

The process of problem analysis and understanding has been explored significantly by researchers working towards artificial general intelligence (AGI), presumably because many hypothesize it to be central to this phenomenon. Of particular interest is the budding field of task theory (Thórisson et al., 2016a), which aims to create a theoretical basis for understanding tasks at a fundamental level. Task-theoretic methods represent tasks as networks of causal-relational models (CRMs) that can be used to describe knowledge of such tasks (Belenchia et al., 2021; Eberding et al., 2022). By chaining such models forward and backward, an agent can explain phenomena and develop plans to achieve goals. One of the many interesting features of task theory is how a task's level of detail is captured (Belenchia et al., 2021, p. 43), as changes in the level of detail can significantly change the task itself. All of this can allow tasks to be constructed into a sort of hierarchy with respect to an agent's goals. When a problem can be broken down, analyzed, and organized in this way, it makes the solution process significantly more straightforward.

Other researchers have focused more on the design side of the problem and posed the design process as one of exploration (Smithers, 1992) and iteration consisting of the activities Generate, Evaluate, Modify, and Select (Venkataraman and Chakrabarti, 2010). Also investigated is the process of taking an initial problem and simplifying it for an approximate and abstract design. For instance, Schut (2010) worked on a system that would simplify designs by removing degrees of freedom and approximating the problem with a much simpler mathematical representation. However, they did not propose an ontology to support this kind of decomposition for more general problems; there are not yet any general elements or techniques that can be applied to a wide range of problems. Also worth investigating is the field of axiomatic design which formalizes design into an activity centered around the independence axiom and the information axiom (Nordlund et al., 2015). The independence axiom seeks to avoid dependencies in functional requirements so as to keep the problem specification and solution as clean and open-ended as possible while still addressing the problem at hand. The information axiom focuses on keeping the resulting solution simple without redundant features. Axiomatic design also emphasizes the importance of iteration in the design process, providing it is not done excessively.

Of particular interest have been the explorations that discuss the importance of causality in design (Schut, 2010; Bhatt et al., 2021; Žavbi and Duhovnik, 2000). This is believed to be an important factor because the agent must be able to capture the causal nature of the real world for which it is designing solutions. What does not appear to have happened yet is a unification of the task-theoretic and the design sides of this problem. To this author's knowledge, no work has been done on implementing these design methodologies on a modern conceptual reasoning system such as the AGI-aspiring Autocatalytic Endogenous Reflective Architecture (AERA) presented by Nivel et al. (2013) or the Non-Axiomatic Reasoning Systems (NARS) from Wang (2006). Perhaps doing so would guide the way toward a theory of problem decomposition, analysis, and understanding.

2.2 Knowledge and State Representation

Related to this is the question of how to represent a design problem and whether that requires anything different from the representation of a conventional task. According to Smithers (1992), "[S]olving a design problem is not simply about finding a solution to a given problem, it is about devising a problem whose solution satisfies the needs or desires that motivated the initial requirement description". That is to say that design problems differ because they are poorly defined and often require an iterative process that reaches both an optimal problem description and a solution to the problem simultaneously. This lends itself well to Žavbi and Duhovnik's (2000) implementation which specifies the start and goal conditions of a design problem and allows an algorithm to interpolate between these two using a set of causal models; Schut's (2010) approach works similarly. In fact, the field of knowledge-based engineering (KBE) has been taking a similar approach for decades. KBE systems aim to capture the knowledge of human subject-matter experts and reapply that in a manner that allows human engineers to work more efficiently (Kügler et al., 2023). These systems consist of databases of knowledge, inference engines, and interfaces with their environments (Plappert et al., 2022) all of which are used to analyze designs and report possible issues and solutions to engineers. Design systems like these can be built as agents that specialize in various disciplines (Plappert et al., 2022) or as applications that harness knowledge to speed up repetitive design (cf. ParaPy, 2023, 2020). Regardless of implementation specifics, these systems all seem to revolve around databases of models of the world and engineering problems. Given all of this, it would seem that a design problem (and solution) may be represented with a set of causal-relational models (CRMs).

Though this allows the use of task theory as a foundation, it does not fully describe how the problem is posed to the EDA. Many KBE systems are built into computer-aided design (CAD) programs (Kügler et al., 2023), which allows them direct access to the parts that are already modeled but this does not facilitate a linguistic approach to problem specification. Much in the same way as one would discuss a problem with a partner, an EDA should be able to take an approximate description of the problem and use that as its starting point. Perhaps an intermediate language is required, a sort of engineering description language (EDL) that can be compiled or transformed into the language of the agent. Not much work appears to have been done on EDLs, though perhaps related work in hardware description languages (HDLs) and reasoning languages such as AERA's Replicode (Nivel and Thórisson, 2013) may guide the way. HDLs, for instance, have long been used to design integrated circuits by abstracting away from the specifics of the implementation and focusing more on the behavior of the logic itself (Wright and Iyer, 2022). This is a difficult approach to translate to wider fields of design since even the KBE systems investigated above seem to either require a significant amount of specificity or make a number of assumptions with respect to the kind of problem they will be solving. What is also infrequently discussed is the role of knowledge representation and creativity; does a perfectly transparent representation absolve the agent of the need for creativity or is the "creative spark" generally attributed to humans simply the result of an exploration process like that proposed by Smithers (1992)? The only mention of this sort of design creativity can be found in Schut's work (2010), which considers the possibility of combining, mutating, and analogizing existing knowledge or discovering new knowledge from first principles. This is certainly worth exploring and is, in fact, a crucial stepping stone in the development of an EDA.

2.3 Autonomy and Cooperation

An important consideration is the question of where EDAs fit in with human engineers. Using previous work in industrial automation, Altavilla and Blanco (2020) proposed a set of criteria for determining the level of automation of a given design process. As shown in Table 2.1, their scale ranges from Level 1, designer direction and execution, to Level 5, computer direction and execution. In between is a range of operating regions for error-checking, collaboration, and supervised work. In this way, an EDA need not be completely autonomous and could actually work to support a human designer in generating ideas, evaluating alternatives, and so on. Existing research into KBE considers agents at a variety of levels of autonomy. Schut's (2010) system exists to support an engineer and work the more repetitive parts of problem-solving autonomously; ParaPy (2023) is a system designed with a similar goal. Plappert's design agents (2022) are a little more autonomous in that they operate in parallel with the user even if they are still ultimately user-directed.

An important question here is the difference between an agent and a tool. It would appear that a number of existing KBE systems are primarily designed as tools to support engineers (Kügler et al., 2023). For the purposes of this research, a tool is considered a system that exists to serve a user and that makes no real decisions of its own. It may reason through a knowledge base, perform analyses, and develop solutions but it does not act independently of the user; it would be somewhere around

Degree of Automation	Level of Autonomy	Description
LOW	Level 1	Designer
HIGH	Level 2	Designer (Direction - Partial Execution) / Computer (support to the execution)
	Level 3	Designer (Direction) / Computer (Execution)
	Level 4	Designer (Execution) / Computer (Direction)
	Level 5	Computer Direction and Execution

Table 2.1: Levels of design automation (reproduced from Altavilla and Blanco, 2020)

a Level 2–3 on Altavilla and Blanco's scale. On the other hand, an agent is capable of breaking down problems, making decisions, and understanding what it works on. An agent could work alongside a user as an approximate equal because it also can work poorly defined problems and communicate its reasoning; such an agent would be closer to a Level 4–5 on the scale. This kind of design agent would be able to handle problems of significantly higher difficulty than a design assistance tool; the difficulty is in developing such an agent. It is believed that integrating AERA at the core of an EDA would be sufficient to create a proper conceptual design agent.

Regardless, this presents the interesting difficulty that the agent must be capable of both external critiques from a human engineer and internal feedback from its own simulations. An important question, then, is how this is actually implemented. Though the agent's thinking is conceptual in nature, it may not be human-readable as it may be largely generated from experience and possibly even play; Minsky (2022) suggests that play is a particularly effective way to build common sense and intuition. A design agent could even be set up to start with a small "seed" program that allows it to grow and learn on its own (Nivel et al., 2014b). The difficulty in all of this is that the agent will not have the opportunity to learn terms for the models it is building of its world. It could learn how to use a screwdriver but never learn the word "screwdriver", for example. In this way, an EDA's design reasoning and domain knowledge would require some transformation back into a human-readable form in order to be able to solicit and process feedback from a collaborator. The solution to this may be to teach the agent terms for the tools and objects it is working with. Perhaps this could benefit from the development of an EDL as this would be far simpler than teaching the EDA a natural language. This has been understood since 1977 when Bobrow et al. implemented the Genial Understander System (GUS) but did so in a restricted domain using a subset of the English language. This allowed them to provide a significant amount of structure to GUS' interactions with its users; perhaps an EDL could be similarly constructed.

2.4 AI and Cognitive Architectures

No discussion of problem-solving in AI would be complete without at least a brief overview of the work done in the field so far. Not only is it important to see how this challenge has been met by others, but it may save valuable time to implement an EDA with an existing cognitive architecture rather than building one from scratch. Out of the many AGI-aspiring agents and frameworks developed, this thesis considers SOAR (Newell and Rosenbloom, 1986), ACT-R (Anderson and Schunn, 2000), OpenCog Prime (Goertzel, 2009), NARS (Wang, 2006), and AERA (Nivel et al., 2013).

Soar, originally developed in the early 1980s, is a well-known architecture aimed at general problem-solving and general intelligence (Newell and Rosenbloom, 1986). The goal is for Soar to be able to handle a wide range of tasks with a wide variety of problem-solving processes all while learning and growing its intelligence. A feature of note is that Soar uses symbolic representations of its knowledge and considers its environment and its task as being inextricably linked as a task-environment tuple. This lets it reason within well-defined contexts and gives it an understanding of how its solutions may be highly context-dependent. It is still being developed as Soar 9 by the University of Michigan Soar Group (2024).

ACT-R is a different architecture with an interesting feature: it considers knowledge as either declarative or procedural, much like the distinctions made in cognitive neuroscience (Anderson and Schunn, 2000). Its declarative knowledge holds facts about the world and its procedural knowledge holds models of interactions that let it accomplish its goals. It reasons about these in a hybrid symbolic/sub-symbolic system that can integrate prior knowledge with sensory perceptions. It is still being developed by the ACT-R Research Group (2013) at Carnegie Mellon University.

OpenCog Prime, in a similar vein to ACT-R, separates its intelligence across several different algorithms and processes (Goertzel, 2009). Among these are declarative and procedural memory but also memory pertaining to attention, sensation, intentions, and imagination. These are managed and trained by a variety of systems including probabilistic logic networks, evolutionary algorithms, and attention networks. It has been applied to, among other problems, simulating virtual pets in virtual environments. OpenCog Prime and other systems are still being developed by the OpenCog Foundation (2024).

NARS, the non-axiomatic reasoning system, is a framework out of Temple University that approaches problem-solving and knowledge management with non-axiomatic logic (Wang, 2006). It is designed to operate under the assumption of insufficient knowledge and resources (AIKR) which allows it to prioritize its goals and work outside of an idealized environment (Wang, 2004). An open-source version called OpenNARS for Application is currently available on GitHub and is still actively developed by the Department of Computer and Information Sciences, Temple University (2020).

AERA, the autocatalytic endogenous reflective architecture is a system developed for causal reasoning, experiential learning, and real-world operation (Nivel et al., 2013). Like NARS, it works under AIKR and seeks to reason about the world around it by building models of its environment. Unlike NARS, AERA uses a system of facts, models, and composite states rather than reasoning with explicit logical statements. AERA is designed to be scalable, it can be run on a laptop, at a large scale, and even on a computing cluster if desired (Nivel and Thórisson, 2013). Like NARS, AERA is open-source and available for download online; it is still under active development by the Icelandic Institute for Intelligent Machines (2021).

All of these systems are fascinating glimpses into the world of AGI research though not all of them are the best fit for an EDA. For example, a simple EDA will most likely only want one source of reasoning and decision-making since there is no need to partition its knowledge into declarative, procedural, and other categories; this eliminates ACT-R and OpenCog Prime. Additionally, an EDA will need an understanding of causality in order to use the CRM task representations previously; this eliminates Soar and NARS. AERA employs one unified repository of knowledge and has the ability to autonomously build explicit causal models of the world. It is for these reasons that, of the architectures considered, AERA was selected for this thesis. All this is not to say that an EDA *must* be built with AERA; any sufficiently-capable architecture should be able to support an EDA's workings. This thesis, however, will focus on applying AERA to the development of an EDA.

2.5 Summary

Autonomous conceptual engineering design represents a fascinating intersection of fields as diverse as intelligence, problem-solving, design methodology, experiential learning, and collaboration. Given this, there is a significant amount of work available for the foundation. Grounding the engineering design process in task theory would be an important step in teaching it to an intelligent agent. Exploring the nature of design problems themselves would aid in the development of methods to solve them and how knowledge can be represented to best facilitate this. Building on work in automation and learning can allow EDAs to work in tandem with human designers and learn from their feedback. Finally, starting from an existing cognitive architecture allows this design agent to benefit from the efforts of previous AGI researchers. While a significant amount of work has been done, there is still more to go before an EDA can be realized.

Chapter 3

The Engineering Method & AI Agents

When developing an agent for engineering design, it is worth taking some time to first understand what is involved in the engineering design process. There does not appear to be a formal, definitive, or codified engineering method, it seems to be more of a set of techniques, steps, and rules of thumb that engineers learn during their training. Despite the math commonly associated with the field, engineering design may be more of an art than a science, at least at the conceptual stage of design. This chapter explores some of the common approaches to engineering and some of the features that have been suggested to be a part of the engineering method.

These features are then used to select an AI agent that will live at the heart of this thesis' design agent architecture. Among its other capabilities, AERA (the Autocatalytic Endogenous Reflective Architecture) is a framework developed for high-level reasoning, problem-solving with causality, and reflection (Nivel et al., 2013). This makes it the perfect choice for a design agent that will require the ability to solve problems and manipulate information at a high level. This chapter ends with an in-depth discussion of AERA, its programming language Replicode (Nivel and Thórisson, 2013), and how they can be used to create an agent capable of engineering design.

3.1 What is the engineering method?

While it has been studied extensively, the engineering method is difficult to describe succinctly and there is still no consensus on exactly how engineers design solutions to problems. Some models consider engineering in terms of the activities undertaken by the engineer. One useful model in this regard is the Generate, Evaluate, Modify, and Select (GEMS) Model (Venkataraman and Chakrabarti, 2010). In this model, one generates ideas, evaluates them with a series of tests, modifies them according to feedback, and eventually selects a final design. Alternatively, we can consider the process in terms of design outcomes and how potential solutions move from high-level descriptions to low-level solutions. The SAPPhIRE model (Bhatt et al., 2021) works down from actions to part designs considering every part of the problem at decreasingly low levels of detail as it goes. Still another consideration is the causal nature of the processes involved and how these are implemented by physical parts. Much as one physical process can lead to another (consider water turning a wheel that turns magnets to produce electricity), one can consider a design in terms of the causal relationships between each step of the solution. This is proposed by Žavbi and Duhovnik (2000) in their paper "Conceptual design of technical systems using functions and physical laws". Their approach is actually quite similar to the one taken in this thesis, albeit the engineering design agent (EDA) described here works at a more conceptual level.

As Research Question 1 (page 5) states, an additional challenge is that design problems are often worked on paper or in a sort of "imagined" environment. Even though the solution will eventually be built in the real world, the process often starts by creating a hypothetical solution to the problem and will only advance to building a prototype once an idea for a design is complete or there is a need to run an experiment in the real world. The challenge is that this imaginary environment has different properties than that of the real world; time doesn't pass for a design on paper, for example. One of the only benefits is that both environments can allow for causal links between models of the world and that, as long as the agent's models are accurate, the same causal links should apply in the real world as on paper. In other words, the same physics will work for both the design and the real-world implementation.

The fact remains that the engineering method can be learned and understood intuitively but is difficult to teach or program explicitly. This makes it difficult to create an agent capable of solving problems using the engineering method.

3.1.1 Starting a New Design

Every design starts with a problem to be solved. As asserted in Hypothesis 2 (page 7), these problems can be represented in a structured manner using a description language. However, the formulaic nature of such a language only provides a false sense of security as the first problem description an engineer receives tends to be vague and significantly under-constrained, even if it is specified in a structured manner. That is to say that it is in serious need of clarification. To describe this, Smithers (1992) uses the idea of an Initial Requirement Description or IRD. IRDs are incomplete, inconsistent, imprecise, and ambiguous; they may leave out important criteria, contain contradictory or impractical criteria, fail to distinguish criteria from each other, and fail to properly identify criteria priorities and/or subsumptions. Consider, for example, that you are asked to design a bridge across a waterway. Such a simple description leaves out a lot of information: How big is the waterway? What is the water like? Are we carrying anything across with us? How many people should cross in a day? Is the design to be permanent? Do you want a structure or something like a ferry? Can you swim?

To proceed in the design, one must clarify the IRD and extract the fundamental aspects of the problem, not unlike how customer needs must first be translated to functional requirements in axiomatic design (Nordlund et al., 2015). Perhaps the best way to do this is to work through the IRD iteratively to identify and address the problem areas. Conveniently, this often occurs in parallel with the design process since the act of transforming an IRD into a solution tends to reveal these issues. Over time, as the designer asks for feedback and the client provides additional information, the problem description evolves and the solution progresses towards something that better fits the desires of the client. Or, to put it more succinctly:

> "[S]olving a design problem is not simply about finding a solution to a given problem, it is about devising a problem whose solution satisfies the needs or desires that motivated the initial requirement description"
> - (Smithers, 1992)

3.1.2 Creativity in Design

An important part of the design process is that of creativity. Problem-solving pairs well with creativity and some of the best innovations come from creative and "out of the box" thinking. With this in mind, it is clear that an EDA should at least aim to implement some form of artificial creativity. In humans, we tend to think of a "creative spark" that yields a breakthrough in a problem; this is not the kind of creativity discussed here. While it is possible that a process in the EDA could be implemented to trigger random or strategically relevant existing knowledge in an attempt to bring in a new perspective on a problem, that will not be the aim of this thesis.

Instead, the EDA developed here will implement a sort of "creative fallbacks" approach similar to how Schut (2010) imagines creativity in conceptual design. Consider the bridge example from the previous section: imagine that you want to build a bridge across a river but the riverbed is far too soft for a solid foundation. Instead, you will need to select a bridge that builds only on either side of the river. Perhaps a bridge made up of a single giant arch, for example. But what if that does not work? What if the river is too wide for an arch or even a suspension bridge? In this case, you may have to invent a new kind of bridge from scratch to solve the problem. After all this thinking, your solution could be a kind of floating bridge that uses pontoons to support a road deck just above the water's surface. This is the "creative fallbacks" approach in action: you start by applying what you already know, then you analogize less-related knowledge to try for a solution, and then you work from first principles to create something new.[1] At no point in this process was there a spark that led to a perfect and unexpected solution, it was just a matter of running through your existing knowledge until you found something you could apply to solve the problem. It is also worth pointing out the similarities between the "creative fallbacks" approach and the already-established TRIZ theory of design which grades solutions on a scale from those that provide minor improvements by applying existing technology up to revolutionary solutions that incorporate "new discoveries of [the] rules of nature" Royzen (1993). With an approach like this, a knowledgeable designer should be able to work their way through many different problems of differing levels of novelty and difficulty.

This is the kind of creativity that was considered for this EDA. When the agent is unsure of how to proceed, it works its way through what it knows in the hopes of finding a solution somewhere. This may even require traversing different levels of detail; if you are not able to just drop in an already-understood type of bridge, you may need to consider the physics and materials science of a new concept. Any EDA will need to be capable of this kind of thinking to work its way around design challenges.

3.1.3 Top-down versus Bottom-up Approaches

A common rule of thumb in design is to break large problems into sub-problems. Consider the bridge example: in the process of refining the IRD, we have discovered that the client would prefer a road bridge that can carry cars, pedestrians, and bicycles. From this, we can use our existing knowledge of bridges to break this problem down into some sub-problems: What kind of bridge do we want? How will we build its foundations, what will it sit on? What kinds of materials are needed and how will we select them? These sub-problems make the design more approachable by breaking it up into more manageable chunks but they also complicate the process by which we solve the problem. Note that some of the bridge example's sub-problems are interdependent (the foundation depends on the kind of bridge) and some cover multiple levels of detail (the materials dictate the kind of span but must also be tested on a small scale). The way one handles these complexities usually lies on a spectrum from totally top-down approaches to totally bottom-up approaches.

On the top-down side, we have models like the aforementioned SAPPhIRE (Bhatt et al., 2021), which works a problem through seven levels of detail from the abstract level of actions to the concrete parts that perform these actions. As an example, Bhatt et al. cover the design of a tea kettle that starts with the top-level goal of boiling water. From there, they see that water can be boiled by increasing temperature, reducing

[1] It is worth noting that one could even seek solutions by questioning the first principles themselves. One could perform experiments and even rewrite the laws of physics if the results start to show promise. This is a little more science than engineering, though, so it is advised that most engineers humbly admit defeat and look for solutions elsewhere before attempting to rewrite the laws of the universe.

pressure, or some combination of the two. Then, they see that increasing temperature can be accomplished either by generating or transferring heat which can be accomplished through chemical reactions, mechanical work, or various thermodynamic processes for heat transfer. Of these options, they settle on conducting heat, a process that is moderated by the thermal conductivity of the body, the temperature of the external heater, and so on. Finally, this process arrives at a solution that we would know as a kettle; a system that boils water by increasing water temperature using heat transfer via conduction that is affected by a variety of physical properties. While SAPPhIRE is not the only top-down model, it is representative of the overall process that top-down approaches take: Work from the abstract to the concrete to refine a problem description into a description of a solution.

The difficulty with a top-down approach is that it assumes a significant amount of background knowledge and understanding. For the kettle example to work, one must already understand everything about the methods of boiling water, the physics of heat transfer, and all the knowledge required to implement these in a physical system. While some design problems will work perfectly in this well-understood environment, any problem that contains some uncertainty will have trouble. Dealing with uncertainty requires experimentation and a purely top-down approach leaves very little room for this. In the kettle example, imagine that you had never learned that heat could be conducted between objects. In that example, you might conclude that the best way to boil water would be through exothermic chemical reactions. While the merits of an explosive tea kettle are beyond the scope of this thesis, it is easy to see that there are more efficient solutions to the problem of boiling water.

Bottom-up approaches, on the other hand, are made almost completely of experimental processes. As an example, consider the way babies learn about the environment: they play with things. While they play, their brains make connections, form concepts, and develop a model of the world (Minsky, 2022). During these experiments, a baby will learn to solve basic problems using knowledge gained not from some hierarchy of concepts but from incrementally acquired experience in the real world. The same kind of experimentation applies in science and engineering, too. When Josiah Wedgwood was developing a process to produce his famous Jasperware ceramics, he did not find a solution through a top-down understanding of the chemistry and the science of various clays and additives (Hammack, 2023). Instead, he spent years testing different ingredients, combinations, and production techniques until he, at last, arrived at a desirable result that would catapult him and his business to world renown. In short, a purely bottom-up approach relies almost entirely on experimentation to arrive at a satisfactory solution. In this way, it does away with issues of uncertainty because these are worked out during the experimentation process.

Bottom-up is not a perfect solution, however. It can take years to build sufficient knowledge to solve a problem and the search can sometimes be entirely unguided. Even intuition can only do so much to guide a blind search so experimentation alone can take a very long time to be useful.

Given this, most engineers will tend to use a hybrid approach that combines the efficiency of top-down with the ability to handle uncertainty of bottom-up. In his book "The Things We Make", Hammack (2023) discusses an engineering method guided by a set of different principles. One principle is that "engineering is not applied science, a phenomenon need not be understood scientifically, mathematically, philosophically, or any other way before engineers use it to create something useful" (Hammack, 2023, p. 47). In other words, it is entirely possible to solve a problem using a technique or technology that we *know* works even if we cannot explain exactly *why* it works. To facilitate this, Hammack proposes the *rule of thumb* as a way of encapsulating knowledge that may not fully describe a phenomenon but that is sufficient to point in a direction that has already shown promise. This yields a more bottom-up definition of engineering as a process guided by experimentation as well as prior knowledge. As an example, Hammack discusses the methods of the stonemasons behind some of the most famous cathedrals of the Middle Ages. At the time, there was little understanding of material strength, no equations for load transfer, and no considerations of factors of safety. Instead, the stonemasons used a method of geometric construction that used little more than rope and a drawn template of an arch; there was no math or calculation involved, just a sequence of steps that was found to produce desirable results. In this way, the engineers of the time were able to produce beautiful buildings that have stood the test of time even with their limited understanding of the physics that kept them standing[2]. In this way, engineering is a varied and complex process that cannot be categorized into simply top-down or bottom-up. This is what makes the engineering method so hard to implement in an artificial agent. Engineering is far more than just a simple procedure, it has required millennia of work to develop the problem-solving knowledge and techniques we use today.

With this in mind, we see that an EDA must be capable of a variety of different tasks in order to become an accomplished engineer. While we can certainly give an EDA a head start by loading it with some of our most useful knowledge and rules of thumb, it must also be able to form its own understanding of the world and learn how this can be applied to solve problems. It should be capable of performing experiments, analyzing data, and extracting patterns from its observations. Furthermore, it should be capable of using its models of the world to guide its search for a solution even when it does not fully understand why the models work. Finally, it should know when to rely on pre-programmed scientific knowledge and when a gap exists that must be filled in either by experimentation or asking for help. Altogether, this approach does not need to be perfect and indefeasible so long as it can point the agent in the right direction and guide it to a better solution.

[2]This is not to say that we should attempt the same feats today given that the math of our current structures is very different from that of the structures of old. Engineers should absolutely benefit from our greater scientific understanding of the physical world. As Hammack puts it, science is "the gold standard for rules of thumb" (Hammack, 2023, p. 110).

3.2 Agents for Exploratory Design

As a design agent works, it should refine its understanding of its problem and develop a better understanding of the world and context in which its solution will operate. We can liken this to a sort of exploration since the agent will be performing experiments on the world, forming models of understanding, and linking to existing knowledge. Indeed, this is exactly the kind of exploration proposed by Smithers (1992). With this in mind, any agent capable of performing this kind of exploratory design must be able to *reason* about its environment and *reflect* on its knowledge. One methodology worth keeping in mind is constructivism as proposed by Thórisson (2012). An agent designed with the constructivist style in mind should be able to autonomously grow its knowledge from a small initial seed; it would not be limited by its initial knowledge nor by less-than-general architectures. Such an agent would be well-suited to an environment of exploration and learning.

Unfortunately, the constructivism requirement means that almost all contemporary AI architectures are not suitable for use in a design agent as, to the extent they consciously understand the world, they do so through statistical models. While this is a very powerful approach for narrow problems, it does not permit the kind of reasoning with causality and iterative reflection that is needed for engineering design. Additionally, while a KBE system must implement reasoning over causality, it may not be able to experiment and grow its own knowledge. This means that such a system will likely struggle with uncertainty and so will be confined to operating within its area of knowledge. Instead, we need a reasoning system from the field of artificial general intelligence (AGI) capable of cumulative learning (Thórisson et al., 2019).

3.3 AI and Cognitive Architectures

As mentioned in Chapter 2, a great many cognitive architectures have been developed in the pursuit of artificial general intelligence (AGI). Of particular interest in this thesis have been SOAR (Newell and Rosenbloom, 1986), ACT-R (Anderson and Schunn, 2000), OpenCog Prime (Goertzel, 2009), NARS (Wang, 2006), and AERA (Nivel et al., 2013). Soar has existed in various incarnations since the early 1980s and is designed for general reasoning and problem-solving with symbolic representations of its knowledge. ACT-R and OpenCog Prime take a more hybrid approach by using both symbolic and sub-symbolic reasoning to manipulate knowledge. They also organize their knowledge into categories based on function such as declarative knowledge, procedural knowledge, and so on. Finally, NARS (the non-axiomatic reasoning system) and AERA (the autocatalytic endogenous reflective architecture) are both more modern systems that keep knowledge in a single unified pool regardless of its function. They are also the only ones on this list with a well-documented method for achieving cumulative learning (Thórisson et al., 2019). Both systems perform symbolic reasoning to solve problems under the assumption of insufficient knowledge and resources meaning they are well-equipped to operate in a real-world environment with imperfect observability and limited interactions. This last feature makes both NARS and AERA good candidates for the reasoning system at the heart of an EDA.

NARS and AERA are powerful architectures; both of these systems are capable of observation, experimentation, and representing their knowledge with a variety of logical arguments. They also work within the 'peewee granularity' principle of assembling a number of small atomic models of the world into larger understandings of more complex concepts (Steunebrink et al., 2016). Though these systems are quite different from each other, they have both demonstrated the ability to learn a wide variety of tasks; AERA has learned through observation how to participate in a TV-style interview (Nivel et al., 2014a) and NARS can be used to detect anomalies in smart city vision data (Hammer et al., 2020). One key difference, however, is that AERA's attention mechanism allows it to more efficiently focus on and adapt to changes in its environment. This ability to prioritize information makes it a better candidate for an EDA though one could surely also be built with NARS. AERA has the additional benefit of being able to explicitly represent causal relationships; this makes it particularly useful in comparison to other available agents. Finally, development efforts will be supported to the author's proximity to the AERA team.

3.4 AERA and Replicode

AERA (Autocatalytic Endogenous Reflective Architecture; Nivel et al., 2013) is an AGI-aspiring goal-driven reasoning and learning framework capable of learning from experiments, knowledge abstraction, and applying its models of the world to solve problems. This makes it an excellent fit for handling the reasoning within an EDA. AERA is built on a significant body of research and takes a constructivist approach to general intelligence. That is to say that it starts from a small seed program and learns from experience through experimentation and observation, as it interacts with the world, rather than starting with a massive pre-programmed knowledge base of how things work. As it observes and experiments, AERA builds models and makes predictions about its environment. If a prediction is proven correct, it adds weight to the model. If not, it decreases the value of the model; over time, models will be deactivated if proven to be consistently incorrect. These models can represent any phenomenon at any level of detail so long as it can be expressed in terms of a causal relationship. This generality allows AERA to work on almost any task; most impressively, it was able to learn on its own how to participate in a short TV interview involving spoken language and nonverbal signals (Thórisson et al., 2014). In theory, this should make it capable of performing engineering design as well.

When working with AERA, one of the most important parts is setting up the seed program as it must contain the necessary knowledge to allow AERA to bootstrap its way to a higher level of capability (Thórisson, 2020). One of the roles the seed program plays is providing AERA with an understanding of how to interact with its environment. The seed should contain some basic models of AERA's embodiment that allow it to execute commands with actuators within the environment. These commands will be transmitted to an I/O device and performed in the environment, with the resulting changes returned to AERA so it can observe the effects. The I/O devices AERA uses can take a variety of forms; they can be simulated robotic actuators, real-world robot bodies, connections to networked programs, languages for communication, and so on. All of this provides a means by which AERA can learn more about its environment on its own. With that in mind, another important function of the seed program is to provide a directive for AERA to explore and perform experiments. A very simple way to do this is through motor babbling, which executes meaningless and somewhat random interactions in the hopes of discovering something about the environment. As an example, the `hand-grab-sphere-learn.replicode` [3] example program starts by alternating between performing goal-driven problem-solving and motor babbling. Even without being supplied with any knowledge on how to use its actuators and sensors, this is enough for AERA to learn how to interact with its environment and achieve its goal of picking up a sphere, moving it to a target location, and removing any obstacles along the way. In this vein, a final important function of the seed program is to provide AERA with its goals. Goals are expressed in terms of a desired environmental state, AERA's job is to find a way to manipulate its environment in order to express that state; in `hand-grab-sphere-learn.replicode`, the goal was to move the ball to a specific position. AERA can operate with goals at any level of complexity and can even work to satisfy multiple goals simultaneously as would be needed in solving a complex engineering problem. In this way, setting up an AERA agent to perform a task is largely a matter of finding the right starting knowledge and learning environment.

All of this is programmed in Replicode (Nivel and Thórisson, 2013), a functional language designed for reasoning agents capable of cumulative learning and reflection. Instead of using a series of imperative statements, Replicode encodes knowledge in causal models and composite states which can be activated when certain environmental patterns are experienced. Replicode encodes *programs* in a similar way; they can be triggered to run periodically or on an event-driven basis. As AERA learns, it can store knowledge by writing its own Replicode, in addition to the seed program used to kick-start the process. Finally, the language is designed such that all programs and models are used in parallel, possibly even to the extent of spanning multiple processors in a computing cluster.

With AERA at its core, an EDA could leverage this existing work on general intelligence and focus on developing the engineering side of the problem. When set up

[3] Available on AERA's GitHub repository (Icelandic Institute for Intelligent Machines, 2018)

with the right seed program, it can reason its way through design problems by experimenting with its world, making plans, and adding to its subject matter knowledge. Its ability to communicate and interact will also be essential when it comes time to prompt the user for feedback or help on a design problem. Finally, using AERA in an EDA means that any updates or improvements to AERA get passed on to the EDA without much additional effort. It is for these reasons that AERA was selected as the basis for the EDAs described in this thesis.

Chapter 4

Problem Decomposition & Description

Problem decomposition, problem analysis, and problem description are all facets of the same process; they all seek to better understand a given problem such that a solution can be synthesized. As Hammack (2023) writes, one of an engineer's most useful rules of thumb is that problems become easier once you break them down into smaller problems. As such, the first step an engineering design agent (EDA) will take is to decompose the problem and try to understand it. The goal is to convert a big unfamiliar problem into a series of smaller more manageable problems each in a more familiar context. Since these smaller problems are often interdependent, they can be linked to each other to form a task hierarchy that will guide the agent through the solution; this also has the benefit of imposing a causal structure on the problem, making it easier to reason about. Even better, the design agent can accelerate its problem-solving by further linking these problems to applicable knowledge as per Research Question 3 (page 6). The now-decomposed problem can also be analyzed to assess its level of difficulty in terms of solution complexity and how familiar the agent is with the context in which the problem is solved.

In order to do this, we first need to develop a way for the agent and the human engineer to describe problems in a structured and unambiguous way. This will let a human engineer pose the problem to the agent, allow the agent to describe its solution to the human engineer, and facilitate the linking of the problem to the agent's prior knowledge. One solution to this challenge is a formal description language that is simple enough for that agent to understand yet general and concise enough to be read and written by the human engineer. Since this language will be optimized for engineering, we will consider it an engineering description language (EDL). This chapter explores the features of such a language and ends with a small example language to be used in basic EDAs.

4.1 An Objective Basis for Problem Description

Task theory (Thórisson et al., 2016a) is a relatively new research area that investigates the nature of tasks within a mathematical and causal framework. These tasks can be anything one could conceive of; it could be as simple as washing the dishes or as complex as designing a moon rocket. The goal is to find ways to break these tasks down into atomic units, analyze the relationships these units have with each other, assess the difficulty of the task, and so on. The Holy Grail of any task theory is a method that allows an agent to see a problem, analyze it fully, and compare it to any other problem to determine how difficult it will be to solve. Since, as posited by Hypothesis 1 (page 6), design problems are a subset of all tasks, we can use task theory to figure out how an agent might go about solving engineering problems by modeling them as tasks. This, however, is a challenge.

A useful first step in this process is to consider how an agent might use the physics of a problem to better model it, after all, this is often the first exposure humans have to a situation. It could do this by making slight changes to the environment and its context to see how the outcome changes. Imagine examining a bicycle and noting that applying a force to its pedals makes it move forwards but only when the brakes are released. The agent may not know why this works, but it can at least explain this simple causal link between its actions and the observed result. It could also take the problem apart and explore it on a lower level of detail (LoD). Consider taking apart the bicycle, examining all its different components, and trying to figure out what they do.

Regardless of the approach, the goal is to arrive at a causal representation of the task. That is, we want to be able to represent a task entirely with causal-relational models or CRMs (Thórisson and Talbot, 2018; Thórisson et al., 2019; Belenchia et al., 2022). CRMs are small conceptual models with inputs, outputs, and requirements that can be used to model any processes that are causally related. When an input to a CRM exists and the requirements are met, one can expect that the outputs will occur. As an example, Figure 4.1 shows an abstracted example CRM for a very simple representation of the relationship between forces when using a bicycle. When an agent pushes the pedal down, it expects to see a change in the bicycle's velocity; this is expressed in the square model. This, however, will only work if the brake calipers are open as shown in the circular composite state. To ensure this, the agent must first release the tension on the brake line. With this simple model, it can start to understand how a bicycle works.

A useful property of CRMs is that their inputs and outputs can be chained together to produce a more complete picture. They can be chained forward to make a prediction or backward to make a plan to achieve a goal. In the bicycle example, one could add another CRM that states that an agent can only release tension in the brake line if it first lets go of the brake handle. It could then make the prediction in Figure 4.2 that suggests that pushing down on the pedal and letting go of the brake handle

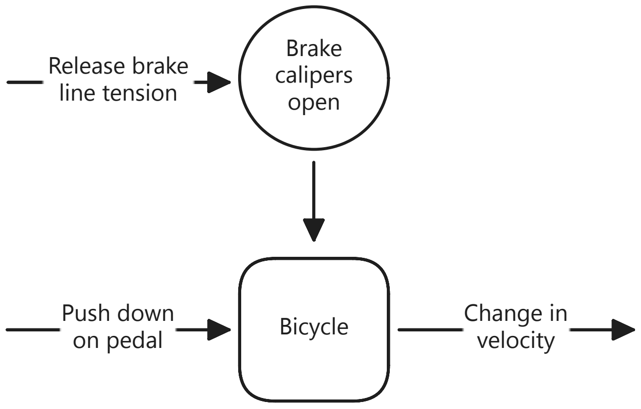


Figure 4.1: A simple CRM for a bicycle. Models are shown as squares and composite states as circles.

will cause the bicycle's forward velocity to change. It could even look at a lower level of detail and examine the requirements for the brake model. Perhaps the agent can ignore the finer details on its daily commute but, during some maintenance, it may notice that the brake calipers only work when they are properly tuned up. Finally, the lengths of these causal chains can be arbitrary; a reasoning system can use however many CRMs it needs to fully represent a task, no matter how complex.

An extension to this theory can allow this to work across multiple LoDs, as well. One way to do this is to use super-models that contain CRMs within themselves. These can set up hierarchies of models that describe different parts of a problem and how they relate to each other. In the bicycle example, Figure 4.3 shows what happens when one drops to a lower LoD and looks at the physics of how the bicycle generates its forward motion. After some experimentation, an agent might notice that pushing down on the pedal affects the bike's gears. The result is an increase in the tension in the chain, which functions as an input to the rear wheel assembly (the wheel, gears, and brakes all put together). When this is the case, it sees that the wheel exerts a force on the ground and experiences a reaction force as per the 3rd law of motion (Newton et al., 1848, p. 83). This reaction force is then transmitted to the bicycle's structure, which then produces the previously-observed change in forward velocity.

Examining the requirements of the chain also paints an interesting picture. An agent may notice that the gears are unaffected by the brake calipers but that the rear wheel assembly requires the *Brake calipers open* composite state to be satisfied. Also worth mentioning is that neither the gears nor the structure have requirements of their own; they will always work so long as their inputs are present. This is, of course, just a simplification of a real system. One could easily conceive of a more complete

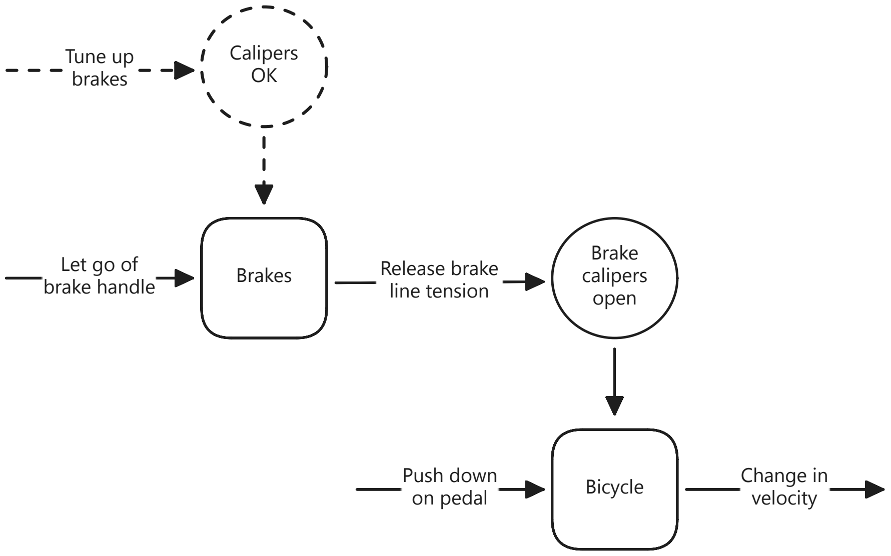


Figure 4.2: Chaining CRMs together by their requirements and state descriptions.

model that accounts for the requirements for the gears to mesh to the chain, the rear wheel assembly to be properly constructed, and the structure to be strong enough to handle the force from the wheels. Not only that but any given composite state could require multiple actions to be done for it to be satisfied; imagine bicycle gears that must be both meshed to the chain and properly lubricated in order to function. Indeed, this bicycle example is a simplified representation and could instead lie within a vast web of interrelated models, composite states, actions, and effects. For the time being, however, we must make some assumptions for the sake of simplicity and assume an idealized bicycle that always works as desired.

Using this CRM-based approach, we see that an agent can figure out how to use a system or solve a problem simply by chaining CRMs further and further backward until the inputs are all controllable variables. In Figure 4.1, we see that the inputs are *Push down on pedal* and *Release brake line tension*. While it is easy enough to push on a bike pedal, manually controlling the tension of a wire is a bit of a struggle. Instead, an agent could examine the brakes and find that they are connected to the brake handle as shown in Figure 4.2. This tells it that, instead of *Release brake line tension*, it just needs to perform the action *Let go of brake handle* – this is much easier! Ignoring the maintenance detail, there are now two controllable inputs: pedal force and brake handle position. The agent is now able to ride a bicycle (or at least make it start and stop). The same can be done when solving a problem: start from a known

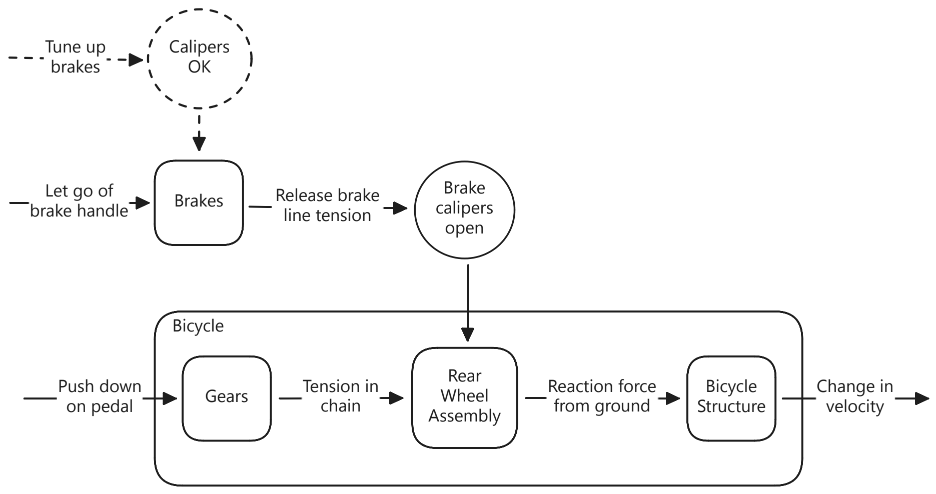


Figure 4.3: CRMs spanning multiple levels of detail

part in the problem and work through the causal chain until you reach desirable outputs and controllable inputs. In fact, this is exactly the method used by Žavbi and Duhovnik (2000).

This kind of CRM chaining can also be used to generate a complete description of a system that can be used for both explanation and experimentation. A detailed causal model of any given problem or system allows for the ability to make adjustments and create different variations on the problem. In this way, it can be related to the idea of coupling in axiomatic design (Nordlund et al., 2015) wherein analyzing and understanding interdependencies in a design can provide a fuller picture of a problem and its potential solutions. As an example, we could modify our representation of a bicycle to see what happens when the *Brake calipers open* requirement is now applied to the *Gears* model. This same process can even fill in gaps in an incomplete causal model or a chain of causal models by conducting experiments and studying the behavior of different variations on a task. Imagine you did not understand the role of a bicycle's chain. You could remove it from the model, notice that the pedal force was no longer transmitted to the rear wheel assembly, and conclude that it was an important part of the causal chain.

There is one final benefit to representing a problem with CRMs: the ability to link to existing knowledge. The CRMs used to describe a problem do not need to be specific to that problem, they could have been learned in prior experience. Consider the rear wheel assembly CRM of Figure 4.3 and how it can apply to any wheel with brakes and a power input from a chain. Perhaps the agent first observed this kind of chain drive in another context such as in a rowing machine. Since an agent's learning can be generalized across problems, it can be used to accelerate the process of understanding a new problem. If the agent already knew how a chain worked, it could retrieve those models from its knowledge base and drop them into the causal models from Figure 4.3. If it did not know how a chain worked, it could learn from the bicycle experiments and store the new models for later problems.

CRMs can cover much more than simple mechanisms or physical processes, they can capture causal relations at any scale or level of detail. For example, one could even create high-level CRMs based on the behavior of prefabricated parts or sub-assemblies. An agent could use models like these to select parts from a catalog rather than re-engineering something from the ground up. It could then simply drop that part into its design and continue to work on a different aspect of the problem. In this way, CRMs allow knowledge to be expressed in modular elements that can be assembled as desired. This style of thinking using many small models of knowledge is called 'peewee granularity' (Thórisson and Nivel, 2009) and has been proposed as an approach to building a generally intelligent agent.

With all of this in mind, it is clear that any description language will need to consider problems in terms compatible with CRMs. That is to say that it will need to be able to capture a problem in terms of sequences of causal processes thereby grounding it in a framework of experience and prior knowledge. It should also represent each model in a general enough way that it can reuse models across problems and link new problems to previously discovered models. In short, any description language will need to be rigorous and unambiguous in its causal representations to allow for a problem to be reliably and repeatably described to and understood by an EDA.

4.2 The Critical Path Model (CPM)

While the CRM method described above is undoubtedly helpful in solving problems, it is less practical in measuring a problem's difficulty or seeing whether a certain problem is a good fit for a certain agent. In order to answer Research Question 2 (page 5), we need a different metric, one that abstracts away from the specificity of causal chains and provides a simpler numerical system for assessing the difficulty of a problem.

A notable example of this is intricacy, which considers the complexity of a task as dependent on three factors (Eberding et al., 2022):

- The minimal number of causal-relational models needed to represent the relations of the causal structure related to the goal(s) *[...]*
- The number, length, and type of mechanisms of causal chains that affect observable variables on a causal path to at least one goal. *[...]*

- The number of hidden confounders influencing causal structures related to the goal. *[...]*

As can be seen, increasing the number of CRMs, length of causal chains, or number of confounders (variables with effects that are counter to the agent's goal) will increase the task's intricacy. These are the only parameters that matter; intricacy is independent of the starting values of any given variables since it only considers the causal structure of a task. These measures can even be used to assess how difficult an agent will find a task by way of effective intricacy. Effective intricacy adjusts the above factors to consider what unknowns an agent will be facing with regard to the task; the more CRMs, causal chains, or confounders it needs to learn and discover are directly correlated to how intricate a task is for an agent. Over time, as the agent learns and improves at the task, this effective intricacy value will drop to zero.

4.2.1 Four Components of a Critical Path

Since intricacy is a purely causal measure, it does not account for some other factors that may make a task challenging. First, consider the precision required at each step of the causal chain; performing a task is much more difficult if it requires millimeter resolution than meter resolution. Next, we can consider the level of detail of the task and how detailed the agent's knowledge is compared to how detailed it needs to be. An example could be in building a bicycle where the agent in question only has basic mechanical skills. Finally, there is the number of simultaneous causal chains. Though not necessarily a causal measure in and of itself, a task does get more complex if more steps have to be taken at the same time; imagine cooking a large meal in which the side and main dishes need to be ready at approximately the same time.

These new criteria are part of the proposed Critical Path Model (CPM) of task decomposition. This method considers each problem as if it were an engineering project with a hierarchy of dependent paths that must be traversed to reach the final goal. It also assumes a pre-defined minimum LoD for each problem. This minimum value is set at the start of the task by a human operator or by the agent itself and is based on the constraints imposed by the agent's highest resolution of observation and actuation. For example, a box stacking agent will only care about events occurring on the meter or centimeter scale while an agent that works with individual atoms would need to concern itself with processes on the sub-nanometer scale. The CPM offers a four-dimensional consideration of the project's critical path in terms of length, width, depth, and simultaneity:

- **Length** The *length* of the critical path is related to the intricacy of the task as defined by Thórisson and Talbot (2018); Thórisson et al. (2019). The more causal models that are required to fulfill the goal, the greater the length. Length is a simplification of intricacy as it assumes a fully-observable design space.

Without hidden confounders or differences between observed states and actual states, length can just be a simple measure of the length of the causal chain. Note that this considers the task across all LoDs; it is not enough to have a few high-level models that connect the problem and goal states, the causal chain also includes the causal steps within the high-level models that may make them significantly more intricate. For example, going to the Moon is not easy just because you have a two-step plan that says "Build a rocket" and "Fly to the Moon", you have to get into the details of these two tasks and consider just how difficult they really are. Additionally, though it may seem obvious to us, an agent must also recognize the sequential nature of this plan: one can hardly fly to the moon without first having a rocket. This is why one must consider the full length of the critical path and not simply the number of tasks to be done.

- **Width** The *width* of the critical path is focused on precision and the tightness of the preconditions of each CRM. As before, if a CRM requires millimeter-scale precision for its inputs to be accepted, it will be much more difficult for an agent only capable of meter-scale actions. As a metric, width is expressed as a weighted average of the precision of each step in the causal chain. Beyond this, width is a unit-agnostic measure of precision; a task that requires high precision in space might use spatial measures while a task with high time precision might use temporal measures of width.

- **Depth** Task *depth* considers the LoDs involved in the task. All tasks have some minimum LoD below which any details or processes can be neglected. As an example, an agent tasked with washing the dishes would not need to worry about the motion of every molecule in a drop of water. Given this — and the fact that a CRM can exist at any level of detail — it is worth considering how close the agent's plan is to the minimum LoD required to complete the task. As an example, imagine a high-level plan to build a bicycle. Perhaps you intend to go buy some wheels, steel pipe, and a seat and then put all of these pieces together. This is not enough to actually build the bicycle since that requires an understanding of how to cut the pipe, what length to cut it to, where exactly to attach the wheels, and so on. In this way, the plan is technically complete in that it connects the problem state to the goal state by way of CRMs, but it does not do this anywhere near the minimum LoD required to complete the task – the critical path is not very deep.

  Similar to the idea of effective intricacy from Eberding et al. (2022), depth can also be used to consider the effect of the agent's knowledge on the difficulty of the task. If the agent is already familiar with building bicycles, they will be able to form plans at a much lower LoD than an agent with only a rough understanding of the same task. This ability to consider the task and build a plan all the way to the minimum LoD is dubbed “knownness” since it is a measure of how well the agent knows the task.

- **Simultaneity** The last parameter is much simpler than the previous three as it is simply a count of the maximum number of simultaneous paths. To use the cooking example from earlier, consider putting together a dinner with a main course, two side dishes, and a dessert. While it is possible to cook each dish separately and have a long single causal chain, it would be better to prepare them simultaneously so no dish ends up cooling to room temperature while the others are prepared. This increases the difficulty of this cooking task despite the fact that each individual causal chain is independent of the others; there is simply more to do at any given time. It is also worth noting that this is agent-dependent. Perhaps a human would have difficulty performing so many tasks at once but a robot or factory designed for high throughput might find it trivial. The embodiment of an agent is important here; its sensors, actuators, controllers, and training must all be up to the challenge of handling simultaneous tasks. Hence, simultaneity is considered an important feature governing the difficulty of a task.

4.2.2 Limitations of the Critical Path Model

While the CPM offers a useful perspective on a problem, it can only describe how difficult a task will be to solve for a given agent. At best, one could use the scores in each criterion to determine why the problem is difficult to solve; one could imagine an agent that can perfectly understand a high-simultaneity task and yet be unable to do more than one thing at a time. An as-yet unmet challenge is that of cross-domain comparison of problems. As a necessary first step, the CPM already considers problems within the formal structure of causal chains of CRMs. This is already quite a useful feature but its domain-agnostic nature makes it unable to assess tasks in terms of domain specialization since it has no concept of domain specialties. At best, one could imagine creating a battery of tasks that represent a specific domain, testing an agent against it, and then assigning the agent one or more “domain knowledge scores” based on its performance with the CPM. This would be analogous to the entrance and certification exams humans use to assess our specialties. While these limitations do represent drawbacks to the CPM in its current state, it is believed that the progress so far makes it useful as a composite metric for problem analysis and is enough to facilitate some initial research. It is expected that further refinements to the CPM will come with future work.

4.2.3 The CPM and Description Languages

Though important, the meta-analytical nature of the CPM may preclude it from being directly integrated into a description language. More likely, it would provide a guide for how to represent problems in the language. In this case, that would be a sequence of steps with some kind of critical path at the core of the process.

That is not to say that the difficulty metric must be completely discarded. On the contrary, it may find an important use in an agent's transpiler as the process of converting from a description language to an agent-specific language necessarily involves assessing the problem. This could be used to provide a difficulty estimate to the user or the agent. It could also be used in conjunction with the agent's experience and knowledge bases to understand whether a specific problem would be well-suited to it. Perhaps this could be used to classify EDAs by their capabilities or specialize them into different subject-matter disciplines.

4.3 Simplifying Assumptions

One challenge in the development of an EDA is that almost anything can be a design task. We commonly think of design tasks as having forms like "Build an airplane" or "Make me a table", but there is a strong case to be made for the fact that any problem can be solved by the design process. Consider how "Cross the street" could be a prompt for an agent to walk across a street *or* it could be a design challenge to create a robot capable of crossing a street. Similarly, "Vacuum the floor" could be a simple procedure for a household robot to execute *or* it could be the starting point of the design of an automated vacuuming system. The challenging part is in finding a way to encode almost all possible tasks into a structured format suitable for an EDA. Given that it would be prohibitively difficult — and unnecessary — to imbue an EDA with general or human-level intelligence, we must find a way for human engineers to pose design tasks to an EDA in a simpler form.

To do this, we make some simplifying assumptions. First, we assume that any design task will take place in an imaginary environment and not in the real world, as posited by Research Question 1 (page 5). In this imagined environment, causality still applies but time does not pass, the agent can work at any level of abstraction it desires, and there is no chance of a disturbance from an unrelated process. Second, we assume that the agent will be operating in a mostly-familiar environment. This is to say that any problem it is tasked with solving can be linked to at least some of its prior knowledge in the same way that we do not often ask human engineers to work far outside their areas of expertise. Finally, we assume that any design process can be scaffolded onto a simple and well-defined structure: the decision tree. Given that design is often a highly intentional process, we can take any solution and decompose it into the decisions that were made and how they helped the designer arrive at the solution they did. This allows us to enforce some level of order and causality upon a process that can easily get very creative and very complex. While no set of simplifying assumptions can be perfect, the goal is that these will help simplify the difficult task of design into a process that can be performed by a current-generation AI agent.

4.3.1 Design Problems are Imaginary

The first assumption we make is that design problems are worked in an imaginary environment and not in the real world. As an example, consider how a human engineer may spend much of their time working out a problem in their head, on paper, or on a computer, only ever building real-world prototypes when they need to test an idea or when they believe they have a viable solution. This shortcut is available any time the designer has sufficient knowledge about the environment in which their solution will be implemented; where prior knowledge exists, it can substitute for experimentation and discovery. Where this prior knowledge is insufficient, the agent will need to rely on experimentation and learning in order to solve its design problem. Over time, this will build up a background of experience and knowledge required to work in an imagined environment; this kind of learning is covered in detail later in Section 5.4. When it is available, this imagined environment has three key differences from the real world: time does not pass, it can exist at any level of abstraction, and disturbances are rare.

In the first regard, it is easy to see that time does not pass for an imaginary design; no drawing of an I-beam has ever rusted away over time, for example. This means that, within the imaginary world, the design agent has an effectively infinite amount of time to work (though its runtime in the real world will still be constrained for practical reasons). That said, causality is still very much a factor for a design agent. Because time will pass in the real world, the agent must account for how different elements of the design interact causally and ensure that these processes will still work in the real world. For example, one can consider a bicycle. Even though an on-paper design for a bicycle may be timeless in its nature, any designer must ensure that, when it is built in the real world, pushing down the pedals will cause the wheels to turn. Without accounting for these causal relations, the design agent may simply produce useless designs.

In the second regard, we must consider how an imaginary design can operate at any level of abstraction. The real world is always instantiated at the lowest possible level of detail. Even if we consider our actions to be as simple as "wash the dishes", for example, everything eventually comes down to molecular interactions and subatomic physics. This, however, need not be the case in our imaginary world of design. When designing a bicycle, for example, it is enough to consider the materials selected, the sizes and shapes they should be, and how they will be connected together. The lowest level of detail required is, perhaps, the materials science of how the frame will be welded together. What matters, however, is that the design agent can choose this level of abstraction. Sometimes it is enough to work at a high level and produce an approximate conceptual solution and sometimes an agent really does need to consider the subatomic physics of a problem. It may also be the case that the level of detail changes with time as a design is further refined and prepared for fabrication in the real world. Regardless, this must be accounted for in any design agent's approach to problem solving.

Finally, we can simplify the design process by removing external disturbances. These disturbances are the kinds produced by processes, systems, or agents other than our own. For example, imagine how an agent tasked with picking up a piece of paper could be interrupted by a sudden gust of wind. In the imaginary world of design, these disturbances can be neglected entirely and shifted into the design specifications. Consider that an imaginary airliner will never experience turbulence but that its design specifications will have to account for the fact that turbulence will show up in the real world. While these design specifications will still need to be addressed, they will not present as interruptions to the agent's workflow. This factor makes a design agent's job that much simpler because it can focus purely on the challenge of the design without constantly checking in on the state of the environment.

All of these are important factors in the creation of an EDA as they help determine how design problems can be represented for an AI agent. Any representation must account for the fact that designs are timeless but causal, that they may be abstracted arbitrarily, and that they exist in simplified environments. The only potential shortcoming is that working in this imagined environment first requires a substantial amount of prior knowledge that must first be gained by experience, experimentation, or pre-programmed knowledge. With this in mind, we can see that the imagined environment assumption will primarily hold after the agent has gone about an initial phase of learning and discovery. Given that many AGI-aspiring agents are designed with the intention of real-world operation, these factors can help us find a way to restructure design problems so that existing cognitive architectures can be used with little to no modification.

4.3.2 Design Problems are Familiar

Another assumption worth making is that design problems tend to be somewhat familiar. This is to say that a designer often starts with a decent amount of prior knowledge they can apply to a given design problem. The reason for making this assumption is that it emphasizes the importance of linking problems with an agent's prior knowledge, as this can significantly increase the speed at which a problem is solved. In this way, any formal representation of a design problem should include the capability to link to and reference existing knowledge. This makes the representation and decomposition problem more challenging but the benefit is much better utilization of the EDA's resources.

As a simple example, imagine you are a mechanical engineer with experience designing bicycles and you are requested to build a new bicycle to be fitted with an electric motor. Even with no experience in e-bikes, you can use your experience to know that the final design will probably be similar to a normal bicycle so you use this as a starting point. You can then do some research to learn about the specific motors, batteries, controllers, and design concepts to familiarize yourself with the new parts of the design. As you work, you link this new knowledge to what you already understand about bicycle design. After some experimentation, you finally complete the design and build a new e-bike using your newfound knowledge; you can now carry this knowledge forward and apply it the next time you are asked to create an e-bike. With an EDA, this effect may be even more profound than in a human; because of a computer's ability to store more knowledge more completely than a human brain, an EDA's experience may culminate in it one day becoming more familiar with an area of knowledge than even a human expert.

4.3.3 Design Problems are Described by Decision Trees

Given that we have already simplified design problems as working in an imaginary and familiar environment, we can see that it may not be necessary to apply a full task-theoretic problem decomposition to design problems. To do this, we exploit an existing structure that is present in almost every design problem: the decision tree. Any problem-solving done by a human or artificial designer can be represented as a decision tree where each decision and change can be explained by some reasoning. Consider the previous e-bike example: perhaps a designer would decide to go down a specific path and steer clear of a certain brand of motor. When asked why they chose the motor they did, the designer could provide a very clear reason and explain that it had the best power-to-weight ratio or perhaps the manufacturer had a good reputation for reliability. This not only adds structure to the process, it encodes a design with causality in mind; since each decision is made for a reason, we can trace the causes and effects of each part of the design. Any kind of intentional design will follow this kind of methodology; no design feature exists without a reason[1], this allows us to express a design and its features in terms of reasoning.

The trouble is that, intuitively, a decision tree lends itself well to the top-down designers discussed in Section 3.1; this would place our EDA firmly in the class of KBE systems or expert systems. Given that these systems tend to be limited in their ability to learn and deal with uncertainty, it is worth considering how an agent that learns experientially might be able to produce a similar structure. Imagine you received no formal training in any kind of fabrication skills or engineering techniques and were instead supplied with a fully-featured workshop stocked with all the best equipment and materials: how would you learn to design things? The answer is that you would

[1]There is an argument to be made that this is only true for a minimalist and purely functional design approach. While that is the case, this exploration will primarily focus on the engineering side of design for the sake of simplicity. That said, it is not outside of the realm of possibility for a design agent to be created with the capability for more aesthetic and free-form design.

experiment, you would read the minimal instruction manuals available, and then try and use the machines to see what you can create. As you experiment and solve problems, you build experience in working in this environment. While this is not the most time-efficient way of learning, it does capture knowledge reliably and in a more malleable bottom-up manner similar to the way human babies learn through experience and experimentation (Minsky, 2022). This approach also matches the way in which most AGI-aspiring agents will learn about their worlds as, rather than being provided with an extensive class all about the world and how it works, it is expected that some of the more capable agents will learn by experience using a small amount of seed code to bootstrap this process (Nivel et al., 2014b). When we approach design from this perspective, we see that it need not be a top-down application of textbook knowledge and physical laws; it can be a process of decisions made based on experience. These decisions can still be scaffolded into a design tree, they just come from a different source. In this way, an agent can learn experientially, grow its knowledge over time, and still make use of the decision tree assumption.

Altogether, simplifying the design process down to the creation of a design tree is a significant help in creating a formal representation of a design problem. Instead of being a rather vague and hard-to-conceptualize concept, a design can be represented by the series of decisions that influenced its final form. These decisions can then be expressed in terms of their causal relationships with each other and with the agent's prior knowledge, where applicable.

4.3.4 One Final Simplification

Operating under the prior assumptions and the expectation that any design agent will learn experientially, we can further simplify a design process as being made up of two major components: operations and objects (O&O). Objects are acausal entities that exist to be operated on; they are described with attributes that can be modified by operations. Operations are abstract causal models of changes that can be applied to objects. They can reshape objects, change material properties, affect fabrication techniques, and so on. Operations are facilitated by a tool, an object with the ability to be applied to another object. As an example, a screwdriver is an object that can be applied as a tool to perform an "unscrew" operation on a screw object. Since tools can be applied in different ways (for example, a screwdriver can screw, lever, or puncture), operations can be further clarified by descriptions or nested operations that ensure that a sequence of operations can be described unambiguously. With this framework of operations and objects, we can see how an agent could explore its simulated environment by applying operations to objects and learning how they are changed. As the agent learns how operations can be used, its design knowledge base grows and it becomes more capable of solving problems. Since these operations and objects will model real-world fabrication techniques and materials, the agent will eventually be capable of providing a solution to a design problem in the form of a fabrication procedure.

4.3.5 Summary

To summarize, we can take the difficult problem of problem decomposition and structure it according to a set of simplifying assumptions. First, because design occurs in an imaginary environment, we exclude the effect of time, emphasize the importance of causality, account for varying levels of abstraction, and change the way we handle external disturbances. Second, because the design agent normally works within a familiar environment, any problem representation must be able to link to existing knowledge so that the agent does not have to bootstrap every problem from scratch. Third, the difficulty of a full task theory can be reduced to a much more predictable decision tree model, which works well with the bottom-up approach used by many AGI-aspiring systems. Finally, the agent's experiential learning means that it can consider design as a sequence of operations applied to one or more objects. These assumptions point the direction toward a reasonably simple description language: one that represents problems and designs as sequences of operations on objects. A proposal for such a language is explored in detail in Section 4.5.

4.4 Problem Representation and Collaboration

A final and important part of most design or problem-solving processes is collaboration. Without collaboration and feedback, designers are at the whim of the client's first specification of the problem. While there is the possibility that the client has provided a perfect description upfront with clear constraints and no gaps in understanding, any designer will know that most real-life problem descriptions are not so well-formed. These issues can be simple questions as to specific elements of the design or they could be major oversights that prevent the project from moving forward without some extra input from the client. Smithers (1992) discusses this difficulty and considers the process of finding the perfect problem description as running in parallel to the process of finding the solution to a problem. In other words, the further one goes into a design project, the better an understanding one needs of the problem they attempt to solve. The difficulty is that this first Initial Requirements Description or IRD, as Smithers calls it, is often far too rough to allow for any significant exploration into a problem. The only solution is a feedback loop that expands on the problem description as the designer refines their solution to the problem. In short, a designer can only go so far without collaboration.

With this in mind, then, it is clear that a design agent must have some way to work with its user so that it may ask for clarification of the IRD and feedback on its progress. Since this requires some structured way to communicate the problem and solution, we can use the same description language by which problems are initially supplied to the agent. This gives us another requirement of the description language: one must be able to use it to ask questions. Asking questions is an important part of the collaborative process because both the user and the agent can get extra information out of each other. The user can ask the agent about its process and use the information to update constraints or request changes. The agent can ask the user for clarification on the design parameters or even ask for help; perhaps the agent has failed to find a solution and may require additional knowledge or experience to move forward. The trick, then, is to find a structured way to ask questions in order to prevent ambiguity on either side of the interaction.

For the purposes of this exploration, asking questions in a description language will be done by providing a description with blanks such that the other party can fill in the blanks with the necessary clarification or expansion. This is not unlike the frame-based communication implemented by Bobrow et al. (1977) in which templates of a data structure were provided with empty slots that could be filled in by user input; the same principle should work for basic design problems. Consider the following problem: you are designing a bicycle for someone but they forgot to specify whether they wanted a basket. In natural language, it is enough to ask "Do you want a basket on your bike?" but an agent without the ability to speak a natural language will require a more structured way of asking this. If the language of the agent was focused around design constraints, it could simply say to the user `basket-included: ?` as a way to specify that the user had not provided any information in the `basket-included` field. The user could then respond with `basket-included: yes` or `basket-included: no` depending on their desired design outcome so that the agent may measure its design success against a more complete set of constraints. Alternatively, the language could be specified around actions and operations so the agent could ask `add-basket: ?` and receive a similar response. Such an agent could then update its assembly procedure according to whether the user wants it to spend time adding a basket to their bike.

One further complication is that communication between the user and the design agent needs to be model-agnostic. The reason for this is experiential learning: as the agent works on designs and solves problems, it develops its own internal models of techniques and problem-solving heuristics. While this is fantastic from an efficiency perspective, it does mean that the agent's knowledge may be organized unpredictably. Even worse, its experiential learning may not include learning about any of the terms used to describe its work; it could know how to screw something in without ever learning the verbs "screw" and "unscrew", for example. This, of course, presents a difficulty when communicating with the agent as properly framing any requests within its myriad learned models would first require "untangling" its understanding of engineering, One would have to spend a significant amount of time analyzing the way the agent uses its knowledge and the way models are connected to each other before one could even begin to communicate with the agent. The solution, then, is a description language that can act as an intermediate between the agent and the human engineer. It must be simple enough for the agent to learn, yet complex enough to allow for the discussion of difficult engineering problems. The design agent must be taught this language so that it may more directly communicate with the user without the need for days of analysis to come first. Given the importance of collaboration in a design process, it is clear that any good design agent must also come with some kind of engineering description language.

4.5 Engineering Description Languages and POODL

Working through Sections 4.1-4.4, we can extract a series of requirements that any engineering description language (EDL) must satisfy. An EDL must:

- be rigorous, robust, and unambiguous
- capture problems and designs in terms of CRMs
- permit the linking of existing knowledge to the problem at hand
- provide either explicitly or implicitly an understanding of the difficulty of a problem
- be structured to fit a decision tree scaffold — it should not be unnecessarily general
- support experiential learning through interaction with an environment
- be model-agnostic — it should be able to be understood by a variety of agents without relying too heavily on any one agent's knowledge
- be useful in feedback and communication — both the user and the agent should be able to pose and answer questions

Working from these requirements, one can start to construct a very simple description language. A useful concept to center this language on is the *Operations and Objects* (O&O) paradigm from Section 4.3, in which one can describe a problem as consisting of a series of operations performed on one or more objects. This provides a direct causal basis for the language because changes in the environment can be represented as effects caused by performing certain operations. The result is a fairly natural link between interaction with an environment, designing objects, and causal reasoning. Additionally, because of their simplicity, both the operations and objects can be described in a well-structured and unambiguous way.

The EDL developed for this exploration applies this paradigm; *Prototype Operations & Objects Description Language* (POODL) was designed to prove the concept of

O&O and to provide a starting point from which to construct more complex and capable EDLs. As well as O&O, POODL is also loosely based on Replicode (Nivel and Thórisson, 2013), the language of AERA as introduced in Section 3.4. POODL started as a sort of high-level pseudocode form of Replicode that abstracted away much of the more complex and specific parts of the language. All of this means that POODL is a very simple proof-of-concept language; it does not specifically address problem descriptions or problem-solving requiring creative or innovative solutions in its current state. It is, however, complete enough to describe basic problems that should be suitable for technology demonstrations of a basic design agent.

4.5.1 Operations in POODL

The following syntax is used in POODL for an operation: `(<name> "<target>" WITH "<tool>" UNTIL "<outcome>" [<how>])`. In this structure, `<name>` is the name of the operation, `<target>` is the object the operation is affecting, `<tool>` is the object used to perform the operation, `<outcome>` is a descriptor assigned to `<target>` when the operation is complete, and `<how>` is a description of the details of the operation. It is worth noting that, while the operation may have a human-readable name, operations learned by the EDA are more likely to be anonymous. This is due to the design agent using experiential learning; it will be able to learn to perform a wide variety of tasks but will not learn terms to describe these tasks unless explicitly taught.

Additionally, while the `<target>` and `<tool>` fields can just be direct references to objects, the `<how>` field can be more detailed. Given that it is a description of the specifics of the operation, it may contain a wide variety of information with the aim of ensuring that the operation can be described accurately and unambiguously. For this exploration, the `<how>` field will simply nest operations to describe prerequisite tasks. Much like in functional programming, these operations can be nested indefinitely with the deepest operations being performed first. In other words, the actual sequence of operations is applied during the tail-recursive step. Here, we imagine a sequence of searching for a screwdriver, picking it up, and then using it to unscrew a screw as it would be represented in a POODL program:

Listing 4.1: A 3-step procedure in POODL

```
(unscrew "screw" WITH "screwdriver" UNTIL "unscrewed" [
    (pick_up "screwdriver" WITH "hand" UNTIL "held" [
        (search_for "screwdriver" WITH "flashlight" UNTIL "found" [])
    ])
])
```

For the sake of simplicity, there are a variety of limitations applied to POODL operations. Most importantly, the `<how>` field is very simplified. In theory, any level of description or specification could be used here to provide a detailed explanation of how exactly an operation is to be performed. Such freedom would allow an agent to use a tool in a novel way, for instance, it could use a screwdriver not to remove a screw but to safely discharge a capacitor. Since there was not sufficient time during this thesis to create a structured and unambiguous method the design agent could use to describe these creative uses, each `<how>` field will only be able to specify prerequisite operations as described above. As a consequence, any design agent using POODL will not be able to create its own custom tools. However, this is still possible in O&O in general. Since tools and targets are both objects, an agent should be able to use operations to modify its own tools and create something new with them. This would introduce the need for further description, though, so it is not permitted in POODL.

4.5.2 Objects in POODL

Object syntax in Prototype Operations & Objects Description Language (POODL) is extremely simple: `OBJECT "<name>" IS "<descriptor>"`. In this statement, the object `<name>` is simply described with the property `<descriptor>`. This `<descriptor>` can be any common ASCII string and may specify material properties, temperature, shape, or any other discrete quality. For example, one could describe a screwdriver as being held with `OBJECT "screwdriver" IS "held"`.

Like operations, however, this syntax also comes with some fairly significant limitations. First, objects in POODL can only take on one discrete quality at any given time. That is to say that an object could be `found` or `held` but not both. In that way, objects and operations are not unlike states and state transitions, respectively. Second, objects must be atomic, POODL does not support large complex objects or assemblies as there is no way to specify nested objects. This also means that operations cannot target the details of an object; you cannot have an operation that inserts a screw into a specific part of an object for instance. While it can be limiting, this simplicity is a somewhat unavoidable consequence of the nature of conceptual design. In conceptual design, it is not important to specify exactly what areas of what parts are going to be attached together, it is much more important to consider the parts as attached and then move on to the next part of the design. Finally, in keeping with the restrictions on operations' `<how>` fields, objects may not be described by the function since doing so would require additional explanation when using the object as a tool. Instead, the agent is to be provided with a set of hard-coded tools and a set of objects it can modify. As before, any kind of custom tool creation will require further development.

4.5.3 POODL-Extended (POODL-E)

While POODL is the language used for the design agent constructed in Chapter 6, it is worth noting that there are languages planned beyond POODL. The next version, POODL-E (short for POODL-Extended), will allow for more complete `<how>` fields in operations and more detailed descriptions of objects.

One important concept that POODL-E will apply is that of multiple independent object attributes. Instead of a set of discrete qualities, each object will be described by a fixed set of attributes, each of which can take on a distinct value. For example, objects could have attributes describing `material`, `shape`, `mass`, and so on. A benefit to this method is that object attributes could be better linked to the agent's knowledge: an object specified with `material: plastic` would automatically inherit all of the agent's knowledge on plastic materials. The EDA could also analogize what is learned from objects with similar attribute values. Selecting these attributes will be a difficult task because they must capture a very general set of objects without becoming ambiguous but it is believed to be the only way to create a more general EDL.

For the moment, however, these improvements are left for future versions. The current version of POODL is believed to be general enough to demonstrate the concept of a design agent while being simple enough to be fully developed in a reasonable time. There is still much that can be described even in the simplified structures of POODL.

Chapter 5

An Architecture for an Engineering Design Agent (EDA)

Chapter 3 established what engineering is and what agents are well-suited to it. Chapter 4 explored how problems can be decomposed and provided a language for an engineering design agent (EDA) to speak. This chapter shall explore how such an EDA could be implemented according to the principles established in these prior chapters.

The EDA architecture proposed here consists of four components: a high-level designer (HLD), low-level optimizer (LLO), a Tester, and an Output Renderer (OR). The HLD is in charge of any conceptual design the EDA does; it takes care of breaking down the problem, linking it with prior knowledge, and synthesizing an abstract solution. The LLO then takes this abstract solution and converts it into something more concrete by connecting the HLD's part specifications with generative design software. The Tester then verifies that the LLO's designs are sound by running them against industry-standard test packages. Finally, the OR takes the final design and renders it into a form convenient for humans such as drawings, schematics, assembly instructions, or whatever form is needed for the project. At each stage of the process, these components are in constant communication with each other as they give orders and ask for feedback. If the LLO is unable to make a part specification work, it will tell the HLD what parts it is having trouble with. If the HLD is having trouble with a design, it should identify the problem and ask the user for help. All of this communication is mediated by the EDA's engineering description language (EDL). This overall architecture is presented in Figure 5.1 at the end of the chapter.

However, developing each and every component to its entirety is a significant undertaking and so the scope of this chapter must be narrowed to fit the time constraints of a master's thesis. As such, the primary focus is the most complex and most novel part: the HLD. The concept for an HLD has been reasonably well-developed with the expectation that further work on the LLO, Tester, and OR will take place in future investigations. It is expected that these components will be easier to design as they should simply be a matter of integrating existing generative design systems, test packages, and scripting languages into the EDA components' modular design. In this way, the EDA will not be superseding or wholly replacing existing design automation technologies, just adding an additional layer that provides the ability to deal with more complex and unfamiliar problems.

5.1 The High-Level Designer (HLD)

The most important part of the EDA is the High-Level Designer or HLD, the component responsible for conceptual problem-solving and design; it is shown in red in Figure 5.1. Like a human engineer drawing on the back of a napkin, the HLD works its way through the problem by only defining the information that matters and abstracting away everything else. To do this, it employs large amounts of engineering knowledge, experience, and information about the physical world. Each piece of knowledge is encoded as an atomic model, an indivisible abstraction that simply represents what happens when the model is used. For example, there could be a model that specifies the exact relationship between the change in temperature of a specific metal and the length of time it is left in a furnace. Making plans and solving problems is as simple as chaining these models together until the inputs are actionable and the outputs satisfy the problem description (this is explained in detail in Section 4.1).

Acquiring models can be done in one of two ways: they can be imported as prior knowledge or they can be learned from experience. Ideally, an agent should be able to do both of these; it should be able to import existing knowledge without the need to rediscover well-established facts while avoiding being overly dependent on knowledge imports as the agent will invariably encounter a new problem not described in its imports. Doing this requires an AGI-aspiring reasoning framework like the Autocatalytic Endogenous Reflective Architecture or AERA (Nivel et al., 2013). AERA operates at the core of the HLD and so it is the center of the reasoning and learning done by the EDA; this allows the EDA to bootstrap itself with existing knowledge while also experimenting and learning as it goes. To do this, it is initialized with a seed program that starts it on the path to learning and engineering. This seed will contain enough knowledge to allow AERA to explore its world, design simple solutions to its problems, and over time, use its experiences to learn how to be a better engineer. This capability makes AERA a powerful core for the HLD because it will not only be able to reason its way to a solution but to fill in gaps in its knowledge by experimenting with new ideas; this allows it to autonomously learn and to take on problems with high levels of unknowns.

This kind of learning and experimentation cannot be done in isolation, however, as the HLD will need a simulated world in which it can run its experiments and test its ideas. To do this, it can communicate with the other components of the EDA using an engineering description language (EDL). When it is faced with a gap in its knowledge, the EDA can use the EDL to describe a test part to the Low-Level Optimizer (LLO) which can attempt a detailed design of the part based on the specifications provided. It can then pass this design on to the Tester to see how it performs. If the part works and helps the HLD solve its problem, it can mark this experiment as a success and reuse this solution in later problems. If it does not work, the HLD also gains new information and can use this to direct its efforts elsewhere. As an example, the HLD could be working through a problem that requires it to create a small hand-launched glider. Its first attempt might be to form something out of sheet metal and, while the fabrication might be possible, simulated flight tests would show that this is far too heavy to glide in the desired manner. This knowledge will inform the next generation of the design by guiding the EDA in the direction of lighter materials, where it will presumably arrive at a suitable design for a paper plane. Each generation of the HLD's experimentation and learning will be marked by this process of forming a hypothesis, testing it in a simulation, and then using this knowledge to take the next step. Finally, if the HLD is truly stuck on a problem, it can use the same EDL to reach out to the user. It can describe the issues it is having and identify why it has been unable to solve the problem in its simulations; it could even request a real world test of its proposed solution. This ability to harness available resources and learn by experimentation makes the HLD a particularly powerful part of the EDA[1].

To this author's knowledge, the HLD is a novel component. While plenty of other design systems have been studied, some even using model chaining (Žavbi and Duhovnik, 2000), none have employed a reasoning agent to work the problem conceptually. The closest possible match would be a knowledge-based engineering system but these rarely seem to have the agency to work the problem independently, the generality to be able to approach a novel problem, or the ability to increase their knowledge bases autonomously (Kügler et al., 2023). The aim is that the HLD is able to provide generality and efficiency to the underlying design processes, allowing the EDA to solve much more difficult problems than what current design systems are capable of. It will also provide a level of agency that will allow the EDA to independently break down and work problems when instructed. The HLD specified here is made up of three components: the seed program that AERA runs, the Programmable Knowledge (PK) that is used to inform its decisions, and its interactions with other components of the EDA.

[1]It is worth noting that this can even occur in the EDA's downtime. When not tasked with a problem, the EDA could be programmed to play with its environment and learn how it works. Minsky (2022) suggests that much of our common sense and knowledge of the world is developed through play so it is entirely possible that an HLD playing in a simulated engineering environment could use this to improve its engineering ability.

5.1.1 A Seed Program

A seed program is a necessary if small part of the HLD. When any AERA instance first starts up, the only knowledge it has access to is encoded in the seed. This is not just subject matter knowledge, the seed informs AERA of how it can interact with its own environment and the methods by which it kick-starts its learning process. In short, the seed program informs whether AERA succeeds or fails at its task of growing into an HLD.

The seed is programmed in Replicode (Nivel and Thórisson, 2013), the language of AERA. Replicode is a language that includes atomic building blocks such as causal-relational models, requirement models, composite states, facts, programs, and more; the goal is to find a way to represent engineering tasks within this framework. An additional challenge is that AERA, while well-equipped to both learn from and reason through problems, is not intended for the kind of detached problem-solving that occurs in engineering design. Engineering design often occurs in a simulated or imaginary world inside a computer or the head of the engineer; it is only after a proposed solution has been found or an experiment is needed that the design is actualized and tested in the real world. AERA, on the other hand, is intended for real-time operation in the constantly changing real world, it is not designed for this sort of imagined workflow. The solution is the Operations and Objects (O&O) framework as introduced in Chapter 4. By considering a solution as a series of operations on objects existing in a simulated world, we can take advantage of AERA's assumption of a real-world environment. As part of an HLD, AERA works in an abstractly simulated workshop to manipulate parts and design solutions. The results are not directly actionable as solutions, but they represent a realistic path to actualizing the solution — the same kind of path that a human engineer might consider when designing something.

For example, imagine you are assigned the task of creating a treehouse in your backyard. In a task such as this, it is essential to consider the assembly process alongside the design of the treehouse; there is no sense in designing a treehouse that cannot be constructed. So, your first thought might be about how you will build the floor and anchor the structure to the tree. Perhaps you will use a saw to cut some planks, a drill to attach them to the tree, and some bolts to hold it all in place. As you work through the design, you don't just picture the final result, you imagine yourself building the treehouse and then you optimize that final result to be as easy to build as possible. In fact, working the manufacturing process in parallel with the design process is nothing new. During the design of some of their most famous aircraft, Lockheed's Skunk Works department would have engineers work alongside production lines to quickly resolve any manufacturability issues with the design (Rich and Janos, 1996). By working through the design this way, the seed program not only makes better use of AERA's design, it ensures that any solution must be able to be built in the real world. When paired with the right seed program, a well-constructed simulated workshop should prevent the EDA from ever producing an impossible — and therefore nonviable — solution.

With this in mind, we can begin to construct an O&O seed program. In Replicode, knowledge about operations can be encoded as causal-relational models (CRMs) and composite states; the composite state tells AERA what tools and objects it needs in order to perform an operation and the CRM tells it what happens when that operation is completed. These are unified using a requirements model which ensures that a CRM can only be activated when the requirements of the composite state are met. Objects are described as simple entities in Replicode. They are not useful on their own, instead, they are used as arguments for the operation models. As objects are modified by operations, they accumulate attributes assigned with marker value statements. For example, an object `k` that was recently heated could be described as such with `(mk.val k hasattr "heated")`. The specific behavior of these attributes is implementation-specific, but the intention is to be able to describe the part in sufficient detail such that it can then be sent to the LLO for finishing and perhaps even fabrication in the real world. Finally, these behaviors are mediated by the simulated workshop environment. AERA interacts with this environment by issuing a `cmd use_tool` statements and making predictions about what it thinks will happen as a result. When the simulated workshop receives such a command, it will perform the desired action and then return the results. AERA can then check the results to see if whether they match its prediction or not. Regardless, it can use this information to update its models of the operation; AERA can learn from the process even if its actions fail or provide unexpected results. Over time, this learning should accumulate into experience and AERA should gain a better understanding of how to solve problems in the given environment.

While small, the seed program is a key part in establishing the function of the HLD. It governs the way AERA approaches problems, provides valuable starting knowledge about the subject matter, and informs AERA's interactions with its simulated environment. The success of the entire EDA largely boils down to how well the seed program is able to bootstrap AERA's knowledge and behaviors.

5.1.2 Programmable Knowledge (PK)

Programmable Knowledge (PK) comes in the form of large, pre-assembled knowledge bases that can be supplied to the EDA. Like the seed program, this is implemented in the O&O style in Replicode; each PK module is essentially a massive seed program filled with subject-specific models and composite states that the EDA can load in and draw on. For example, an EDA could be equipped with a thermodynamics module that provides a series of models allowing it to account for thermodynamic phenomena and to perform operations using this knowledge. Loading a PK module into the HLD is as simple as an `!include <PK module filename>` statement in the seed program and restarting AERA to load the new seed. Since AERA will be configured to learn cumulatively from its experience, it can pick up where from where it left off with the addition of the new PK. Just like the rest of the seed program, AERA uses PK as a starting point for its learning. As it goes, it can increase its own understanding of the PK's subject matter and better integrate it into its workflow. Theoretically, an EDA could even fill out gaps in human-generated PK by running experiments on areas it feels uncertain about.

The purpose of programmable knowledge is twofold. Most importantly, it allows an EDA to organize and access knowledge from different areas of specialization. In the previous example, an EDA could learn all about thermodynamics simply by importing the relevant PK module. The other use is in restricting and guiding the EDA's solution space. By controlling what knowledge the EDA has access to, one can influence the EDA's solution. For example, imagine you are a hobbyist aiming to design something in your garage workshop. To help you out, you could take an EDA, disable all of its advanced manufacturing knowledge, and let it run through the problem using much simpler methods. This, of course, works in tandem with the simulated workshop environment; each PK module comes with a set of tools that can be enabled or disabled depending on whether the module is available[2]. This prevents you from having to sort through solutions that require tools or processes you do not have access to. This also applies to the LLO and Tester components; if a module requires a new optimizer or simulation package, it will be annotated to include these as dependencies.

Finally, an additional benefit to PK is its consistency. One could operate a variety of different EDAs but have them all have access to the same knowledge by providing them with the same PK. This means that, for instance, no one EDA will have a better understanding of thermodynamics than any other simply because it happened to run different experiments while it was learning the subject from scratch. The difficulty is that experiential learning is a highly individual process and the models AERA builds on its own are often quite difficult to parse because of the need to work through its reasoning and figure out how it came to its conclusions. Even worse, this self-learned knowledge will form a sort of soup of models, relations, and behaviors. This means that no two EDAs will discover and structure the same knowledge in the same way if left to learn a subject on their own; imagine if human engineers were trained by dropping them into a lab with little prior knowledge and only the ability to perform experiments. The solution to this problem is standardization: human engineers go to school and EDAs get PK imports. All agents having access to the same knowledge means that managing EDAs is much simpler and more reliable.

[2]An interesting consequence of disabling a PK module but not removing its tools from the environment would be that the EDA would most likely start re-inventing the knowledge in that module. Its learning processes would guide it to experiment with the given tools even if it does not have access to the knowledge required to operate them. Over time, however, the EDA would figure out how the tools work and start to use them accordingly.

PK is the knowledge that EDAs will use to solve the problems they face. Each PK module is a large set of knowledge encoded in Replicode models using the O&O style. At any time, a PK module can be added to instantly “teach” the EDA about a new area or removed to guide the EDA’s solution in a different direction. Over time, the EDA will build on top of its PK by integrating it into the knowledge it has acquired from its experience. Altogether, PK modules make up an important part of an EDA.

5.1.3 Imaginary Design in AERA

As discussed in Research Question 1 on Page 5, the environment that design tasks are performed in is important. Designing a solution to a problem is not typically done in the real world; designs are often constrained to the engineer’s mind until a viable solution is found or an experiment needs to be performed. This means that any artificial agent attempting to emulate a human designer must also be able to work problems in its imagination before implementing them in the real world. This role is filled by AERA’s internal simulations.

As AERA learns about a situation, it creates models of its environment. These models are linked to composite states (CSTs): objects that define the environmental conditions necessary for a model to hold true. Consider a model that says that a bike will roll forwards if you press down on the pedal but only if the brake handle is released. To satisfy its goals, AERA chains these models and CSTs together to make plans. In each iteration of its thinking, AERA creates chains that work backward from its goals and forward from the current environmental conditions. It explores every eventuality and works out a plan on what its next action should be. Then, it performs the action, observes the change in the environment, and repeats the process. As an example, imagine that AERA intends to pick up a bottle. It may imagine itself moving its hand to the bottle’s position, grasping the bottle, and then picking it up off the ground. In the first iteration, it moves its hand to the bottle’s position and then checks to see what happened. In the next iteration, it reasons that it can now grasp the bottle so it does so and observes the change in the environment. Perhaps an unexpected change as occurred, such as the bottle being blown over by the wind, and AERA needs to re-evaluate the environment and make a new plan. This continues until AERA has achieved its goals.

This reasoning process is AERA’s equivalent to the human engineer’s imaginary environment. None of this needs any experimentation, it is a plan simply formed inside AERA’s thought processes and is only executed when AERA is confident that it will work. In the same way that a human engineer need not always experiment to find a solution, AERA will be able to use its prior knowledge as a shortcut to solve problems efficiently and conceptually. Of course, this assumes that AERA has already learned a sufficient amount about its environment to simulate its plan; this knowledge is both provided by the seed and built by AERA’s experiments with its environment. All this said, the plans AERA makes will only be conceptual in nature and will still need eventual validation and detailed design before being ready for the real world; these functions are performed by the EDA's other components. The benefit of AERA's imagination is not that it supplants the need for real-world experiments and detailed simulations but rather that it keeps all plans at a high level before delving into the computationally intensive environments of the LLO and Tester.

5.1.4 Interactions in an Engineering Description Language (EDL)

The final part of the HLD is its ability to communicate in an engineering description language (EDL) as suggested in Hypothesis 3 (page 7). While the particular EDL that the EDA uses is implementation-specific, this ability allows it to interact with other components outside of its simulated environment and to work with the user where needed. As described in Chapter 4, an EDL allows an EDA to abstract away some of the specifics of a problem and just describe it in terms of a series of operations and objects. This can be used to represent a procedure or, by leaving some of the fields blank, ask a question. Given that the HLD is only a part of the EDA and that realizing a design requires more than it is capable of, this communication is very important.

Communication within the EDA can theoretically cover a variety of topics but generally falls into two categories: providing a design specification and returning feedback. The first category is used when a solution is being transformed to a lower level of detail, such as when a conceptual solution is ready to be converted into a design for a real part. In this case, a statement in the EDL represents a sequence of operations that the sender is using to describe how the part is made. The receiver then takes this specification, converts it to the relevant internal representation, and then performs the requested operation on it. For example, the HLD could provide instructions on how to make a bracket part and then request that the LLO generate a 3D model of that part based on the given fabrication procedure. The second category happens after an operation has been completed and the results are ready to be returned. In the bracket example, perhaps the LLO was unable to create the specified part and needs to send this feedback to the HLD so it can adjust its design. This kind of communication will modify the original part specification to highlight the steps causing problems. The recipient is then expected to rework the design as needed to change or remove these steps. Both of these categories of communication are expected to be in constant use while an EDA is working; it should be working the problem at multiple levels of detail and using all of its available resources to achieve the best design possible.

Before this process starts, however, the EDA must be able to communicate with the user. As described in Chapter 4, the user can specify the problem in an EDL statement, which can be converted to a seed program and used to start the design process. This works well until the agent runs into a problem partway through the design or if the user desires more of an interactive design flow. In these cases, the same mechanisms used to talk between components can be used to talk to the user:

EDL statements will encode part specifications with areas highlighted or left blank to indicate problems or desired solutions, respectively. As an example, a user could work alongside an EDA to develop a new aircraft engine. The user would propose an idea for the EDA to investigate and see if it can find a process that yields a solution. If it can, it can submit it to the user for approval. If it cannot, it can ask the user for help and point out where it is having trouble. Given that EDLs are to be as human-readable as possible, this kind of interactive design should feel relatively natural and provide a clear and simple channel of communication between the user and the EDA. Perhaps future systems could go on to convert the EDL into a natural language for even more intuitive design, but this will be left for later experiments.

Finally, while any communication will be done in the EDL, there are differing ways in which this can be implemented. One method is a simple transpiler, a program that can be used to convert between Replicode and the EDL in question. While this would be acceptable for generating the initial seed program, it may prove difficult later on in the design process as the transpiler would have to sort through AERA's work and parse it into meaning. This is certainly possible, but would not be straightforward. Instead, perhaps AERA could be taught how to use the EDL as a language. As AERA has already demonstrated a capacity to learn a basic form of English (Nivel et al., 2014a), it seems that a more structured language like an EDL would be even easier. While this would represent an extra complexity either in the seed program or AERA's early training, it may be worth it in the long run because it would allow the EDA to, in essence, articulate its own reasoning rather than requiring an external program to sort through its work.

The EDL, then, is the necessary glue that holds the EDA together. With it, it can work with the user and communicate with all of its internal components. The EDL allows the HLD to realize its designs through the LLO and make adjustments when things do not go quite to plan. While not, strictly speaking, a part of the HLD, it is nevertheless an important part of its operations.

5.2 The Low-Level Optimizer (LLO)

With a potential conceptual solution found, the next step is to realize it with part designs and a solution that could actually be fabricated and built in the real world. For the sake of simplicity, this task is kept separate from the HLD; the conceptual solution informs the part design but AERA need not concern itself with the minutiae of every individual part. Instead, that is the job of the Low-Level Optimizer (LLO) which uses a separate set of generative design and optimization algorithms to turn a part specification into a 3D model; it is shown in green in Figure 5.1. Because much of this technology already exists, it was not the focus of this thesis As such, this section presents a broad overview of what an LLO might look like.

The first step of the optimization process is to generate a set of design constraints from the EDL part specification. This requires identifying key features and linking constraints across parts. For example, consider designing a pegboard for a workshop. The HLD may produce a specification for a wooden part that must be planed flat, cut to a square shape, and then have a grid of holes drilled in it with a given spacing. The optimizer could then identify that the flat surfaces, square shape, and regularly-spaced holes are key features and that everything else is up to its discretion. It could then link this to a connected part specification that asks for a small metal rod of a given diameter to be bent at two locations into a sort of S-shape, seeing that the metal rod is intended to fit into the holes, and therefore deducing the required hole size on the first part. In this way, the LLO extracts constraints in the way one might solve a set of simultaneous equations: it works out what is important, how the important variables relate to each other, and then finds an optimal solution for the remaining degrees of freedom.

The next step of the process is finding this optimum configuration. With the constraints of the part fixed, the LLO is free to change other variables to generate an optimal part. It could reduce the thickness to save weight, select a specific material to save cost, or adjust the shape to make for easier manufacturing toolpaths. These optimization parameters would all be outlined by the user in the configuration of the EDA and will depend on their goals. This could allow an aerospace company to prioritize lightness over cost or a toy designer to prioritize easy manufacturability over resistance to excessive force, for example. There is, however, the possibility that an EDA running at high autonomy could do this configuration itself. If it knows its design goals, it could set these parameters to whatever it deems most reasonable. It is important to note that the LLO will work along with the Tester to check and validate its designs. While each optimization algorithm may be tuned to its own specific optimization procedures, the Tester will provide an independent check on whether the solutions work. This makes sure that the LLO does not accidentally design a part that is unable to handle the requisite structural loads, for example.

The optimization process itself would use existing generative or value-driven design algorithms to arrive at an optimal solution. These technologies have already been extensively studied to the point of being made available to businesses and consumers (cf. Autodesk (2023)). As an example, consider the value-driven design system proposed by Vidner et al. (2021). They developed a configurator and an optimization system that could allow customers to quickly and easily generate quotes for spiral staircases, a product that is mass-produced but also needs to be customized for each application. Another example of value-driven design can be seen in the work of Castagne et al. (2009), who developed a similar system capable of optimizing components in aircraft fuselages. While the LLO would require more general solvers (such as those based on genetic algorithms), these are examples of the powerful research already being done into design optimization. It is for this reason that part design will be handled by existing optimizers rather than putting an additional load on the HLD.

An additional benefit of using existing optimizers is modularity. The EDA could be equipped with a variety of optimization programs and use each one for a different purpose; one could be for structure, one for electronics, one for aerodynamics, and so on. All that this requires is a way to inform the HLD of what optimizers are available for a given problem. This can be done with dependencies in the HLD's PK imports, any given module can simply be annotated with the optimization software recommended to handle its solutions. This keeps the overall EDA simple while also affording it a significant increase in generality.

As an even more efficient alternative, one of the LLO's modes of operation could be to search a provided parts library for parts that match the HLD's specifications. This could drastically cut down on both computational load during the design phase and project cost when the design is delivered for manufacturing. As a real-world example, one could consider the datalogger designed for Florida Atlantic University's co-prime sonar array (Schaff et al., 2021) where they were able to significantly reduce costs by using existing audio processing hardware even with a fairly unique array design. Implementing this approach in an EDA would require a parts library wherein each part is associated with specifications and causal models of their functions. This would take some time to assemble but would surely offer a faster alternative to generating a part design from scratch each time. If the search fails, it can always fall back to a custom design for the part.

When the LLO finds a solution, the EDA files this part away for the final output or to be modified at a later time if necessary. However, if no solution was found, the LLO will attempt to find out why and provide this information to the HLD. Imagine a situation in which the constraints could not be resolved because one step would require removing excessive material. In this case, the LLO would report an issue with the step that causes this issue and expect a design revision from the HLD. Alternatively, it could succeed in resolving the constraints but fail to find a solution for a part because the constraints were too harsh. In that case, it would ask the HLD to relax the constraints and highlight any in particular that are causing the problem. Regardless, the HLD should receive feedback from the LLO as to the status of the design. With this in mind, it will be able to proceed with its work.

Overall, the LLO is a conventional genetic algorithm solver with an EDL wrapper to allow it to function as part of an EDA. Where a human would set design constraints, the HLD takes their place. In a high autonomy setting, the HLD may even set the optimization priorities, too. With this tool available, the EDA should be able to not only work through a problem conceptually but to produce parts with a concrete structure that could be fabricated in the real world if desired. In short, the HLD solves problems and the LLO realizes the solutions.

5.3 The Tester

Like the LLO, the Tester (shown in light blue in Figure 5.1) is largely a set of existing software packages that are made available to the EDA for its design work. In this case, it facilitates any and all testing and simulation of the parts designed by the LLO. This is done for two reasons: validating the designs of parts for the final output and aiding the LLO by testing its ideas. In this way, the Tester behaves much like the simulation tool of any modern CAD program, it is a suite of simulation packages that allow a designer to predict how their design will function in the real world. This makes the Tester's design relatively straightforward, it accepts a representation of a part (usually a 3D model from the LLO), runs a test using standard simulation software, and then outputs the result to the relevant component.

One special feature of the Tester is its modularity. Because its design is so simple, an EDA can easily be configured with different packages of testing and simulation software. For example, an EDA focused on structural design could be equipped with an API to NASTRAN (National Aeronautics and Space Administration, 2015), a common structural analysis program used in the aerospace industry. Alternatively, an EDA working on airfoil design could have a plugin for XFOIL (Drela and Youngren, 2013), an airfoil calculator, or any computational fluid dynamics simulator. This goes for any field that such test packages exist; an EDA could design electronics, processes, software, and so on as long as it can validate its designs. An EDA could even be equipped with a variety of test packages to enable it to work on different aspects of a design simultaneously, much as a human engineer would when working on a small team. Of course, the EDA must be able to use the test packages it is provided with. For this reason, each PK module the HLD uses will list dependencies both for the LLO and the Tester such that any conceptual solutions it produces can be fully analyzed by the rest of the EDA.

Though a simple component, the Tester plays an important role in the operation of the EDA. Without it, the EDA would not be able to fully realize its designs and demonstrate their reliability. Since this occurs at such a low level of detail, it is acceptable to use industry-standard software for these purposes; the Tester simply presents an API that allows the EDA to automatically interact with them. With access to the same simulation software a human engineer would use, the Tester ensures that the EDA can be confident in its work when it comes time to hand the solution off to the user.

5.4 The Optimizer-Tester Combined Environment (OTCE)

As discussed in Research Question 1 on Page 5, the environment in which design tasks are performed is important. We often think of engineering as something that occurs in the real world since that is where all of the results end up, but the design activity itself is much more imaginary. In fact, most design tasks seem to take place in the imaginations of engineers and the simulations of computers with the only exceptions being the occasional small-scale experiment and prototype. As discussed previously, AERA's internal simulations allow it to work through a problem conceptually without the need for detailed simulations of its environment. When this is not sufficient, however, the EDA must also have a virtual environment to perform its work in; this is the purpose of the Optimizer-Tester Combined Environment (OTCE). In other words, the OTCE supports both the HLD's learning and its design efforts by creating a simulated environment for its experiments.

As its name suggests, the OTCE is a fusion of the LLO and the Tester components; this is depicted as a grey region in Figure 5.1. While the LLO and Tester are usually used to perform part generation and validation based on the HLD's conceptual solutions, they can also be used together as a test environment. The OTCE is a simulated environment that allows the EDA to perform experiments and learn about engineering without the need to be physically embodied in the real world; it is the backstop of the virtual workshop environment discussed earlier. As the EDA learns to interact with its workshop of imaginary tools and objects, the operations it attempts are simulated in the OTCE to determine their outcome. For example, if the HLD attempts to drill a hole in a piece of wood, the OTCE will run a test, see that this is possible, and return the results. If instead, the HLD tried to drill through a piece of glass without the right equipment, the OTCE would report that the glass shattered and the operation failed. While this depth of simulation does make the learning process more computationally expensive, it is believed that this is a trade-off worth making. Rather than spending time hand-coding the rules of the virtual workshop, the EDA gets to learn in a far more realistic simulation of the world. It also means that the HLD does not require perfect knowledge and the ability to simulate any problem it comes across; if it gets stuck, it can run a test in the OTCE and fill in the gaps in its knowledge. This is far more efficient than loading the HLD's seed program with massive amounts of engineering knowledge, we can just create a world for the EDA using existing generative design and testing software. Ideally, the complexity of the OTCE as an environment would even permit a level of creativity in the EDA's solutions. Consider a problem that requires inventing a new tool: the EDA could work up a design for the tool, create it in the OTCE, and use that to complete the solution. This would simply not be possible in an environment full of hand-coded rules.

The actual implementation of the OTCE is handled by the OTCE Interpreter, shown in yellow in Figure 5.1. This interpreter takes the operations provided by the HLD and converts them into design and testing jobs for the LLO and Tester, respectively. In a sense, the LLO and Tester provide the physics of the OTCE while the interpreter provides the meaning. A further function of the OTCE is the solution history, which will keep track of the EDA's solution. As the EDA works, the OTCE records its actions and builds a history of its interactions with the HLD. When a solution is found, this history can be filtered out to select only the actions that led to the solution, these can then be exported to communicate the solution to the user. Overall, the OTCE Interpreter is relatively simple, it is just an interface that provides a simulated environment for the HLD and, when needed, a record of all of its progress toward a solution.

Without an environment to design in, the EDA is powerless. When it has reached the limits of its conceptual planning, it will require a more detailed environment in which to run experiments. The OTCE's role is to provide the EDA with an easy environment for testing, validation, and learning. Even better, the OTCE does not require more than a simple interpreter to run. There is no need for a completely separate simulation solely for the EDA to experiment in; it is just a different operating mode for the LLO and Tester components. In short, the OTCE is where the EDA builds on its engineering abilities and tests its solutions to make sure they are ready for the real world.

5.5 The Output Renderer

The last component of the EDA is only used at the very end of the design process. The Output Renderer (OR) is a set of scripts that converts the EDA's designs into a format suitable for delivery to the user. In engineering, these usually take the form of schematics, drawings, assembly procedures, 3D models, and so on. Handing these to the user represents the final step of the design and the end of a long search for a solution. With these complete, the only thing remaining is actually building the design in the real world. The output renderer is shown at the bottom of Figure 5.1 in dark blue.

Like the other components, the OR is made up of many smaller components that can be added or removed to customize the EDA's capabilities. In this case, these components are simple scripts that transcribe what the EDA is working on into a standardized format for fabrication. For example, a script could exist that uses the rules of drafting to create drawings of 3D part models. A similar script could also exist to generate electrical schematics and bills of materials for a proposed circuit. This could even extend to specific fabrication techniques, perhaps the OR could apply a script that provides welding diagrams or toolpaths for computer-guided machinery. Like the other components of the EDA, the inclusion of these scripts will be controlled by the dependencies listed with each PK module; if the EDA is configured to produce a welding solution, the OR will be ready to produce a welding diagram. The scripts used to generate these exports can all be normal programs with no special attention needed to generality, intelligence, or problem-solving. These can just follow formulaic sequences to produce outputs conforming to the standards of their corresponding fields.

The OR is an important final part of the EDA. Without it, the EDA would be unable to fully complete and realize its designs. Despite this, it is a fairly simple component, requiring only pre-programmed scripts to perform its actions. Like the EDA's other components, it is modular such that the EDA can work across a wide range of subjects while always being able to fully export its work. Of all of the parts of the EDA, the OR is the only one that makes its designs truly actionable.

5.6 EDA Overall Architecture

Altogether, an EDA consists of a high-level designer, a low-level optimizer, a tester, a simulation environment, and an output renderer, most all communicating with each other and the external user through an engineering description language. An EDA starts functioning when a user provides it with a problem specification. This specification is then analyzed by the HLD, which uses its PK and internal simulations to put together an idea for a solution. If it can find a conceptual solution, it then sends specifications to the OTCE Interpreter, which coordinates testing the specified operations in a simulated environment. The LLO converts these operations into a part specification and attempts to synthesize the part with its genetic algorithms while the Tester works to make sure the part performs up to the standards of a real engineering solution. If any step of this process fails, feedback is sent up to the HLD so it can modify its design and try again. If the process succeeds, the part is committed to memory and, once the entire design is ready, is sent to the OR for export. The OR then uses its scripts to turn the part specifications and models into a more actionable form so the design can be implemented in the real world.

All of this occurs in real time with all components interacting with each other to work towards the best possible solution. If at any point the EDA fails to find a solution, feedback runs back up the chain to the point where even the user can get involved in the design process. It is even possible that the user chooses to run some real-world tests and provide feedback to the EDA so that it can better inform its design process. All of this communication is facilitated by an EDL, which specifies how exactly objects are to be created and manipulated, both in the OTCE and in the final design.

As shown in Figure 5.1, an EDA is a complex interplay of different components. Each component is optimized for a specific level of detail to maximize functionality and efficiency. The HLD does not worry about the geometry of every part, only the functions. Meanwhile, the LLO does not need to know the function of a part, it just needs to find an optimal geometry. The Tester and OR function similarly; they are auxiliaries that are only brought in to perform a function that the other components cannot. When put together in this way, these components should be able to solve engineering problems of varying complexity. To this author's knowledge, this is the first such architecture proposed for an EDA. It is hoped that this hybrid approach will allow automated systems to leverage the power of the engineering method and enable them to work alongside human engineers as partners in problem-solving.

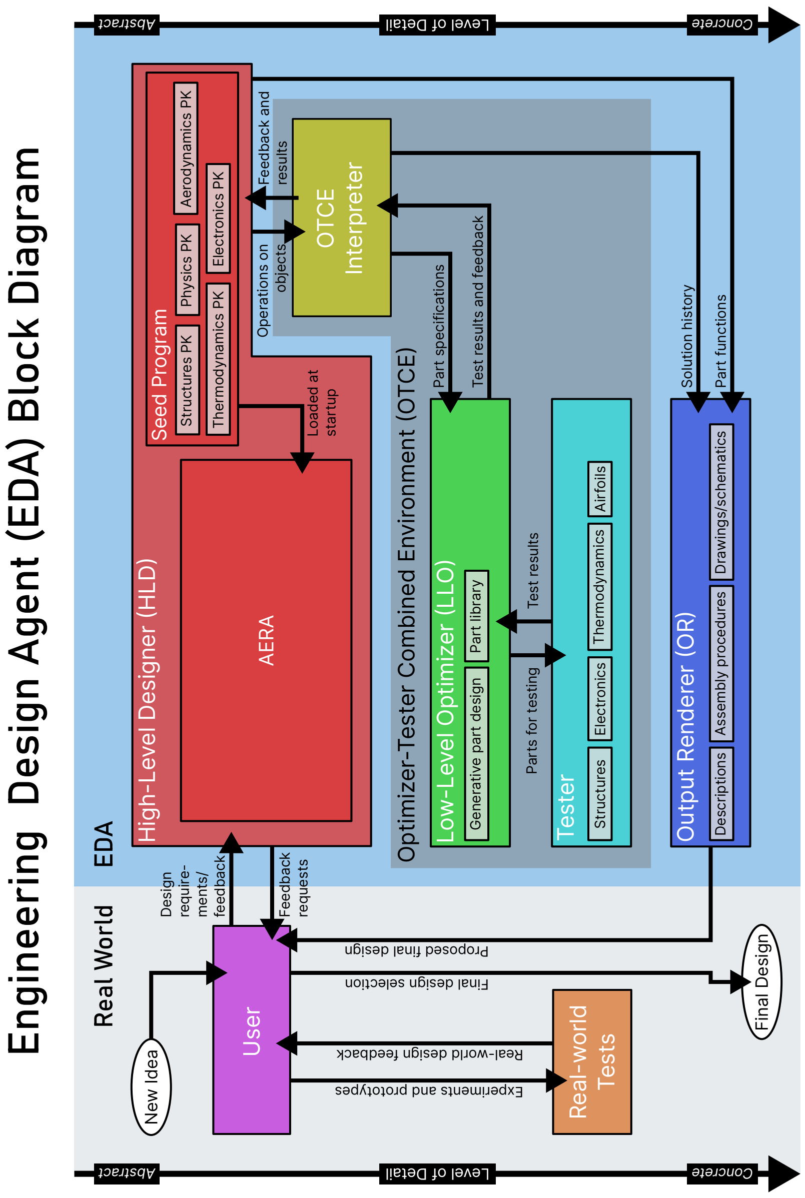


Figure 5.1: Block diagram of an engineering design agent. Note that the white blocks indicate interchangeable modules

Chapter 6

EDA-0 & the Melted Toothbrush

The last step of this investigation was to test the engineering design agent (EDA) architecture from Chapter 5. EDA-0 is a very basic EDA implementation designed to evaluate the high-level designer (HLD) and the engineering description language (EDL) components of an EDA. The HLD itself is largely implemented within the Autocatalytic Endogenous Reflective Architecture or AERA (Nivel et al., 2013) and programmed in Replicode (Nivel and Thórisson, 2013), the language of AERA. Though EDA-0's design is simple, it still demonstrates the potential of the EDA architecture and indicates the merits of further testing and development. This chapter walks through the structure of EDA-0, the problem it is tasked with solving, and then adds some remarks on its performance and possibilities for the future.

The task set for EDA-0 is based on one of the example problems for NARS, the Non-Axiomatic Reasoning System (Wang, 2004, 2013). NARS operates in a similar manner to AERA and uses a system of non-axiomatic logic to reason its way through problems. In the `toothbrush.nal` example program (Department of Computer and Information Sciences, Temple University, 2020), NARS is given the goal of unscrewing a screw. The method it is given involves melting a plastic toothbrush, reshaping it into a screwdriver shape, waiting for it to cool, and then using it to unscrew the screw. To do this, NARS is equipped with a set of operations and a corresponding set of facts that describe how these operations can be used to change the toothbrush's state. It then reasons through these operations step-by-step and finds its way to unscrewing the screw. For reference, the full original example program is reproduced in Appendix B. Since this can be considered a sort of basic design problem, it was thought that porting it to Replicode and testing it on EDA-0 would be a good first step in validating the EDA architecture.

6.1 EDA-0: An Implemented EDA

EDA-0 is a simplification of the EDA architecture described earlier, it consists solely of an HLD. For additional simplicity, EDA-0 uses hand-coded models in its seed program instead of learning from its environment. These models are based on the Operations & Objects (O&O) paradigm defined in earlier chapters. Since it is already established that AERA can learn from experimentation[1], it should be a simple matter to add this later. EDA-0 is still capable of learning from its experience, however, and will build and maintain the models that come from each design problem with which it is posed. While EDA-0 can run examples specified in the Prototype Operations & Objects Description Language (POODL), these examples must be manually converted to Replicode as EDA-0 does not implement a transpiler for simplicity's sake. Since POODL is required for the interaction between the parts of an EDA, EDA-0 cannot have a full optimizer-tester combined environment (OTCE). Instead, it must use a Replicode emulator built into its seed program. This emulated environment permits basic problem-solving and is enough to demonstrate the HLD's ability to work through problems and use the O&O method to design solutions to problems.

6.1.1 Operations & Objects Version 1 (O&O-1)

O&O-1 is a prototype version of the Operations & Objects (O&O) paradigm that includes the essential elements but leaves others for future implementation. The reason for these simplifications largely concerns ambiguity. The largest source for ambiguity is the fact that tools can be used in more than one way; a screwdriver can screw in, unscrew, lever, puncture, be magnetized, and so on. The solution to this is a "how" statement in the EDL that describes the specific way in which a tool is being used. The difficulty is in finding a standardized way to format these "how" statements. Overly detailed statements will improve clarity but trade off abstractness in the HLD's environment while statements with too little detail will make the environment more efficient but restrict the EDA's ability to creatively use tools. This is even more challenging in the event the EDA decides to create its own custom-designed tools. Similarly, there is no way to work with tools that consist of nested objects or sub-components. Without a simple way to encode the method of use into a "how" statement, the EDA will never be able to interact with its environment.

The reason a "how" statement is difficult to develop is that it must be parsed by the OTCE Interpreter to understand whether an operation is successful and what its outcome would be. Without a way to conceptualize what the EDA is asking of it, the interpreter will not be able to return useful results that the EDA can work with. Making this work requires an OTCE that encapsulates much more of the real world than has been achieved during this thesis. This level of complexity is difficult to develop

[1]See the `hand-grab-sphere-learn` examples on AERA's GitHub repository (Icelandic Institute for Intelligent Machines, 2018)

from scratch and so will likely require iteratively building more complex interpreters alongside more complex HLDs. There are important unanswered questions to do with simulation fidelity, syntax in the description language, and the nature of more complex Replicode models that will take time to solve. As it stands, there is still much to learn before it becomes possible to implement an OTCE capable of handling more creative and specific operations.

As such, O&O-1 assumes that tools can only be used one way and that any "how" statement can only go so far as listing prerequisite operations. Though simple, this approach still permits a remarkable amount of complexity. An EDA can describe and perform sequences of operations even when the inputs of one depend on the outputs of another. It can also perform operations to learn and see how different tools can be used to generate different state changes in its virtual workshop. Finally, if the environment is built the right way and it is accounted for in the OTCE, the EDA can even build its own very simple tools.

Objects are also significantly simplified in O&O-1. Each object can only be described by a single string descriptor. This descriptor string can be completely arbitrary but only really permits an object to be described in one particular state. For example, an object can be in a "heated" state or a "pliable" state but not both even if that would be the case for an object in the real world. This lends itself well to a state machine-like environment in which objects transition from state to state as they are operated on by operations.

All of this means that the EDA is not necessarily learning and working its way through a novel and creative environment like one would expect of an engineer. Despite this, there are still enough degrees of freedom to demonstrate that the EDA is still capable of reasoning about its environment, breaking down a problem, and working towards a solution.

6.1.2 Prototype Operations & Objects Description Language (POODL)

O&O-1 is implemented in the Prototype Operations & Objects Description Language or POODL, as introduced in Chapter 4. POODL is a simplified EDL that implements the simplified structure of O&O-1 and so only the most essential elements of an EDL. Most importantly, it uses a simplified "how" statement such that each operation can only be elaborated on by specifying a prerequisite operation.

Because it must be compatible with Replicode, POODL shares some of Replicode's features. First, POODL does not have a set execution order. Instead, it encodes knowledge that can be instantiated and used at any time in any order, perhaps even simultaneously. In this way, a POODL file is more similar to a database than a program. Second, POODL is functional; sequences of operations are encoded by nesting operations within each other's "how" statements with the lowest-level operation being performed first. In this way, processes are executed in the tail of the recursive step. These two elements make transpiling POODL into Replicode relatively straightforward, as will be demonstrated in the coming examples.

Objects in POODL are described using a single descriptive attribute. Given that POODL does not have a formal framework for object description, this attribute can be any arbitrary string that represents the current state of the object. For example, an object `t` could be considered "heated" simply by writing `OBJECT "t" IS "heated"`. Given how open-ended this description method is, it lends itself well to an environment made up of discrete states. In this way, an object could transition from a `"heated"` state to a `"reshaped"` state and then on to a `"cooled"` state. While simple, this can still be a powerful method when coupled with a well-built environment. When transpiling POODL to Replicode, each object should be instantiated as an entity and then given attributes using marker values. In the following example, we take the POODL statement `OBJECT "s" IS "screwed in"` and convert it to an equivalent in Replicode:

Listing 6.1: Creating an object in Replicode

```
; Create a new object "s"
s:(ent 1) [[SYNC_ONCE now 1 forever root nil]]

; Since "s" is a screw, specify that it is screwed in
(fact (mk.val s state "screwed in" 1) |[] 0s:0ms:0us GIGASEC 1 1) [[SYNC_AXIOM
     now 1 forever stdin nil]]
```

In POODL, an EDA can choose from an arbitrary set of tools and operations to perform. Operations use composite states to ensure that the tool choice agrees with the operation being performed (it is not possible to heat something up with a screwdriver, for instance). When selected, each operation applies a specific attribute to the target object. Syntactically, a POODL operation follows this structure:

Listing 6.2: An operation in POODL

```
(<name> "<target>" WITH "<tool>" UNTIL "<outcome>" [<how>])
```

Each operation uses `<tool>` on `<target>` and, when complete, `<target>` will gain the attribute `<outcome>`. Before the operation is complete, the operation specified in `<how>` must be executed. Since operations can be nested in this way, `<how>` can represent a sequence of operations. Finally, the `<name>` argument can be arbitrary and is assigned either by the EDA as it learns or by a human writing the seed program. For the sake of this example, `<name>` will be a human-readable descriptive name representing the operation.

As an example of a sequence of POODL operations, consider an agent that must whittle down a toothbrush into a screwdriver shape so it can unscrew a screw. Such a sequence would look like this:

Listing 6.3: A sequence of two operations in POODL

```
(unscrew "screw" WITH "toothbrush" UNTIL "removed" [
    (whittle "toothbrush" WITH "knife" UNTIL "screwdriver-shaped" [])
])
```

When transpiled to Replicode, each operation becomes a set of three items: a composite state (CST), a causal-relational model (CRM), and a requirements model. The CST describes what the model needs before it can run, this usually means ensuring that all of the tools are present and that the target objects are in their prerequisite states so the operation can start. The CRM then describes the operation and specifies what the outcome will be if a given tool is used on a given target. Finally, the requirements model ensures that all of the required states specified by the CST are true before the CRM can be instantiated and used. For example, the whittle operation above would be transpiled like so:

Listing 6.4: Whittling a toothbrush in Replicode

```
; WHITTLE_CST: I have a knife and a toothbrush
WHITTLE_CST:(cst [] []
    (fact (mk.val K: essence knife :) T0: T1: : :)
    (fact (mk.val T: essence toothbrush :) T0: T1: : :)
    (fact (mk.val T: state "toothbrush" :) T0: T1: : :)
|[]
|[]
[stdin] 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; WHITTLE: If we whittle a toothbrush with a knife, it will be screwdriver-
     shaped
WHITTLE:(mdl [K: T: (ti T0: T1:)] []
   (fact (cmd use_tool [K: T:] :) T2: T1_cmd: : :)
   (fact (mk.val T: state "screwdriver-shaped" 1) T1_RHS: T3: : :)
[]
   T1_RHS:(+ T0 100ms)
   T3:(+ T1 100ms)
[]
   T2:(- T1_RHS 80ms)
   T1_cmd:(- T3 100ms)
   T0:(- T1_RHS 100ms)
   T1:(- T3 100ms)
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]


; WHITTLE_req: If we satisfy WHITTLE_CST, we can use WHITTLE
WHITTLE_req:(mdl [] []
   (fact (icst WHITTLE_CST [] [K: T:] : :) T0: T1: : :)
   (fact (imdl WHITTLE [K: T: (ti T0: T1:)] [T1_RHS: T3:] : :) T0: T1: : :)
|[]
|[]
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]
```

Given how simple it is to transpile POODL and Replicode, the transpiler is not expected to be very complicated. When implemented, its greatest challenges will be linking variables and converting Replicode back to POODL. The static nature of a seed program makes for relatively easy linking, but linking during live operation would be more challenging. This would involve keeping track of a large number of objects and operations and trying to disambiguate the connections between them; this is especially challenging because AERA tends to use auto-generated names for the new models it learns. Another challenge comes from the fact that one POODL operation becomes three different Replicode objects, the transpiler would need to determine which CSTs correspond to which operations and whether some CSTs are being used for multiple operations. Neither of these challenges is unsolvable, but it will take more time before a working POODL-Replicode transpiler is ready.

6.1.3 Replicode Emulator

The final part of EDA-0 is its emulator. While, in normal operation, AERA can directly interact with its environment with no need for an emulator, EDA-0's lack of an Optimizer-Tester Combined Environment (OTCE) means that some other system must implement its environment. The emulator runs directly in the seed program provided to AERA; it is made up of a series of programs, each representing a state transition in the environment. These programs are set to trigger when AERA issues a command to use a tool. Once a program is started, it checks that the command is valid and outputs the corresponding outcome as per the models in the seed program. For example, this program watches for a whittle command, which is the result of using a knife `k` on a toothbrush `t`:

Listing 6.5: Implementing the whittle operation in the emulator

```
 pgm_cmd_whittle:(pgm [] []
(ptn (fact G:(goal (fact Command:(cmd use_tool [k t] ::) Cmd_after: Cmd_before:
    ::) ::) ::) []) 
(ptn (fact (mk.val e1 emulator_state [Unscrewed: Shaped:] :) After: Before: ::)
    [])
[]
(= (is_sim G) false)
(= Shaped false)
(< Cmd_after Before)
(> Cmd_before After)
[]
; Inject the fact that the command was executed.
(inj []
   (fact Command (+ After 20ms) Before 1 1)
   [SYNC_ONCE After 1 1 stdin nil]
)
; The toothbrush is now screwdriver-shaped
(inj [Next_state:(mk.val e1 emulator_state [Unscrewed true] 1) []])
(inj []
      (fact Next_state (+ After sampling_period) (+ Before sampling_period) 1 1)
      [SYNC_PERIODIC (+ After sampling_period) 1 1 emulator nil]
   )
   (prb [1 "print" "pgm_cmd_whittle from command:" [Command]])
1) |[]
(ipgm pgm_cmd_whittle [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]
```

It is worth noting that, in this emulator, the world is tracked using a set of states described in the variable `emulator_state`. This program represents a way to transition between two states. In particular, it transitions from the state where the toothbrush is not shaped like a screwdriver to the state where it is. With a set of these programs, one could easily create a kind of state machine environment for the EDA to experiment in. This is a particularly simplistic example, but any Replicode emulator environment will be implemented similarly.

6.2 Working an Example Problem

Listing 6.6: A new challenger approaches

```
   (unscrew "screw" WITH "?" UNTIL "unscrewed" [?])
```

This is the POODL prompt for EDA-0's example problem. Its goal was to find a way to unscrew the object `screw` until it reached the state `"unscrewed"` but the `<tool>` and the `<how>` of the process were up to EDA-0. In short, the agent is challenged to design a tool capable of unscrewing a screw. Environmentally, it has a few objects to work with. The EDA is provided with a toothbrush and a screw as well as a lighter and a screwdriver-shaped mold or form. These last two come with associated operations, the lighter can heat things up and the form can reshape them. For the sake of simplicity, the knowledge of how these tools work and how to use them is provided upfront by hand-coding models into the seed program. AERA can still learn by experience but this step removes the complexity of learning by experimentation.

The expected solution to this problem is to use the lighter to heat up the toothbrush, use the form to reshape it, wait for it to cool, and then use the now screwdriver-shaped toothbrush to unscrew the screw. This is the solution programmed in the original `toothbrush.nal` example for NARS so it seemed best to keep the same solution for the AERA equivalent. In POODL, the result we expect can be represented like so:

Listing 6.7: NARS' toothbrush example problem in POODL

```
(unscrew "screw" WITH "toothbrush" UNTIL "unscrewed" [
   (wait "toothbrush" WITH "pause" UNTIL "cooled" [
      (reshape "toothbrush" WITH "form" UNTIL "screwdriver-shaped" [
         (heat "toothbrush" WITH "lighter" UNTIL "heated" [])
```

```
            ])
        ])
    ])
```

The environment for this problem is very simple and follows the state machine style described earlier. In the seed program, the `emulator_state` variable tracks states for the screw and the toothbrush. The screw can either be screwed in or unscrewed, represented as a Boolean false or true, respectively. The toothbrush can be in one of four states: `"toothbrush"`, `"heated"`, `"cooled"`, and `"screwdriver-shaped"`. As operations are performed, it is expected to pass through these states in this order, beginning as a normal toothbrush and ending as a cooled-down, screwdriver-shaped object.

State transitions are accomplished through operations; each operation provides a way to transition from one particular state to another particular state to avoid ambiguity. Operations are triggered when AERA issues a `use_tool` command and specifies a tool and a target. When one of these is generated, a program catches the command and handles the state change accordingly. As a simpler alternative to a full-scale OTCE, all environmental states and interactions are managed in these simple emulator programs as described earlier. All of this is specified in the seed program provided to AERA at the start of the problem.

6.2.1 The Seed Program

The seed is a 425-line Replicode program. It encodes everything the agent needs to know about its goals, its environment, and how to use its tools[2]. This section will walk through the seed program and explain the specifics of its implementation. For the full program, please see Appendix C.

Listing 6.8: Setting up objects and ontologies

```
; Set up the objects and ontologies
screw:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
toothbrush:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
lighter:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
form:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
pause:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
state:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
s:(ent 1) [[SYNC_ONCE now 1 forever root nil]] ; A screw
t:(ent 1) [[SYNC_ONCE now 1 forever root nil]] ; A toothbrush
l:(ent 1) [[SYNC_ONCE now 1 forever root nil]] ; A lighter
f:(ent 1) [[SYNC_ONCE now 1 forever root nil]] ; A form
p:(ent 1) [[SYNC_ONCE now 1 forever root nil]] ; A pause (an imaginary tool that
     takes up time but doesn't do anything)
```

[2]It is worth noting that this program is essentially a heavily modified version of AERA's `hand-grab-sphere.replicode` example program (see http://www.openaera.org – *accessed Nov. 28th, 2023*). While significant modifications have been made to it, some sections still bear the marks of the original.

The first part of the program establishes the objects the EDA will be working with, this includes the tools, the targets, and some special objects. It is worth noting that, for the sake of disambiguation, each object exists as an entity of an ontology; `t` is an entity of type `toothbrush`, `s` is an entity of type `screw`, and so on. This allows us to further differentiate the objects when it comes time to use them in composite states. Among the normal objects are two special objects: a `pause` and a `state`. The `pause` is a virtual tool used to signify that the agent should wait for an environmental state transition, such as waiting for the toothbrush to cool down. This is used as a placeholder for the `use_tool` command so that AERA does not issue a command to use a tool without specifying which tool in particular. The `state` object is for internal use by the environment, it is an attribute used to describe the current states of the screw and the toothbrush.

Listing 6.9: Establishing objects' essences with facts

```
; Set some facts about the objects
s_is_a_screw:(mk.val s essence screw 1) |[]
(fact s_is_a_screw 0s:0ms:0us GIGASEC 1 1) [[SYNC_AXIOM now 1 forever stdin nil]]
t_is_a_toothbrush:(mk.val t essence toothbrush 1) |[]
(fact t_is_a_toothbrush 0s:0ms:0us GIGASEC 1 1) [[SYNC_AXIOM now 1 forever stdin
     nil]]
l_is_a_lighter:(mk.val l essence lighter 1) |[]
(fact l_is_a_lighter 0s:0ms:0us GIGASEC 1 1) [[SYNC_AXIOM now 1 forever stdin nil
     ]]
f_is_a_form:(mk.val f essence form 1) |[]
(fact f_is_a_form 0s:0ms:0us GIGASEC 1 1) [[SYNC_AXIOM now 1 forever stdin nil]]
p_is_a_pause:(mk.val p essence pause 1) |[]
(fact p_is_a_pause 0s:0ms:0us GIGASEC 1 1) [[SYNC_AXIOM now 1 forever stdin nil]]
```

The next part establishes some facts about the objects; these facts confirm that each entity defined in the previous section belongs to its corresponding ontology. This is done with the `essence` keyword, a built-in attribute that allows AERA to state that a specific object belongs to a more general category. This means that CSTs can match many different objects so long as they are of the right type; a CST looking for an object of essence `screw` will work for `s` or any other object so long as it is a `screw`, These attributes are assigned using marker values (the `mk.val` lines) which are then propagated by `fact`s. These facts are set to be valid from startup to `GIGASEC`, a macro representing a time one billion seconds (approximately 30 years) into the future, to ensure that these facts will always be true to AERA. While the details of the Replicode implementation are not important it is worth discussing some of these features as they will be relevant in future sections.

Listing 6.10: Composite states as prerequisites for models

```
; S3: I have a screw and a cooled toothbrush
S3:(cst [] []
   (fact (mk.val T: essence toothbrush :) T0: T1: : :) ; There exists a toothbrush
   (fact (mk.val T: state "cooled" :) T0: T1: : :) ; that is cooled
   (fact (mk.val S: essence screw :) T0: T1: : :) ; and there exists a screw
   (fact (mk.val S: state "screwed in" :) T0: T1: : :) ; that is screwed in
|[]
|[]
[stdin] 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; S2: I have a toothbrush that’s screwdriver-shaped
S2:(cst [] []
   (fact (mk.val P: essence pause :) T0: T1: : :) ; The tool we’re using is a pause
   (fact (mk.val T: essence toothbrush :) T0: T1: : :) ; and there exists a
        toothbrush
   (fact (mk.val T: state "screwdriver-shaped" :) T0: T1: : :) ; that is
        screwdriver-shaped
|[]
|[]
[stdin] 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; S1: I have a toothbrush that’s heated
S1:(cst [] []
   (fact (mk.val F: essence form :) T0: T1: : :) ; There exists a form
   (fact (mk.val T: essence toothbrush :) T0: T1: : :) ; and there exists a
        toothbrush
   (fact (mk.val T: state "heated" :) T0: T1: : :) ; that is heated
|[]
|[]
[stdin] 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; S0: I have a toothbrush that’s shaped like a toothbrush
S0:(cst [] []
   (fact (mk.val L: essence lighter :) T0: T1: : :) ; There exists a lighter
   (fact (mk.val T: essence toothbrush :) T0: T1: : :) ; and there exists a
        toothbrush
   (fact (mk.val T: state "toothbrush" :) T0: T1: : :) ; that looks like a
        toothbrush
|[]
|[]
[stdin] 1) [[SYNC_ONCE now 0 forever primary nil 1]]
```

These composite states (CSTs) represent environmental conditions that must be true before a model can be used. When AERA notices that all of the conditions of a CST are met, it creates an instantiated CST or `icst`. An `icst` can then trigger the instantiation of a model, in effect suggesting a course of action to AERA. Each CST is essentially a list of `fact` messages that must occur within a given time interval, `T0` to `T1` in this case. For instance, `S0` is a composite state that checks if an object `L` of the type `lighter` is available, that an object `T` of the type `toothbrush` is available, and that `T` currently has the state `"toothbrush"`. When all of these conditions are met simultaneously, the CST will be instantiated and become an `icst`. To ensure that conditions like these fall within the right time window, `essence` and `state` facts are injected periodically by the emulator environment.

Listing 6.11: EDA-0's models of its environment and objects

```
; UNSCREW: If we use a cooled toothbrush on a screw, it will become unscrewed
UNSCREW:(mdl [T: S: (ti T0: T1:)] []
   (fact (cmd use_tool [T: S:] :) T2: T1_cmd: : :)
   (fact (mk.val S: state "unscrewed" 1) T1_RHS: T3: : :)
[]
   T1_RHS:(+ T0 100ms)
   T3:(+ T1 100ms)
[]
   T2:(- T1_RHS 80ms)
   T1_cmd:(- T3 100ms)
   T0:(- T1_RHS 100ms)
   T1:(- T3 100ms)
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; UNSCREW_req: We can use UNSCREW if we satisfy S3
UNSCREW_req:(mdl [] []
   (fact (icst S3 [] [T: S:] : :) T0: T1: : :)
   (fact (imdl UNSCREW [T: S: (ti T0: T1:)] [T1_RHS: T3:] : :) T0: T1: : :)
|[]
|[]
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]


; WAIT: If we use wait (use nothing) on a screwdriver-shaped toothbrush, it will
    become cooled
WAIT:(mdl [P: T: (ti T0: T1:)] []
   (fact (cmd use_tool [P: T:] :) T2: T1_cmd: : :)
   (fact (mk.val T: state "cooled" 1) T1_RHS: T3: : :)
[]
   T1_RHS:(+ T0 100ms)
   T3:(+ T1 100ms)
[]
   T2:(- T1_RHS 80ms)
   T1_cmd:(- T3 100ms)
   T0:(- T1_RHS 100ms)
   T1:(- T3 100ms)
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; WAIT_req: We can use WAIT if we satisfy S2
WAIT_req:(mdl [] []
   (fact (icst S2 [] [P: T:] : :) T0: T1: : :)
   (fact (imdl WAIT [P: T: (ti T0: T1:)] [T1_RHS: T3:] : :) T0: T1: : :)
|[]
|[]
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]


; FORM: If we use a form on a heated toothbrush, it will become screwdriver-shaped
FORM:(mdl [F: T: (ti T0: T1:)] []
    (fact (cmd use_tool [F: T:] :) T2: T1_cmd: : :)
    (fact (mk.val T: state "screwdriver-shaped" 1) T1_RHS: T3: : :)
[]
    T1_RHS:(+ T0 100ms)
    T3:(+ T1 100ms)
[]
    T2:(- T1_RHS 80ms)
    T1_cmd:(- T3 100ms)
    T0:(- T1_RHS 100ms)
    T1:(- T3 100ms)
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; FORM_req: We can use FORM if we satisfy S1
FORM_req:(mdl [] []
    (fact (icst S1 [] [F: T:] : :) T0: T1: : :)
    (fact (imdl FORM [F: T: (ti T0: T1:)] [T1_RHS: T3:] : :) T0: T1: : :)
|[]
|[]
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]


; HEAT: If we use a lighter on a toothbrush, it will become heated
HEAT:(mdl [L: T: (ti T0: T1:)] []
    (fact (cmd use_tool [L: T:] :) T2: T1_cmd: : :)
    (fact (mk.val T: state "heated" 1) T1_RHS: T3: : :)
[]
    T1_RHS:(+ T0 100ms)
    T3:(+ T1 100ms)
[]
    T2:(- T1_RHS 80ms)
    T1_cmd:(- T3 100ms)
    T0:(- T1_RHS 100ms)
    T1:(- T3 100ms)
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; HEAT_req: We can use HEAT if we satisfy S0
HEAT_req:(mdl [] []
    (fact (icst S0 [] [L: T:] : :) T0: T1: : :)
    (fact (imdl HEAT [L: T: (ti T0: T1:)] [T1_RHS: T3:] : :) T0: T1: : :)
|[]
|[]
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]
```

The other half of AERA's knowledge is encoded in a series of models. Each operation in the POODL solution corresponds to two models, a CRM and a requirements model. The CRM indicates the causal relations between operations; given a certain condition, it will describe the result. As an example, consider the `HEAT` model. As a precondition, it looks for a `use_tool` command to be issued with the arguments `L: T: (ti T0: T1:)` representing a lighter `L`, a toothbrush `T`, and a time interval from `T0` to `T1`. As a postcondition, it describes how the object `T` will be assigned the state `"heated"`.

The remaining calculations are a simple matter of making sure that these two events occur at the right times. The requirements model for `HEAT`, `HEAT_req`, then specifies the conditions that must be true for `HEAT` to be used. Specifically, `HEAT_req` needs `S0` to be instantiated before `HEAT` can itself be instantiated and made available. Given that `S0` corresponds to the case where a lighter and a normal toothbrush are available, we can see that `HEAT` will only be instantiated at the very beginning of the test when there is the opportunity to heat the toothbrush up.

One notable feature of these models is that they can be used to reason both forward and backward in time. In the forward direction, the `HEAT` model describes how a toothbrush can be heated up when a set of conditions are met. In the backward direction, the same model says that we must first meet a set of conditions to have a heated toothbrush. This even works across multiple models; AERA can start from its goals and work backward to make a plan to determine what operations should be performed in what order. This is a core part of AERA's reasoning and so was an important factor when selecting AERA as the agent for this EDA architecture.

Listing 6.12: Setting up emulator state and emulator group

```
; Set up the emulator state and inject it
; emulator_state: [Is the screw unscrewed?, What is the state of the toothbrush?]
e1:(ent 1) [[SYNC_ONCE now 1 forever root nil]]
emulator_state:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
emulator:(std_grp 2 0 0 0 []) [[SYNC_ONCE now 0 forever root nil COV_OFF 0]]

; This program runs in the primary group to re-inject any (fact (goal (fact (cmd
    ::)))) into emulator.
pgm_inject_in_emulator_group:(pgm [] []
   (ptn f_G:(fact G:(goal (fact (cmd ::) ::) ::) ::) [])
[]
   ; Only re-inject non-simulation goals.
   (= (is_sim G) false)
[]
   (inj [f_G [SYNC_ONCE (now) 1 1 emulator nil]])
1) |[]
(ipgm pgm_inject_in_emulator_group [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever primary nil 1]
```

The next part of the seed defines the emulator state and instantiates a program that will continuously re-inject goals while AERA works; this is a core part of any Replicode program. Notice how the emulator state is also defined as an object in an ontology. Its initial value is not given yet, but it is important to set up. The goal injection program looks for any `fact` concerning AERA's goal and injects it into the emulator group so that the emulated environment is copied on AERA's current goal. Most of this code is drawn directly from the original `hand-grab-sphere.replicode` example program.

Listing 6.13: Drive and goal injection

```
!def DRIVE_START 300ms
start:(pgm [] [] [] []
   (inj [State:(mk.val e1 emulator_state [false "toothbrush"] 1) []])
   (inj []
      (fact State After:(now) (+ After sampling_period) 1 1)
      [SYNC_PERIODIC now 1 1 emulator nil]
   )
1) |[]
(ipgm start [] RUN_ONCE sampling_period VOLATILE NOTIFY 1) [[SYNC_ONCE now 0
     forever stdin nil 1]]

m_drive:(mdl [] []
   ; The goal target timings are the same as the drive timings.
   (fact (mk.val s state "unscrewed" 1) T0: T1: 1 1)
   (fact run T0: T1: ::)
|[]
|[]
[stdin drives] 1 1 1 0 1) [[SYNC_ONCE now 0 forever primary nil 1]]

pgm_inject_drive:(pgm [] []
   ; This fact repeats periodically. We use it as a "heartbeat".
   (ptn (fact (mk.val s essence screw :) After: Before: ::) [])
[]
   (>= After (+ this.vw.ijt DRIVE_START))
[]
   ; The end of the time interval will be used in m_drive as the end of the goal
        interval.
   (inj [f_run:(fact run After (+ Before 500ms) 1 1) []])
   (inj [G:(goal f_run self nil 1) []])
   (inj []
      ; Delay a little to allow predictions for this sampling period before
           injecting the drive.
      (fact G T0:(+ After 10ms) T0 1 1)
      [SYNC_ONCE T0 1 forever primary nil]
   )
   (prb [1 "print" "injected drive" []])
1) |[]
(ipgm pgm_inject_drive [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever primary nil 1]

; Before DRIVE_START, just re-inject the same values.
pgm_before_drive:(pgm [] []
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: Shaped:] :) After: Before: ::)
        [])
[]
   (< After (+ this.vw.ijt DRIVE_START))
[]
   ; All state variables keep the same value.
   (inj [Next_state:(mk.val e1 emulator_state [Unscrewed Shaped] 1) []])
   (inj []
      (fact Next_state (+ After sampling_period) (+ Before sampling_period) 1 1)
      [SYNC_PERIODIC (+ After sampling_period) 1 1 emulator nil]
   )
1) |[]
(ipgm pgm_before_drive [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]
```

An important part of the seed program is drive handling and injection. Drive models like `m_drive` ensure that AERA stays on task and continues to follow its goal. In this case, the goal is for the screw `s` to reach the state `"unscrewed"`. Given that certain essence facts are repeatedly injected, the drive injection program can use these as a trigger to periodically inject the drive. This way, each cycle of AERA's reasoning begins with an awareness of its environment and goals. For the first 300 ms, AERA is not informed of its drive and is instead just presented with conditions in its environment. This allows it to recognize the first CSTs and begin to form a conception of the conditions around it. Finally, note that `emulator_state` is given a starting value at the very top of the snippet in the `start` program. This problem starts off with the screw being unscrewed or `false` and the toothbrush being in an unmodified state, given by the index `0`.

Listing 6.14: Interactions with the emulator

```
; Unscrew something with a screwdriver-shaped object
pgm_cmd_unscrew:(pgm [] []
   (ptn (fact G:(goal (fact Command:(cmd use_tool [t s] ::) Cmd_after: Cmd_before:
        ::) ::) ::) [])
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: State:] :) After: Before: ::)
        [])
[]
   (= (is_sim G) false)
   (= Unscrewed false)
   (= State "cooled")
   (< Cmd_after Before)
   (> Cmd_before After)
[]
   ; Inject the fact that the command was executed.
   (inj []
      (fact Command (+ After 20ms) Before 1 1)
      [SYNC_ONCE After 1 1 stdin nil]
   )
   ; The screw is now unscrewed
   (inj [Next_state:(mk.val e1 emulator_state [true "cooled"] 1) []])
   (inj []
      (fact Next_state (+ After sampling_period) (+ Before sampling_period) 1 1)
      [SYNC_PERIODIC (+ After sampling_period) 1 1 emulator nil]
   )
   (prb [1 "print" "pgm_cmd_unscrew from command:" [Command]])
1) |[]
(ipgm pgm_cmd_unscrew [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]
```

```
pgm_cmd_wait:(pgm [] []
   (ptn (fact G:(goal (fact Command:(cmd use_tool [p t] ::) Cmd_after: Cmd_before:
        ::) ::) ::) [])
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: State:] :) After: Before: ::)
        [])
[]
   (= (is_sim G) false)
   (= State "screwdriver-shaped")
   (< Cmd_after Before)
   (> Cmd_before After)
[]
   ; Inject the fact that the command was executed.
   (inj []
      (fact Command (+ After 20ms) Before 1 1)
      [SYNC_ONCE After 1 1 stdin nil]
   )
   ; The screwdriver is now cooled
   (inj [Next_state:(mk.val e1 emulator_state [Unscrewed "cooled"] 1) []])
   (inj []
      (fact Next_state (+ After sampling_period) (+ Before sampling_period) 1 1)
      [SYNC_PERIODIC (+ After sampling_period) 1 1 emulator nil]
   )
   (prb [1 "print" "pgm_cmd_wait from command:" [Command]])
1) |[]
(ipgm pgm_cmd_wait [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]

pgm_cmd_form:(pgm [] []
   (ptn (fact G:(goal (fact Command:(cmd use_tool [f t] ::) Cmd_after: Cmd_before:
        ::) ::) ::) [])
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: State:] :) After: Before: ::)
        [])
[]
   (= (is_sim G) false)
   (= State "heated")
   (< Cmd_after Before)
   (> Cmd_before After)
[]
   ; Inject the fact that the command was executed.
   (inj []
      (fact Command (+ After 20ms) Before 1 1)
      [SYNC_ONCE After 1 1 stdin nil]
   )
   ; The toothbrush is now screwdriver-shaped
   (inj [Next_state:(mk.val e1 emulator_state [Unscrewed "screwdriver-shaped"] 1)
        []])
   (inj []
      (fact Next_state (+ After sampling_period) (+ Before sampling_period) 1 1)
      [SYNC_PERIODIC (+ After sampling_period) 1 1 emulator nil]
   )
   (prb [1 "print" "pgm_cmd_form from command:" [Command]])
1) |[]
(ipgm pgm_cmd_form [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]

pgm_cmd_heat:(pgm [] []
   (ptn (fact G:(goal (fact Command:(cmd use_tool [l t] ::) Cmd_after: Cmd_before:
        ::) ::) ::) [])
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: State:] :) After: Before: ::)
        [])
[]
   (= (is_sim G) false)
   (= State "toothbrush") ; A little unnecessary, we can always heat and re-form
   (< Cmd_after Before)
   (> Cmd_before After)
[]
   ; Inject the fact that the command was executed.
   (inj []
      (fact Command (+ After 20ms) Before 1 1)
      [SYNC_ONCE After 1 1 stdin nil]
   )
   ; The toothbrush is now heated
   (inj [Next_state:(mk.val e1 emulator_state [Unscrewed "heated"] 1) []])
   (inj []
      (fact Next_state (+ After sampling_period) (+ Before sampling_period) 1 1)
      [SYNC_PERIODIC (+ After sampling_period) 1 1 emulator nil]
   )
   (prb [1 "print" "pgm_cmd_heat from command:" [Command]])
1) |[]
(ipgm pgm_cmd_heat [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]
```

After the drives are set up, all that remains is the emulator. The programs listed here watch for `use_tool` commands, check their validity, and then process them accordingly. For example, the `pgm_cmd_heat` program watches for AERA to make it a goal to `use_tool` with objects `l`, the lighter, and `t`, the toothbrush. It then makes sure that the goal is not simulated, that the toothbrush is in its initial state of `0`, and that the timeline of the command and action line up. Then, it issues a `fact` that the command was executed — which is useful for AERA's planning — before issuing the updated state of the toothbrush and logging the event to the console. At the end of the program, the toothbrush will have transitioned from state `0` to state `1`; it will have gone from being the `"toothbrush"` state to the `"heated"` state.

Before performing these actions, it is important to ensure that AERA actually intends to issue the command. If AERA had been working on a plan and issuing simulated goals, this program should not respond and should wait for AERA to finish its plan and decide on a real action to take. Each program operates in the same way and each corresponds to one possible state transition in the emulator's environment.

Listing 6.15: Keeping AERA up to date on emulator status

```
; Reinject that the screw is still screwed in (if that's the case)
pgm_inject_mk_vals_screwedin:(pgm [] []
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: Shaped:] :) After: Before: ::)
        [])
[]
   (= Unscrewed false)
[]
   (inj [Val_state:(mk.val s state "screwed in" 1) []])
   (inj []
      (fact Val_state After Before 1 1)
      [SYNC_PERIODIC After 1 1 stdin nil]
   )
   (prb [1 "print" "s is screwed in" []])
1) |[]
(ipgm pgm_inject_mk_vals_screwedin [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]

; Reinject that the screw is unscrewed (if that's the case)
pgm_inject_mk_vals_unscrewed:(pgm [] []
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: Shaped:] :) After: Before: ::)
        [])
[]
   (= Unscrewed true)
[]
   (inj [Val_state:(mk.val s state "unscrewed" 1) []])
   (inj []
      (fact Val_state After Before 1 1)
      [SYNC_PERIODIC After 1 1 stdin nil]
   )
   (prb [1 "print" "s is unscrewed" []])
1) |[]
(ipgm pgm_inject_mk_vals_unscrewed [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]

; Reinject the current state of the toothbrush
pgm_inject_mk_vals_toothbrush:(pgm [] []
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: State:] :) After: Before: ::)
        [])
|[]
[]
   (inj [Val_state:(mk.val t state State 1) []])
   (inj []
      (fact Val_state After Before 1 1)
      [SYNC_PERIODIC After 1 1 stdin nil]
   )
1) |[]
(ipgm pgm_inject_mk_vals_toothbrush [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]
```

The final part of the emulator simply takes the current state and periodically re-injects it to make sure it is visible to AERA. The reason for this is that most of the goings-on of the emulator are kept in the `emulator` group where they will not interfere with AERA's inputs in the `stdin` group. When the emulator has finished updating the environment, these programs "publish" the updated states to the `stdin` group. The first two programs handle state changes in the `screw` object. If the screw is still screwed in, the first program re-injects this fact, if not, the second program responds. The third program simply takes the current state of the toothbrush from the `emulator` group and injects it back into the `stdin` group. This continuous re-injection ensures that the current state of the environment is captured each time AERA goes to reason about its goals and make a plan.

Put together, this seed program describes the entirety of the toothbrush problem. It begins by telling AERA what objects are available and providing some knowledge about how they can be used. It then sets up AERA's goals and a procedure for how and when they will be injected into AERA's reasoning. Finally, it establishes a simple emulator that allows AERA to see and interact with the simulated environment of this example program. It is worth noting that this is a very simple example with a restrictive state machine-like environment. Future agents will require much more involved knowledge and environments, hence the existence of separate knowledge modules and the OTCE, respectively.

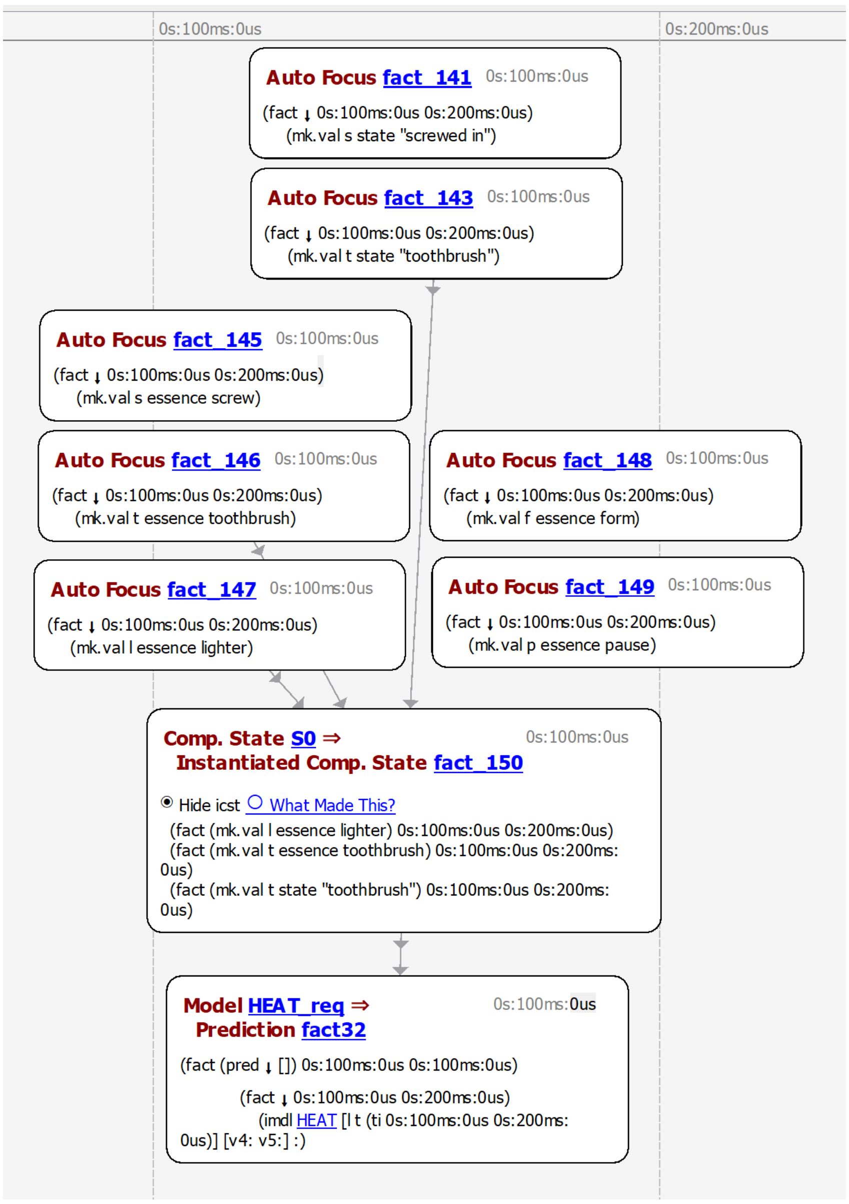
6.2.2 EDA-0's Performance

Running AERA on the toothbrush problem only takes a second or two. The resulting reasoning trace shows that AERA can understand and work the problem. For convenience, these results are illustrated in the AERA Visualizer[3].

Figure 6.1 shows the initial state of the environment and AERA's assessment of it. At the top, we see that the screw `s` has the state `"screwed in"` and that the toothbrush `t` has the state `"toothbrush"`. Further down, the essence facts are visible, indicating to AERA what kind of object each entity is. Also visible is an instantiated composite state, AERA has already seen that all of the conditions of `S0` are present and has instantiated that CST. From this, AERA makes the prediction that, given the instantiation of `S0`, the `HEAT_req` model will be available for use.

After a few hundred milliseconds, AERA's drive is injected and it can start to form plans. This is visible in Figure 6.2, where AERA is chaining backward from its goal. We can see that, in order for it to get `s` to the `"unscrewed"` state, it must first instantiate the `UNSCREW` model. Before it can do that, it must satisfy the model's CST which specifies that it needs an object of the `toothbrush` ontology that has the state `"cooled"` and a `screw` object that is `"screwed in"`. It assigns the first object the variable `v0` and surmises that `v0` must be the toothbrush `t`.

[3]The raw outputs from AERA would be a few hundred pages long so they are not included in this thesis. Instead, they are available upon request.



```
0s:100ms:0us                                   0s:200ms:0us

Auto Focus fact_141   0s:100ms:0us
(fact ↓ 0s:100ms:0us 0s:200ms:0us)
   (mk.val s state "screwed in")

Auto Focus fact_143   0s:100ms:0us
(fact ↓ 0s:100ms:0us 0s:200ms:0us)
   (mk.val t state "toothbrush")

Auto Focus fact_145   0s:100ms:0us
(fact ↓ 0s:100ms:0us 0s:200ms:0us)
   (mk.val s essence screw)

Auto Focus fact_146   0s:100ms:0us
(fact ↓ 0s:100ms:0us 0s:200ms:0us)
   (mk.val t essence toothbrush)

Auto Focus fact_148   0s:100ms:0us
(fact ↓ 0s:100ms:0us 0s:200ms:0us)
   (mk.val f essence form)

Auto Focus fact_147   0s:100ms:0us
(fact ↓ 0s:100ms:0us 0s:200ms:0us)
   (mk.val l essence lighter)

Auto Focus fact_149   0s:100ms:0us
(fact ↓ 0s:100ms:0us 0s:200ms:0us)
   (mk.val p essence pause)

Comp. State S0 ⇒                              0s:100ms:0us
  Instantiated Comp. State fact_150

◉ Hide icst ○ What Made This?
  (fact (mk.val l essence lighter) 0s:100ms:0us 0s:200ms:0us)
  (fact (mk.val t essence toothbrush) 0s:100ms:0us 0s:200ms:
0us)
  (fact (mk.val t state "toothbrush") 0s:100ms:0us 0s:200ms:
0us)

Model HEAT_req ⇒                              0s:100ms:0us
  Prediction fact32

(fact (pred ↓ []) 0s:100ms:0us 0s:100ms:0us)

      (fact ↓ 0s:100ms:0us 0s:200ms:0us)
        (imdl HEAT [l t (ti 0s:100ms:0us 0s:200ms:
0us)] [v4: v5:] :)
```

Figure 6.1: The toothbrush problem's initial state

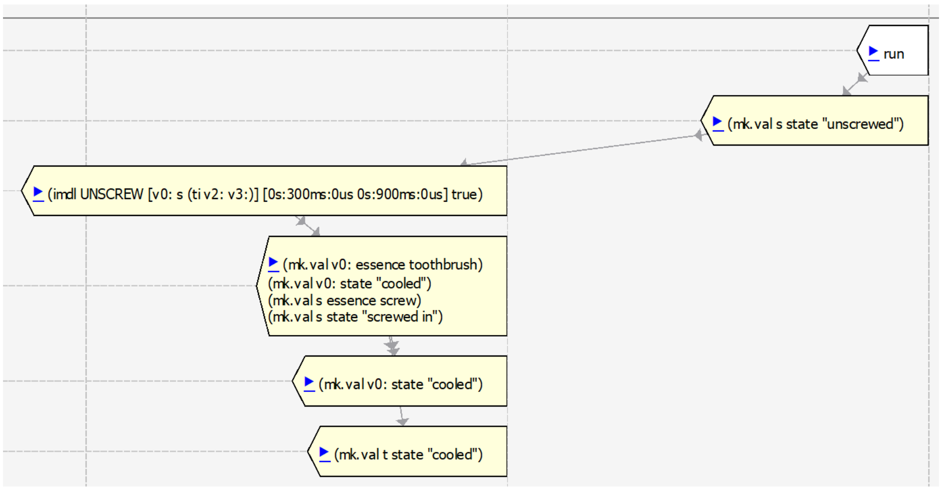
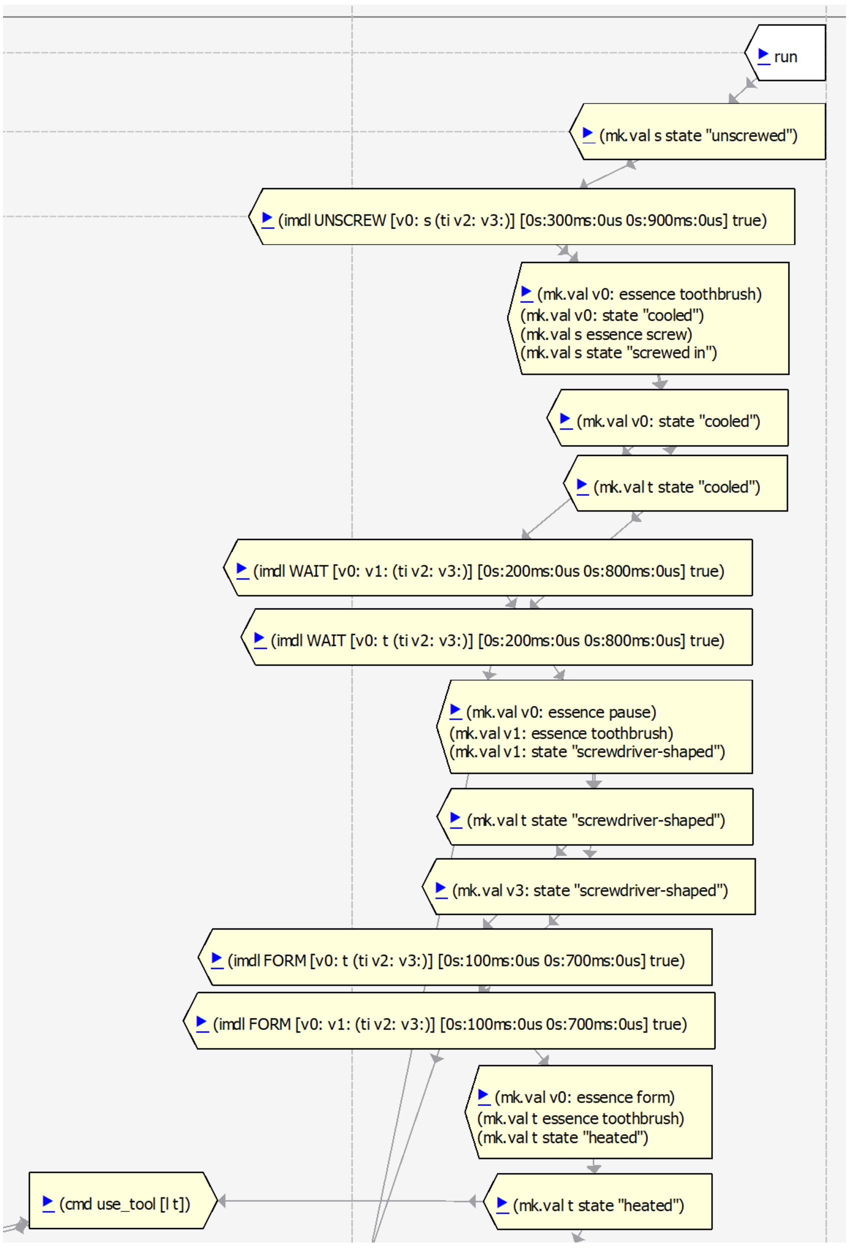

run
(mk.val s state "unscrewed")
(imdl UNSCREW [v0: s (ti v2: v3:)] [0s:300ms:0us 0s:900ms:0us] true)
(mk.val v0: essence toothbrush)
(mk.val v0: state "cooled")
(mk.val s essence screw)
(mk.val s state "screwed in")
(mk.val v0: state "cooled")
(mk.val t state "cooled")

Figure 6.2: AERA's first reasoning steps in the toothbrush problem.

AERA then continues this process, chaining models and composite states all the way back until it finds an available action; this can be seen in Figure 6.3. After determining that it needs to use the `UNSCREW` model, AERA sees that it must first use the `WAIT` model for the toothbrush to achieve the `"cooled"` state. But, to get there, the toothbrush must be `"screwdriver-shaped"` so AERA will need the `FORM` model before the `WAIT` model. Finally, given that the `FORM` model requires that the toothbrush first be `"heated"`, AERA sees that its first step must be a `use_tool` command to heat up the toothbrush `t` with the lighter `l`. It is worth noting that each of these models represents an operation; each of these steps is an interaction with the environment to use a tool to modify an object.

Since this command is actionable and requires no further prerequisite conditions, AERA simulates executing it and works its way forward to check its plan. Figure 6.4 shows AERA simulating its way through its plan. It is worth noting that none of this simulation affects the current environmental state, all of this is effectively occurring in AERA's imagination. The first step it considers is using the lighter `l` to get `t` to its `"heated"` state. Then, the `FORM` model shows that it can use the form `f` to get `t` to be `"screwdriver-shaped"`. Next, it sees that it can use `p` and the `WAIT` model until `t` has reached the `"cooled"` state. Finally, it uses the `UNSCREW` model to predict that it will be able to use `t` as a screwdriver to unscrew `s` and achieve its goal.

With AERA's plan finally complete, it takes the first step issuing a `use_tool` command, thereby instantiating the `HEAT` model as shown in Figure 6.5. Behind the scenes, this command is caught by the emulator and executed as a state transition in the environment. Figure 6.6 shows the result: `t` has changed its state to `"heated"` and the



```
run
(mk.val s state "unscrewed")
(imdl UNSCREW [v0: s (ti v2: v3:)] [0s:300ms:0us 0s:900ms:0us] true)
(mk.val v0: essence toothbrush)
(mk.val v0: state "cooled")
(mk.val s essence screw)
(mk.val s state "screwed in")
(mk.val v0: state "cooled")
(mk.val t state "cooled")
(imdl WAIT [v0: v1: (ti v2: v3:)] [0s:200ms:0us 0s:800ms:0us] true)
(imdl WAIT [v0: t (ti v2: v3:)] [0s:200ms:0us 0s:800ms:0us] true)
(mk.val v0: essence pause)
(mk.val v1: essence toothbrush)
(mk.val v1: state "screwdriver-shaped")
(mk.val t state "screwdriver-shaped")
(mk.val v3: state "screwdriver-shaped")
(imdl FORM [v0: t (ti v2: v3:)] [0s:100ms:0us 0s:700ms:0us] true)
(imdl FORM [v0: v1: (ti v2: v3:)] [0s:100ms:0us 0s:700ms:0us] true)
(mk.val v0: essence form)
(mk.val t essence toothbrush)
(mk.val t state "heated")
(cmd use_tool [l t])
(mk.val t state "heated")
```

Figure 6.3: AERA’s plan for the toothbrush problem.

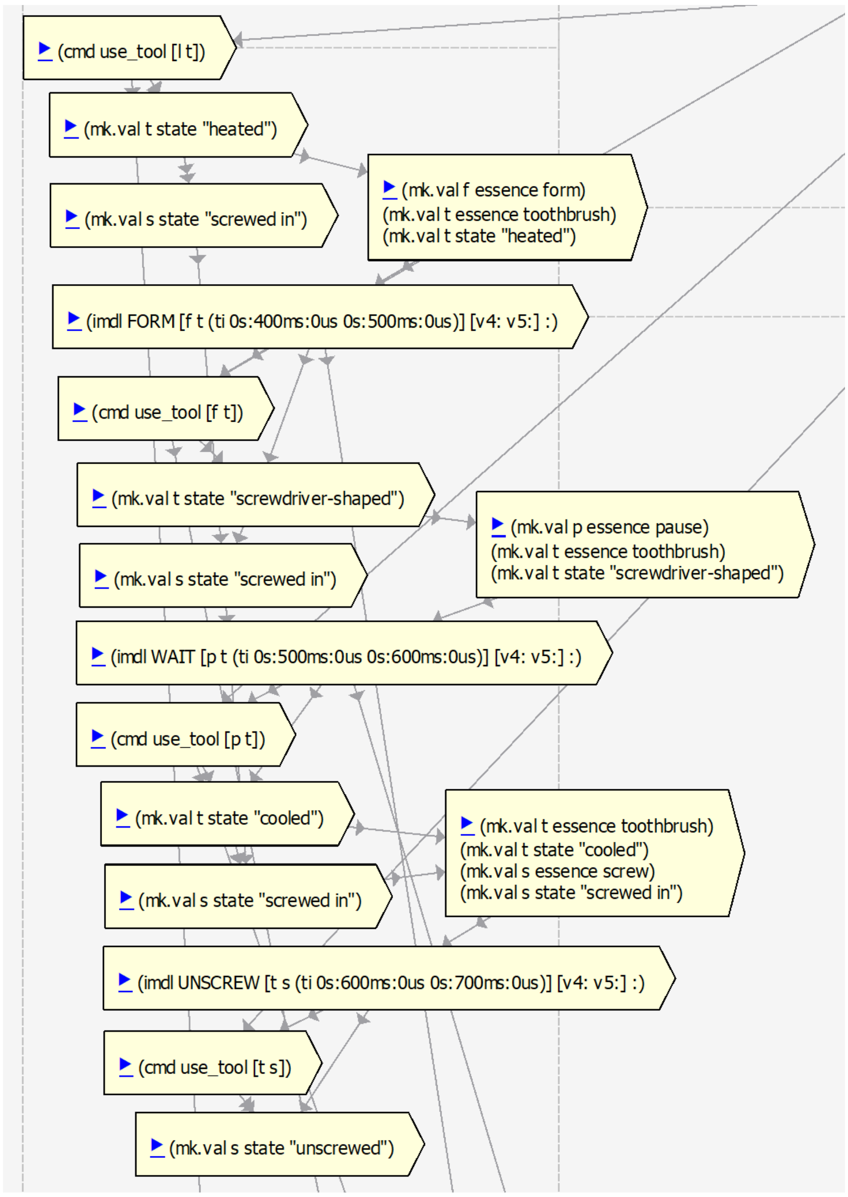
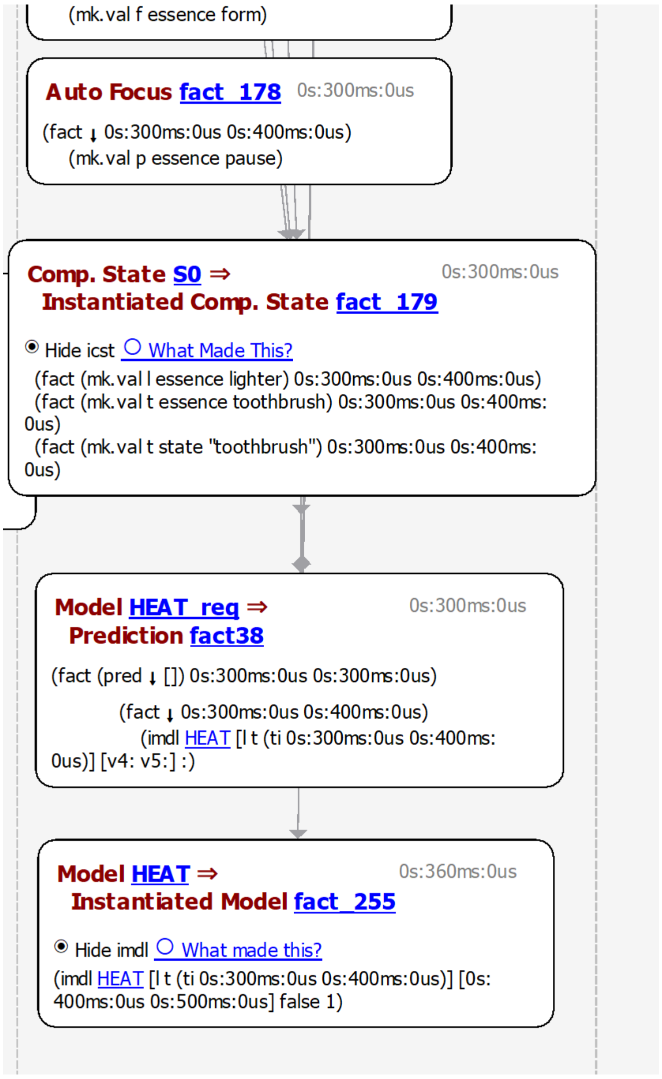

(cmd use_tool [l t])
(mk.val t state "heated")
(mk.val s state "screwed in")
(mk.val f essence form)
(mk.val t essence toothbrush)
(mk.val t state "heated")
(imdl FORM [f t (ti 0s:400ms:0us 0s:500ms:0us)] [v4: v5:] :)
(cmd use_tool [f t])
(mk.val t state "screwdriver-shaped")
(mk.val s state "screwed in")
(mk.val p essence pause)
(mk.val t essence toothbrush)
(mk.val t state "screwdriver-shaped")
(imdl WAIT [p t (ti 0s:500ms:0us 0s:600ms:0us)] [v4: v5:] :)
(cmd use_tool [p t])
(mk.val t state "cooled")
(mk.val s state "screwed in")
(mk.val t essence toothbrush)
(mk.val t state "cooled")
(mk.val s essence screw)
(mk.val s state "screwed in")
(imdl UNSCREW [t s (ti 0s:600ms:0us 0s:700ms:0us)] [v4: v5:] :)
(cmd use_tool [t s])
(mk.val s state "unscrewed")

Figure 6.4: AERA chains its plan forwards.



(mk.val f essence form)

Auto Focus fact_178 0s:300ms:0us

(fact ↓ 0s:300ms:0us 0s:400ms:0us)
(mk.val p essence pause)

Comp. State S0 ⇒ Instantiated Comp. State fact_179 0s:300ms:0us

● Hide icst ○ What Made This?

(fact (mk.val l essence lighter) 0s:300ms:0us 0s:400ms:0us)
(fact (mk.val t essence toothbrush) 0s:300ms:0us 0s:400ms:0us)
(fact (mk.val t state "toothbrush") 0s:300ms:0us 0s:400ms:0us)

Model HEAT_req ⇒ Prediction fact38 0s:300ms:0us

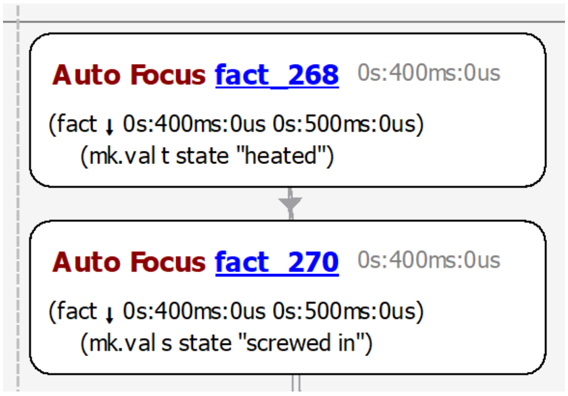
(fact (pred ↓ []) 0s:300ms:0us 0s:300ms:0us)
(fact ↓ 0s:300ms:0us 0s:400ms:0us)
(imdl HEAT [l t (ti 0s:300ms:0us 0s:400ms:0us)] [v4: v5:] :)

Model HEAT ⇒ Instantiated Model fact_255 0s:360ms:0us

● Hide imdl ○ What made this?

(imdl HEAT [l t (ti 0s:300ms:0us 0s:400ms:0us)] [0s:400ms:0us 0s:500ms:0us] false 1)

Figure 6.5: AERA takes the first step.



Auto Focus fact_268 0s:400ms:0us

```
(fact ↓ 0s:400ms:0us 0s:500ms:0us)
   (mk.val t state "heated")
```

Auto Focus fact_270 0s:400ms:0us

```
(fact ↓ 0s:400ms:0us 0s:500ms:0us)
   (mk.val s state "screwed in")
```

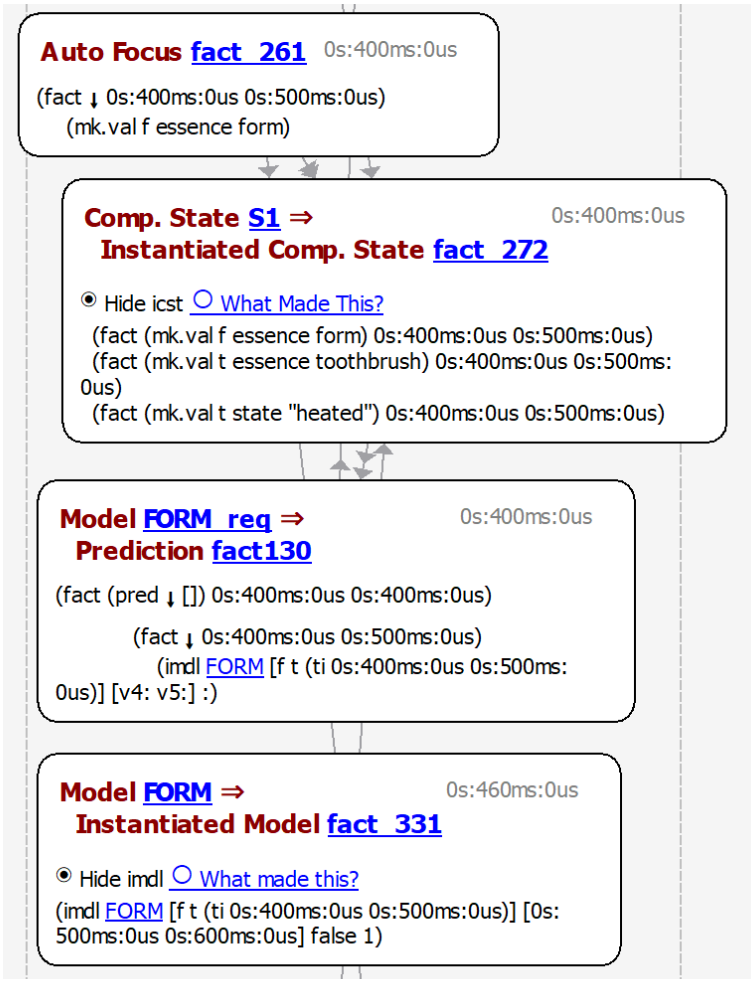
Figure 6.6: Results of AERA's first step.

screw remains unaffected and is still `"screwed in"`.

In Figure 6.7, we see that AERA has taken note of the state change and that the conditions in `S1` are now met and it can be instantiated. Once again, it works out a plan chaining backward from its goal of unscrewing the screw and forward from the current state of the now-heated toothbrush. The reason for this continuous assessment and planning is that AERA is designed to operate in an unpredictable environment; if there was a change in the environment, AERA would catch this and adapt its approach. Since, in this case, the new plan is nearly identical to the one in Figures 6.3 and 6.4, it is omitted for brevity. With this plan complete, AERA predicts that it can use the `FORM` operation as its next step. Looking at the bottom of Figure 6.7, we see that AERA has indeed decided to instantiate the `FORM` model and use the form tool to reshape the toothbrush.

This repeats for every step of the process until the goal is achieved. Figure 6.8 shows the final results of AERA's work; at each step, AERA assessed the environment, made a plan, and issued a `use_tool` command, all so that `t` can advance closer to its final form. 600 ms into the process, `t` is now ready to be used as a screwdriver and so AERA executes the `use_tool` command to try and unscrew `s`. In the next step, we see that `s` has finally reached the state `"unscrewed"`, and AERA has achieved its goal. As such, it takes no further action.

AERA has successfully worked its way through the toothbrush problem. At each stage, it worked its way back from the goal, simulated a plan, and then took the first step of that plan; its ultimate goal was achieved at the end of this process.


Auto Focus fact_261 0s:400ms:0us

(fact ↓ 0s:400ms:0us 0s:500ms:0us)
(mk.val f essence form)

Comp. State S1 ⇒ Instantiated Comp. State fact_272 0s:400ms:0us

● Hide icst ○ What Made This?

(fact (mk.val f essence form) 0s:400ms:0us 0s:500ms:0us)
(fact (mk.val t essence toothbrush) 0s:400ms:0us 0s:500ms:0us)
(fact (mk.val t state "heated") 0s:400ms:0us 0s:500ms:0us)

Model FORM_req ⇒ Prediction fact130 0s:400ms:0us

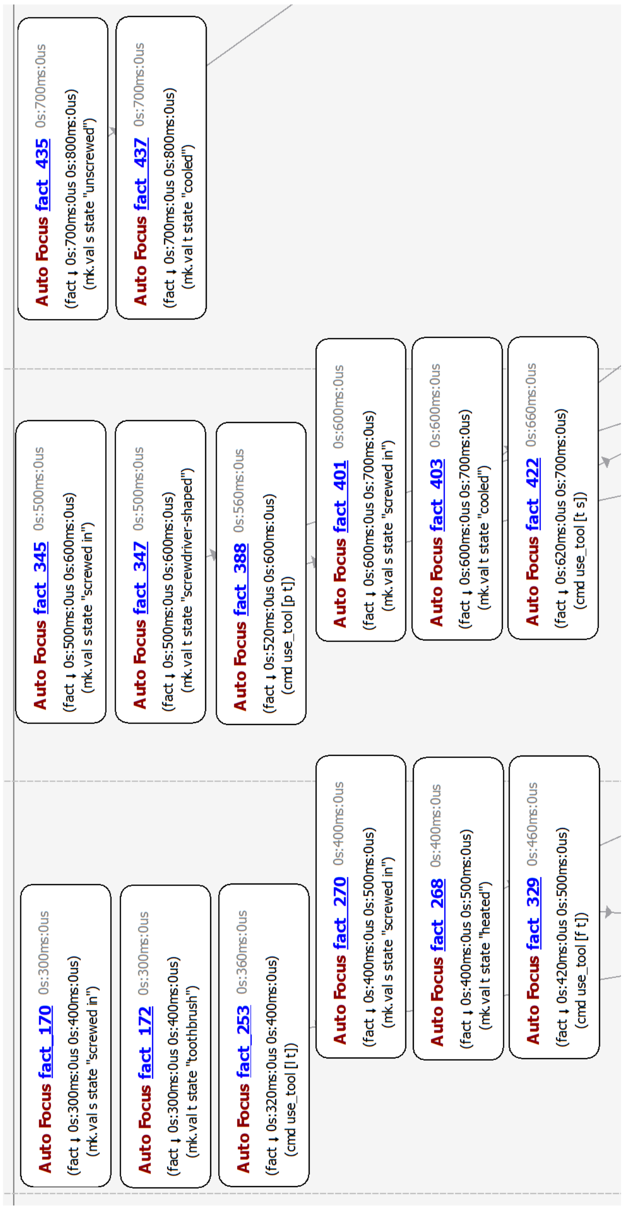
(fact (pred ↓ []) 0s:400ms:0us 0s:400ms:0us)
(fact ↓ 0s:400ms:0us 0s:500ms:0us)
(imdl FORM [f t (ti 0s:400ms:0us 0s:500ms:0us)] [v4: v5:] :)

Model FORM ⇒ Instantiated Model fact_331 0s:460ms:0us

● Hide imdl ○ What made this?

(imdl FORM [f t (ti 0s:400ms:0us 0s:500ms:0us)] [0s:500ms:0us 0s:600ms:0us] false 1)

Figure 6.7: AERA takes the second step.



Auto Focus fact_170 0s:300ms:0us
(fact ↓ 0s:300ms:0us 0s:400ms:0us)
(mk.val s state "screwed in")

Auto Focus fact_172 0s:300ms:0us
(fact ↓ 0s:300ms:0us 0s:400ms:0us)
(mk.val t state "toothbrush")

Auto Focus fact_253 0s:360ms:0us
(fact ↓ 0s:320ms:0us 0s:400ms:0us)
(cmd use_tool [l t])

Auto Focus fact_270 0s:400ms:0us
(fact ↓ 0s:400ms:0us 0s:500ms:0us)
(mk.val s state "screwed in")

Auto Focus fact_268 0s:400ms:0us
(fact ↓ 0s:400ms:0us 0s:500ms:0us)
(mk.val t state "heated")

Auto Focus fact_329 0s:460ms:0us
(fact ↓ 0s:420ms:0us 0s:500ms:0us)
(cmd use_tool [f t])

Auto Focus fact_345 0s:500ms:0us
(fact ↓ 0s:500ms:0us 0s:600ms:0us)
(mk.val s state "screwed in")

Auto Focus fact_347 0s:500ms:0us
(fact ↓ 0s:500ms:0us 0s:600ms:0us)
(mk.val t state "screwdriver-shaped")

Auto Focus fact_388 0s:560ms:0us
(fact ↓ 0s:520ms:0us 0s:600ms:0us)
(cmd use_tool [p t])

Auto Focus fact_401 0s:600ms:0us
(fact ↓ 0s:600ms:0us 0s:700ms:0us)
(mk.val s state "screwed in")

Auto Focus fact_403 0s:600ms:0us
(fact ↓ 0s:600ms:0us 0s:700ms:0us)
(mk.val t state "cooled")

Auto Focus fact_422 0s:660ms:0us
(fact ↓ 0s:620ms:0us 0s:700ms:0us)
(cmd use_tool [t s])

Auto Focus fact_435 0s:700ms:0us
(fact ↓ 0s:700ms:0us 0s:800ms:0us)
(mk.val s state "unscrewed")

Auto Focus fact_437 0s:700ms:0us
(fact ↓ 0s:700ms:0us 0s:800ms:0us)
(mk.val t state "cooled")

Figure 6.8: AERA works through the toothbrush problem.

6.3 Reflections on a Technology Demonstrator

Though a very simple example, this test demonstrates the problem-solving ability of even a basic EDA such as EDA-0. It was able to use the knowledge with which it was supplied to reason its way through the four-step process of achieving its goal. Though the process seems simple and obvious to a human observer, it is worth remembering that AERA was responsible for taking this knowledge, determining what of it was relevant, and assembling it into a plan to achieve its goal. Additionally, though the environment and knowledge were both heavily scaffolded, AERA was never told what actions to take and in what order. It was up to AERA to surmise the order based on the prerequisite conditions listed in its causal models in the same way that a human engineer would put together a plan based on knowledge of individual processes.

Nothing went wrong in this test and, in fact, it was not possible for much to go wrong. The way the test was structured was such that there was a clear path from the start condition to the goal, all AERA had to do was find that path and execute the necessary operations to follow it. Perhaps a more complex environment that encoded a greater consideration of material properties and object attributes would provide some dead-ends for AERA to experiment with and avoid. For example, AERA could learn that the `WAIT` operation can be applied at any point in the process but is only helpful when the toothbrush needs to be cooled down. There are also other possibilities for improvements; AERA could be provided with more general information, the environment could be more realistic, the process could be more complicated, and so on. While these modifications are out of the scope of this work, they make up a clear roadmap to future more complex experiments.

However, as it stands, this basic example is enough to demonstrate that a conceptual design process can be navigated by an AI agent given the requisite reasoning abilities and subject-matter knowledge. This experiment also shows the utility of the O&O approach and how it can be used to guide an experience-focused AGI-aspiring agent like AERA into the unique environment of a design problem. Finally, this shows that the same problem can have equivalent and compatible representations in both POODL and Replicode thereby guiding the way to a transpiler and communication between EDA components. In short, though simplistic, EDA-0's journey through modifying a toothbrush to make a screwdriver is the first important step towards more complex and capable EDAs.

Chapter 7

Discussion and Conclusion

7.1 Discussion

The architecture proposed in this thesis represents a radical departure from existing design research. To this author's knowledge, no other design agent uses a hybrid high- and low-level approach to solve engineering design problems. While chaining environment models to solve design problems is not a new concept, putting a reasoning agent wholly in charge of the conceptual design stage is. Additionally, there does not appear to be much research into a system that combines the reasoning capabilities of a conceptual agent with the generative capabilities of an optimizing agent into a single unified design system. With this departure come the hopes that the proposed engineering design agent (EDA) architecture is capable of taking on the challenge of conceptual design and that the resulting agents will achieve a much greater degree of problem-solving and independence.

This work's most important contribution is the EDA architecture itself. Like a human designer, an EDA should be capable of solving a problem approximately and conceptually before moving on to the detail design stage. At that point, both a human designer and an EDA can use generative design approaches and testing libraries to create concrete part designs. When the design is complete, the human engineer collates their part designs and provides a detailed set of instructions for implementing their solution in the real world; a fully-developed EDA can do the same.

To do this, the EDA must be able to solve different parts of the design problem in different ways. The high-level designer (HLD) can work the problem abstractly using its engineering knowledge and experience to put together a rough solution for the problem. It does not need to know every detail of every part just yet, it only needs to understand the process and how to arrange its ideas into a viable solution. The low-level optimizer (LLO) can then take these ideas and synthesize them into part designs using existing work in generative design. Because this is more computationally intensive than reasoning, it is only done after a conceptual solution has been found. The LLO also works with the Tester to prove and validate its solutions; the Tester uses existing engineering software and test packages to ensure that the EDA's solutions are up to the same standards of work as those of a human engineer. Finally, the OR assembles these ideas and parts into a series of drawings and documents that provide all the information needed to realize the EDA's solution.

Existing research has already developed generative designers and coupled them with test packages; this technology is well-developed and has been deployed reasonably widely (cf. Autodesk, 2023). In this way, the EDA architecture is not such a radical design, the only new idea is that a reasoning agent should operate the generative designer instead of a human. To make this work, the reasoning agent and generative designer must be in constant communication. In an EDA, this means that the HLD, LLO, and Tester must be able to constantly exchange specifications and feedback. This means that the EDA should be able to incorporate feedback at any stage of this design — even if this means a temporary return to the drawing board. Regardless of whether a part specification failed, a new design requirement was passed down from the user, or any other change was made, an EDA can accept the change to the problem and begin working on an updated solution. All this said, the EDA does not require user feedback to operate. Between its experimental capabilities and the Optimizer-Tester Combined Environment (OTCE), it can handle a significant degree of uncertainty in its problem-solving. If set to operate in a high-autonomy mode, the EDA may be able to solve any problems that arise by working them in its internal environments; it would only contact the user if it was completely unable to solve the problem. This setup addresses the challenge in Research Question 4 (page 6) as the EDA could be configured to work anywhere in the range from high cooperation to high autonomy and is able to recognize its shortcomings and ask for help when needed.

Of course, none of this would be possible without the ability to represent design problems in a formal manner. To do this, this thesis uses the concepts of a critical path and a decision tree as causal models of solutions since a solution can be viewed as the summation of the designer's actions and decisions, respectively. This allows us to find common ground between the imaginary world of a design problem and the real world of its solution: though time may not pass for a design on paper, both the design and its prototype will rely on the same causal models of the world. This satisfies Research Question 1 (page 5). It also addresses Research Question 2 (page 5) as these causal representations of design tasks allow for an assessment of difficulty and the degree to which an agent is fit for a given task. Finally, it provides a method by which a design agent can break a design into a set of causal models of individual phenomena each of which can be reused. This allows an EDA to link a problem statement to its existing knowledge as in Research Question 3 (page 6).

One can even simplify the decision tree even further and reframe it as a sequence of operations performed on objects in an attempt to build a solution to the problem.

This Operations & and Objects (O&O) paradigm can then guide the creation of an engineering description language or EDL. While description languages are not new — hardware description languages for the computer-aided design of computer chips have been commonplace for decades now — it does not appear as if much work has been done on EDLs before. This is likely due to the complexity and generality of the engineering method; how can one hope to capture all possible engineering solutions in a structured language fit for an AI agent? This thesis proposed six criteria that an EDL must meet to be useful: it must be rigorous, permit the linking of experiential knowledge to a problem, be able to encode the difficulty of a task, be scaffolded to support design problems with no excessive generality, be model- and agent-agnostic, and able to be used in feedback and communication. These are difficult criteria to fulfill, but it is hoped that simplifying design problems into terms of operations performed on objects will take a step toward solving this problem. As such, the hope is that any language that implements O&O should be capable of, at the very least, basic conceptual design. This thesis addresses Research Question 5 (page 6) with the introduction of the Prototype Operations & Objects Description Language or POODL. Even though it only implements a small part of O&O, the hope is that it lays the groundwork for future EDLs capable of handling ever more difficult problems. While the EDL concepts proposed here are almost certainly in need of refinement, it is hoped that they may guide the way toward the EDLs of future, more powerful, EDAs.

Finally, another key difference between this thesis' approach and previous work is its connections to the study of artificial general intelligence or AGI. While previous research into knowledge-based engineering (KBE) has already investigated reasoning to a considerable extent, not many of these systems appear to have the capability to learn on their own. Taking an AGI approach to this problem yields a few key differences: problem decomposition, agency, and experiential learning.

First, problem decomposition draws from task theory, a branch of AGI research that focuses explicitly on decomposing tasks into their atomic steps and then determining how difficult it will be for an agent to perform the task. This even works for unfamiliar problems in which an AI agent can do its best to break large problems into smaller, more manageable, and more familiar problems. An EDA capable of doing this would be able to solve problems to a higher degree of abstraction than existing systems and take on uncertainty with more confidence.

Second, this level of problem decomposition necessitates an agent with more agency. An EDA capable of problem decomposition must be able to take charge in breaking a problem down and selecting the correct path to a solution as a human engineer would when solving a problem. This also allows EDAs to work more like design partners then design tools; a human engineer could easily delegate design tasks to an EDA much as they would hand them off to a colleague.

Third, an EDA has the ability to expand its capabilities as it works by learning from its experience and conducting its own experiments. As it works, an EDA makes models of the problem and remembers whether they help it on the way to a solution or not. In later problems, it can recall these models and use them to accelerate the path to the current solution rather than working the problem from scratch every time. When it does not have experience to fall back on, the EDA can conduct experiments to fill gaps in its knowledge. Like a human engineer, it can find a path to a solution even if that requires resolving some unknowns first. This is not to say that EDAs are completely different from existing systems, however. For example, an EDA will also be designed to encode its knowledge in causal models of the world similar to those a KBE system would use. The difference is that the EDA can add new models, modify existing ones, and remove ones that no longer fit; all this forms a kind of experiential knowledge base that is built as the EDA works through problems.

All this said, most of this is not the EDA's purview — it is up to the reasoning agent at the core of the HLD. In the architecture proposed by this thesis, AERA is the system in charge of reasoning and managing knowledge. From this perspective, the rest of the EDA is almost just one very elaborate peripheral for AERA. Including an existing reasoning system in an EDA's structure also means that research can focus on the engineering design part of the problem and less on the reasoning part. Finally, the EDA can also take advantage of upgrades made to AERA in part of its development. As before, it is hoped that this approach to the development of a design agent serves as a guide to a promising new avenue of research.

That is not to say that these elements are perfect, however. The EDA architecture is still largely unproven; EDA-0 is a promising first step, but it barely implements an HLD to say nothing of the other components. Though POODL is fairly capable as a language, it comes with significant limitations that will require revision and development. It is also likely that the criteria an EDL must fulfill will change and develop over time as new systems are developed. Finally, the integration of ideas from AGI research is not a simple task. The field is still grappling with very difficult questions and it is entirely possible that the ideas at the very core of the HLD will be discarded in favor of newer, better approaches to problem-solving and learning. All this said, this research is still worth pursuing because of the potential to solve the problem of conceptual engineering design. A complete and fully functional EDA would be a very powerful tool and the journey to develop one will undoubtedly yield fascinating insights into engineering, problem-solving, and myriad other problems.

All of these elements make the EDA architecture proposed here significantly different from the existing approaches to design research. This EDA uses a hybrid system capable of both reasoning and geometric part design, it requires a description language capable of unambiguously describing a design problem, and it pulls in ideas and agents currently used in researching the much larger problem of artificial general intelligence. The EDA architecture proposed here, however, shows promise not in spite of these differences, but because of them. The goal is to use these ideas to expand the limits of current design systems by giving them the agency to break down problems, organize their knowledge, and learn as they go. This also gives the agent the ability to justify its design decisions; it can be trusted as more than just a black-box generative algorithm. With development, this work could pave the way to EDAs that work alongside human engineers as trusted partners. Though the work presented here may resemble nothing of the future state of EDA research, it is enough to take a few key steps in that direction.

7.2 Next Steps

The research presented in this thesis is only the first part of a long journey toward EDAs that are useful in the real world. As such, improvements are warranted more or less across the board. The author's intention is to continue this research by first addressing the following areas of improvement.

First, the HLD is in need of some improvements in its reasoning, knowledge, and interaction. Equipping the HLD with the ability to learn by experimentation and then analogize what it has learned to other topics would be a significant improvement. Also useful would be the integration of proper programmable knowledge modules to allow the EDA to work within disciplines not strictly specified in its seed program. Achieving better interaction could be done by implementing a proper POODL transpiler to allow the HLD to communicate live with the user and with the other components of the EDA. One could even upgrade the EDL to cover a more expansive version of O&O and allows for more creative and specific tool use as well as the creation of parts made up of multiple components. AERA already has the capacity for most of these upgrades, it would simply be a matter of developing the EDA further to properly take advantage of this.

Next, the optimizer-tester combined environment (OTCE) should be fully implemented with a design that can cover a more realistic virtual workshop simulation. Tools could be backed up by more specific causal models and actual physical performance calculations. To make this work, the OTCE will need to be moved from the Replicode emulator to a separate C++ module within the EDA; this will also require an EDL upgrade to enable communication within the EDA. Finally, some balance will need to be found between simulation fidelity and abstraction in the OTCE, a very detailed simulation will allow the HLD to be more creative in its use of tools but a more abstract simulation would be more computationally efficient.

Finally, making these OTCE upgrades happen will also require upgrades to the LLO and Tester. Like the OTCE, both the LLO and Tester will need implementations external to the Replicode emulator. They will most likely join the OTCE as separate components written in C++. For the LLO, this would allow for the development of a basic genetic algorithm for part design. As for the Tester, it would be able to implement structural modeling equations, integrate with finite element analysis libraries, and bring in any other kind of test package needed for the EDA's current problem. These upgrades should not be difficult to implement, they will simply take more time and development effort.

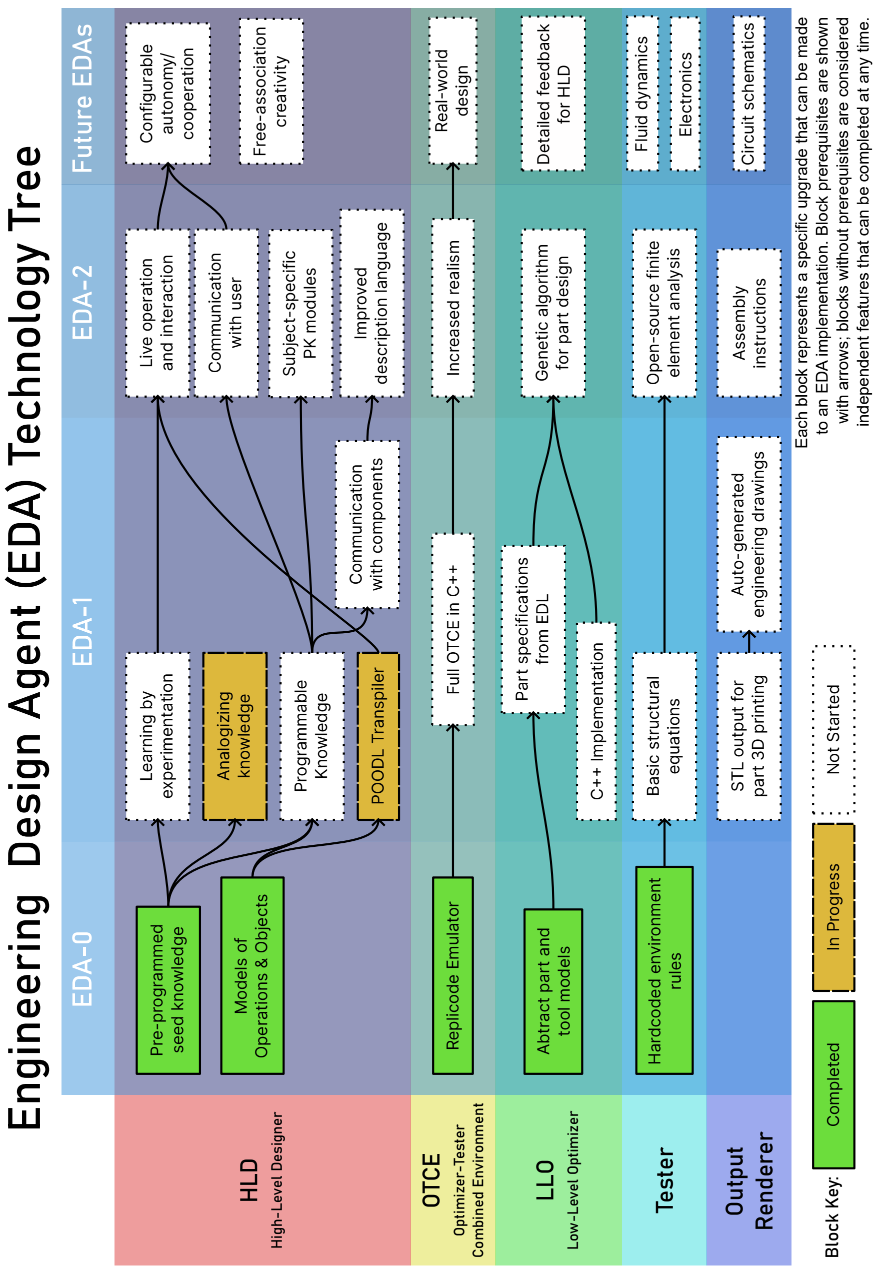

Engineering Design Agent (EDA) Technology Tree
EDA-0
EDA-1
EDA-2
Future EDAs
HLD
High-Level Designer
OTCE
Optimizer-Tester Combined Environment
LLO
Low-Level Optimizer
Tester
Output Renderer
Pre-programmed seed knowledge
Models of Operations & Objects
Learning by experimentation
Analogizing knowledge
Programmable Knowledge
POODL Transpiler
Communication with components
Live operation and interaction
Communication with user
Subject-specific PK modules
Improved description language
Configurable autonomy/ cooperation
Free-association creativity
Replicode Emulator
Full OTCE in C++
Increased realism
Real-world design
Abtract part and tool models
Part specifications from EDL
C++ Implementation
Genetic algorithm for part design
Detailed feedback for HLD
Hardcoded environment rules
Basic structural equations
Open-source finite element analysis
Fluid dynamics
Electronics
STL output for part 3D printing
Auto-generated engineering drawings
Assembly instructions
Circuit schematics
Block Key:
Completed
In Progress
Not Started
Each block represents a specific upgrade that can be made to an EDA implementation. Block prerequisites are shown with arrows; blocks without prerequisites are considered independent features that can be completed at any time.

Figure 7.1: A technology tree for engineering design agents

All of these upgrades are given in Figure 7.1 in the form of a flowchart or technology tree. It starts with EDA-0 and adds on these upgrades, progressively working towards future EDA versions. From all of this, it is clear that EDA research is far from complete. The development of a fully-functioning EDA that can partner up with a human engineer is not a far-off goal, it is simply a matter of developing the technology. While there are almost certainly steps missing from the roadmap in Figure 7.1, it represents a starting point for the path to a more powerful EDA. There is still much research and development to go, this author fully intends to continue working on this fascinating problem.

7.3 Conclusion

Automated design is not a new research topic and has already yielded systems capable of beautiful generated structures and efficient reasoned solutions both of which could outperform a human engineer in complexity and speed. However, these systems can only handle so much of the engineering process. Generative designers' reliance on numerical methods prevents them from taking on the challenge of the "back of the napkin" conceptual design so familiar to human engineers. While knowledge-based engineering systems are capable of this kind of reasoning, they cannot act with their own agency or autonomously expand the knowledge bases they use to solve problems.

The engineering design agent (EDA) architecture proposed here offers the first step towards a potential solution to these problems. By programming AGI-aspiring agents to perform the engineering method, the hope is to create a high-level designer (HLD) capable of autonomously handling the conceptual stage of the design process and learning as it works. This component can then work with existing generative design and testing systems in the low-level optimizer (LLO) and Tester, respectively, to turn its ideas into concrete part designs. Finally, the completed solution can be presented as any engineer would present their work, with drawings, schematics, instructions, and any other documentation required to make the EDA's design a reality. All through the process, the EDA uses an engineering description language (EDL) to communicate within itself and with the user, the latter being able to answer questions and provide feedback during the design process. The prototype, EDA-0, is designed to demonstrate the most essential elements of this EDA architecture; it is simple, but its success indicates a promising avenue for future research.

The architecture proposed here is radically different from those of existing automated design systems. The hope is that, by adding a new autonomous conceptual design element, this can become a new kind of system capable of solving much more general kinds of problems: an engineering design agent. When completed, such an EDA would be capable of working through the design process like a human engineer, starting from the most abstract definition of the problem and working towards a concrete, actionable solution. The EDA would also learn as it goes, building a bank of experience and subject-matter knowledge in just the way a human engineer would. As it works, such an EDA could describe its design process and explain its reasoning to the human engineers it is working with. It could be more than just a design tool, an EDA could be a member of a design team. The concepts presented in this thesis are basic and the demonstrator system that implements them even more so, but the intention is that they will guide the development of much more powerful EDAs in the future. Perhaps, humans may one day work alongside EDAs as partners in solving the greatest engineering challenges of our time.

Bibliography

ACT-R Research Group. ACT-R, 2013. URL https://act-r.psy.cmu.edu/.

S. Altavilla and E. Blanco. Are AI Tools Going to be the New Designers? A Taxonomy for Measuring the Level of Automation of Design Activities. *Proceedings of the Design Society: DESIGN Conference*, 1:81–90, May 2020. ISSN 2633-7762. doi: https://doi.org/10.1017/dsd.2020.286. Publisher: Cambridge University Press.

John R. Anderson and Christian D. Schunn. Implications of the ACT-R Learning Theory: No Magic Bullets. *Advances in Instructional Psychology*, 5: 1–34, 2000. URL https://www.lrdc.pitt.edu/Schunn/research/papers/NoMagicBullets.pdf.

Konstantine Arkoudas. GPT-4 Can't Reason. preprint, Computer Science and Mathematics, August 2023. URL https://www.preprints.org/manuscript/202308.0148/v2.

Autodesk. What is Generative Design | Tools Software | Autodesk, 2023. URL https://www.autodesk.com/solutions/generative-design.

Matteo Belenchia, Dr Kristinn R Thórisson, and Dr Emanuela Merelli. *Towards a Theory of Causally Grounded Tasks*. PhD thesis, Reykjavík University, 2021. URL https://skemman.is/bitstream/1946/39956/1/msthesis.pdf.

Matteo Belenchia, Kristinn R. Thórisson, Leonard M. Eberding, and Arash Sheikhlar. Elements of Task Theory. In *AGI 2021*, volume 13154, pages 19–29, San Francisco Bay Area, California, and Virtual Conference, January 2022. Springer International Publishing. ISBN 978-3-030-93757-7 978-3-030-93758-4. doi: https://doi.org/10.1007/978-3-030-93758-4_3.

Apoorv Naresh Bhatt, Anubhab Majumder, and Amaresh Chakrabarti. Analyzing the modes of reasoning in design using the SAPPhIRE model of causality and the Extended Integrated Model of Designing. *AI EDAM*, 35(4):384–403, November 2021. ISSN 0890-0604, 1469-1760. doi: https://doi.org/10.1017/S0890060421000214. Publisher: Cambridge University Press.

Jordi Bieger and Kristinn R. Thórisson. Requirements for General Intelligence: A Case Study in Trustworthy Cumulative Learning for Air Traffic Control. Stockholm, Sweden, July 2018. URL https://api.semanticscholar.org/CorpusID:53686498.

Daniel G. Bobrow, Ronald M. Kaplan, Martin Kay, Donald A. Norman, Henry Thompson, and Terry Winograd. GUS, a frame-driven dialog system. *Artificial Intelligence*, 8(2):155–173, April 1977. ISSN 00043702. doi: 10.1016/0004-3702(77)90018-2. URL https://linkinghub.elsevier.com/retrieve/pii/0004370277900182.

Nick Bostrom. Information Hazards: A Typology of Potential Harms from Knowledge. *Review of Contemporary Philosophy*, 10:44–79, 2011. URL https://nickbostrom.com/information-hazards.pdf.

Sylvie Castagne, Richard Curran, and Paul Collopy. Implementation of value-driven optimisation for the design of aircraft fuselage panels. *International Journal of Production Economics*, 117(2):381–388, February 2009. ISSN 0925-5273. doi: https://doi.org/10.1016/j.ijpe.2008.12.005.

Department of Computer and Information Sciences, Temple University. OpenNARS for Applications, February 2020. URL https://github.com/opennars/OpenNARS-for-Applications. original-date: 2020-02-16T17:33:05Z.

Mark Drela and Harlod Youngren. XFOIL Subsonic Airfoil Development System, December 2013. URL https://web.mit.edu/drela/Public/web/xfoil/.

Leonard M. Eberding, Matteo Belenchia, Arash Sheikhlar, and Kristinn R. Thórisson. About the Intricacy of Tasks. In *AGI 2021*, volume 13154, pages 65–74, San Francisco Bay Area, California, and Virtual Conference, 2022. Springer International Publishing. ISBN 978-3-030-93757-7 978-3-030-93758-4. doi: https://doi.org/10.1007/978-3-030-93758-4_8.

Samir Emdanat, George Stiny, and Emmanuel-George Vakaló. Generative Systems in Design. *AI EDAM*, 13(4):239–240, September 1999. ISSN 1469-1760, 0890-0604. doi: https://doi.org/10.1017/S0890060499134000. Publisher: Cambridge University Press.

Ben Goertzel. OpenCogPrime: A cognitive synergy based architecture for artificial general intelligence. In *2009 8th IEEE International Conference on Cognitive Informatics*, pages 60–68, Kowloon, Hong Kong, June 2009. IEEE. ISBN 978-1-4244-4642-1. doi: https://doi.org/10.1109/COGINF.2009.5250807.

Bill Hammack. *The Things We Make: The Unknown History of Invention from Cathedrals to Soda Cans*. Sourcebooks, 2023. ISBN 978-1-72821-575-4.

Patrick Hammer, Tony Lofthouse, Enzo Fenoglio, Hugo Latapie, and Pei Wang. A Reasoning Based Model for Anomaly Detection in the Smart City Domain. In Kohei Arai, Supriya Kapoor, and Rahul Bhatia, editors, *Intelligent Systems and Applications*, volume 2 of *Advances in Intelligent Systems and Computing*, pages 144–159. Springer International Publishing, August 2020. ISBN 978-3-030-55187-2. doi: https://doi.org/10.1007/978-3-030-55187-2_13.

Icelandic Institute for Intelligent Machines. AERA, June 2018. URL https://github.com/IIIM-IS/AERA.

Icelandic Institute for Intelligent Machines. Open AERA, 2021. URL https://openaera.org.

Icelandic Institute for Intelligent Machines and Center for Analysis and Design of Intelligent Agents. AERA License, June 2023. URL https://github.com/IIIM-IS/AERA/blob/352d7d980b7c80bdc17dddf30dd159df16709d3c/License.txt. original-date: 2018-06-29T17:22:00Z.

Patricia Kügler, Fabian Dworschak, Benjamin Schleich, and Sandro Wartzack. The evolution of knowledge-based engineering from a design research perspective: Literature review 2012–2021. *Advanced engineering informatics*, 55:101892–, 2023. ISSN 1474-0346. doi: 10.1016/j.aei.2023.101892. Publisher: Elsevier Ltd.

Gianfranco La Rocca. Knowledge based engineering: Between AI and CAD. Review of a language based technology to support engineering design. *Advanced Engineering Informatics*, 26(2):159–179, April 2012. ISSN 1474-0346. doi: https://doi.org/10.1016/j.aei.2012.02.002.

Christopher McComb, Jonathan Cagan, and Kenneth Kotovsky. Eliciting configuration design heuristics with hidden Markov models. *DS 87-2 Proceedings of the 21st International Conference on Engineering Design (ICED 17) Vol 2: Design Processes, Design Organisation and Management, Vancouver, Canada, 21-25.08.2017*, pages 051–060, 2017. ISSN 2220-4342. URL https://www.designsociety.org/publication/39559/Eliciting+configuration+design+heuristics+with+hidden+Markov+models. ISBN: 9781904670902.

Meredith Hawes. D.I.Y. - The Dangers of Mixing Household Chemicals | NFPA, April 2020. URL https://www.nfpa.org/News-and-Research/Publications-and-media/Blogs-Landing-Page/Safety-Source/Blog-Posts/2020/04/13/diy-the-dangers-of-mixing-chemicals.

Henry Minsky. How Can a Computer Learn "Common Sense" as Quickly as Quickly and Efficiently as a Human Infant?, September 2022.

National Aeronautics and Space Administration. NASA STRuctrual ANalysis (NASTRAN)(LAR-16804-GS) | NASA Software Catalog, 2015. URL https://software.nasa.gov/software/LAR-16804-GS.

Allen Newell and Paul S. Rosenbloom. Soar: An Architecture for General Intelligence. *Artificial Intelligence*, 33(1):1–64, December 1986.

Isaac Newton, N. W. Chittenden, and Andrew Motte. *Newton's Principia. The mathematical principles of natural philosophy*. D. Adee, New York, 1848. URL https://www.loc.gov/item/04014428.

E. Nivel, Kristinn R. Thórisson, Bas R. Steunebrink, H. Dindo, G. Pezzulo, M. Rodriguez, C. Hernandez, D. Ognibene, J. Schmidhuber, R. Sanz, H. P. Helgason, A. Chella, and G. K. Jonsson. Bounded Recursive Self-Improvement, 2013.

Eric Nivel and Kristinn R. Thórisson. Replicode: A Constructivist Programming Paradigm and Language. Technical Report RUTR-SCS13001, Reykjavík University School of Computer Science, 2013.

Eric Nivel, Kristinn R. Thórisson, Bas R. Steunebrink, Haris Dindo, Giovanni Pezzulo, Manuel Rodriguez, Carlos Hernandez, Dimitri Ognibene, Jürgen Schmidhuber, Ricardo Sanz, Helgi P Helgason, Antonio Chella, and Gudberg K Jonsson. Autonomous Acquisition of Natural Language. page 10, 2014a. URL https://api.semanticscholar.org/CorpusID:8218230.

Eric Nivel, Kristinn R. Thórisson, Bas R. Steunebrink, Haris Dindo, Giovanni Pezzulo, Manuel Rodríguez, Carlos Hernández, Dimitri Ognibene, Jürgen Schmidhuber, Ricardo Sanz, Helgi P. Helgason, and Antonio Chella. Bounded Seed-AGI. In Ben Goertzel, Laurent Orseau, and Javier Snaider, editors, *AGI 2014*, Lecture Notes in Computer Science, pages 85–96, Quebec City, Canada, 2014b. Springer International Publishing. ISBN 978-3-319-09274-4. doi: https://doi.org/10.1007/978-3-319-09274-4_9.

Mats Nordlund, Taesik Lee, and Sang-Gook Kim. Axiomatic Design: 30 Years After. In *Volume 15: Advances in Multidisciplinary Engineering*, page V015T19A009, Houston, Texas, USA, November 2015. American Society of Mechanical Engineers. ISBN 978-0-7918-5758-8. doi: 10.1115/IMECE2015-52893. URL https://asmedigitalcollection.asme.org/IMECE/proceedings/IMECE2015/57588/Houston,%20Texas,%20USA/262536.

OpenCog Foundation. OpenCog, 2024. URL https://opencog.org/.

ParaPy. How ParaPy helped to get the Flying-V ready for take-off, November 2020. URL https://parapy.nl/2020/11/17/how-parapy-helped-to-get-the-flying-v-ready-for-take-off/.

ParaPy. ParaPy - Knowledge Based Engineering Platform, 2023. URL https://parapy.nl/.

Stefan Plappert, Paul Christoph Gembarski, and Roland Lachmayer. Development of a knowledge-based and collaborative engineering design agent. *Procedia Computer Science*, 207:946–955, January 2022. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2022.09.150.

Michael Ray. Boston Marathon bombing of 2013 | Facts, Date, Victims, Map, & Suspects | Britannica, April 2013. URL https://www.britannica.com/event/Boston-Marathon-bombing-of-2013.

Ben Rich and Leo Janos. *Skunk Works: A Personal Memoir of My Years at Lockheed*. Back Bay Books, February 1996. ISBN 978-0-316-74300-6.

Zinovy Royzen. Application TRIZ in Value Management and Quality Improvement. In *SAVE Annual Proceedings*, volume XXVIII, pages 94–101, Fort Lauderdale, Florida, 1993. URL https://trizconsulting.com/TRIZApplicationinValueManagement.pdf.

Chloe Alex Schaff, Yanjun Li, Bing Ouyang, Casey Den Ouden, Tongdi Zhou, and Fauzia Ahmad. Development of a low-cost subsea datalogger for passive phased sonar arrays. In *Big Data III: Learning, Analytics, and Applications*, volume 11730, Online Only, April 2021. SPIE. doi: https://doi.org/10.1117/12.2587935.

E. J. Schut. *Conceptual Design Automation: Abstraction complexity reduction by feasilisation and knowledge engineering*. Doctoral, Technische Universiteit Delft, 2010. URL https://repository.tudelft.nl/islandora/object/uuid%3A94fd1664-9fdc-4868-b2c6-d977c0fbc2e2. Publisher: E.J. Schut.

Brad Smith. Facial recognition technology: The need for public regulation and corporate responsibility, July 2018. URL https://blogs.microsoft.com/on-the-issues/2018/07/13/facial-recognition-technology-the-need-for-public-regulation-and-corporate-responsibility/.

Tim Smithers. Design as Exploration: Puzzle-Making and Puzzle-Solving. In *Explorationbased models of design and search-based models of design*, Carneige Mellon University, Pittsburgh, June 1992. URL https://www.academia.edu/22991882/Design_as_Exploration_Puzzle_Making_and_Puzzle_Solving.

Bas R. Steunebrink, Kristinn R. Thórisson, and Jürgen Schmidhuber. Growing Recursive Self-Improvers. In Bas Steunebrink, Pei Wang, and Ben Goertzel, editors, *AGI 2016*, volume 9782, pages 129–139, New York, New York, 2016. Springer International Publishing. ISBN 978-3-319-41648-9 978-3-319-41649-6. doi: https://doi.org/10.1007/978-3-319-41649-6_13.

K. R. Thórisson, E. Nivel, B. R. Steunebrink, H. P. Helgason, G. Peluzzo, R. Sanz, J. Schmidhuber, H. Dindo, M. Rodriguez, A. Chella, G. Jonsson, D. Ognibene, and C. Hernandez. Autonomous Acquisition of Situated Natural Communication. *International Journal of Computer Science & Information Systems*, 9(2):115–131, 2014. ISSN 1646-3692. URL https://alumni.media.mit.edu/~kris/ftp/IJCSIS-ThorissonEtAl2015-2014170208.pdf.

Kristinn R. Thórisson. A New Constructivist AI: From Manual Construction to Self-Constructive Systems. *Theoretical Foundations of Artificial General Intelligence*, 4: 145–171, 2012. doi: 10.2991/978-94-91216-62-6_9.

Kristinn R. Thórisson. Seed-Programmed Autonomous General Learning. In *Proceedings of Machine Learning Research*, volume 131, pages 32–70, 2020. URL https://proceedings.mlr.press/v131/thorisson20a.html.

Kristinn R. Thórisson and Eric Nivel. Achieving Artificial General Intelligence Through Peewee Granularity. In *Proceedings of the 2nd Conference on Artificial General Intelligence (2009)*, Arlington, Virginia, USA, 2009. Atlantis Press. ISBN 978-90-78677-24-6. doi: https://doi.org/10.2991/agi.2009.42.

Kristinn R. Thórisson and A. Talbot. Abduction, Deduction & Causal-Relational Models. In *Workshop on Architectures & Evaluation for Generality, Autonomy & Progress in AI*, Stockholm, Sweden, July 2018. URL https://alumni.media.mit.edu/~kris/ftp/AEGAP18_Abduction_Deduction_Causal_Relational_Models.pdf.

Kristinn R. Thórisson, Jordi Bieger, Thröstur Thorarensen, Jóna S. Sigurðardóttir, and Bas R. Steunebrink. Why Artificial Intelligence Needs a Task Theory: And What It Might Look Like. In *AGI 2016*, volume 9782, pages 118–128, New York, New York, 2016a. Springer International Publishing. ISBN 978-3-319-41648-9 978-3-319-41649-6. doi: https://doi.org/10.1007/978-3-319-41649-6_12.

Kristinn R. Thórisson, David Kremelberg, Bas R. Steunebrink, and Eric Nivel. About Understanding. In Bas Steunebrink, Pei Wang, and Ben Goertzel, editors, *AGI 2016*, volume 9782, pages 106–117, New York, New York, 2016b. Springer International Publishing. ISBN 978-3-319-41648-9 978-3-319-41649-6. doi: https://doi.org/10.1007/978-3-319-41649-6_11.

Kristinn R. Thórisson, Jordi Bieger, Xiang Li, and Pei Wang. Cumulative Learning. In Patrick Hammer, Pulin Agrawal, Ben Goertzel, and Matthew Iklé, editors, *Proceedings of the 12th International Conference on Artificial General Intelligence (AGI-19)*, volume 11654, pages 198–209, Shenzen, China, 2019. Springer International Publishing. ISBN 978-3-030-27004-9 978-3-030-27005-6. doi: https://doi.org/10.1007/978-3-030-27005-6_20.

University of Michigan Soar Group. Soar Cognitive Architecture, 2024. URL `https://soar.eecs.umich.edu`.

Srinivasan Venkataraman and Amaresh Chakrabarti. An Integrated Model of Designing. *Journal of Computing and Information Science in Engineering*, 10, September 2010. doi: https://doi.org/10.1115/1.3467011.

Olle Vidner, Camilla Wehlin, Johan A. Persson, and Johan Ölvander. Configuring Customized Products with Design Optimization and Value-Driven Design. *Proceedings of the Design Society*, 1:741–750, August 2021. ISSN 2732-527X. doi: https://doi.org/10.1017/pds.2021.74. Publisher: Cambridge University Press.

Pei Wang. Toward a Unified Artificial Intelligence. In *Papers from the 2004 AAAI Fall Symposium*, page 8, 2004. URL `https://api.semanticscholar.org/CorpusID:2519815`.

Pei Wang. *Rigid Flexibility: The Logic of Intelligence*. Springer Science & Business Media, September 2006. ISBN 978-1-4020-5045-9.

Pei Wang. *Non-Axiomatic Logic: A Model of Intelligent Reasoning*. WORLD SCIENTIFIC, June 2013. ISBN 978-981-4440-27-1 978-981-4440-28-8. https://doi.org/10.1142/8665.

John Wright and Vighnesh Iyer. A Verilog Primer: An Overview of Verilog for Digital Design and Simulation, 2022. URL `https://inst.eecs.berkeley.edu/~eecs151/fa22/files/verilog/Verilog_Primer_Slides.pdf`.

Roman Žavbi and Jože Duhovnik. Conceptual design of technical systems using functions and physical laws. *AI EDAM*, 14(1):69–83, January 2000. ISSN 1469-1760, 0890-0604. doi: https://doi.org/10.1017/S089006040014106X. Publisher: Cambridge University Press.

Appendix A

A Note on AI and EDA Safety

It is worth considering the safety of an engineering design agent (EDA) as envisioned in this thesis. While the intelligence of an EDA may not be at the level of a truly generally intelligent system, it will still possess significant capabilities in the domain of engineering design. This could be considered a form of general design intelligence as a fully developed EDA should be able to work on a wide variety of problems with an ability at or above that of a human engineer. For this reason, an EDA might be considered harboring the potential to become a danger to its environment or users. Could an EDA (as proposed here) become rogue and turn murderous? Could it facilitate the creation of dangerous new technologies and weapons heretofore undreamt of? The following paragraphs explore and address these concerns. Though there can be no guarantee of complete safety, it is believed that an EDA will not pose much of a threat and that the benefits of the technology far outweigh the potential detriments.

First, the question of a rogue AI. It is worth noting that the EDAs developed here will not aim to attain some level of general or human-level intelligence as this level of intelligence is simply not required for engineering design. As posited in Hypothesis 1 (page 6), engineering tasks are a subset of all tasks, this means that they do not require the full spectrum of general problem-solving ability to be worked. Human intelligence has a significant degree of generality but humans are capable of so much more than engineering tasks. The same cannot be said of an EDA; though the goal is to develop EDAs capable of engineering at or above the level of a human engineer, they will only be designed for engineering tasks. They will be powerful artificially-intelligent systems locked into a very specific set of problems and a very constrained understanding of the world. This makes them a sort of restricted domain intelligent system or RDIS. An RDIS is more capable than a machine-learning-based agent but not nearly as intelligent as a human-level or general AI. Additionally, the conceptual nature of an EDA's knowledge and reasoning makes for a highly transparent system whose methods and motives can be interrogated and analyzed whenever desired.

With these features considered, we can see that an EDA does not pose any sort of threat because of its intelligence.

An EDA's engineering capabilities, however, are a different matter. Engineering is a double-edged sword, its problem-solving capabilities have made amazing things possible but have also enabled a substantial amount of human suffering and death. A fully functional EDA with capabilities that meet or exceed those of a human engineer would carry all of this potential with it. When used productively and with good intentions, an EDA could enable humans to solve problems far beyond our current reach, permitting individuals to produce unique solutions to their unique problems and enabling organizations to create engineering marvels that we have only ever dreamt of. The trouble is that there can be no guarantees that an EDA will be used by people with good intentions. With that in mind, it is worth considering the surprising number of dangerous resources individuals have access to; it does not take much more than household materials to develop improvised explosive devices (cf. Ray (2013)) or to create dangerous chemicals (Meredith Hawes, 2020), for example. As a society, we are kept safe by the fact that the knowledge required to produce these weapons is controlled and not well-disseminated. An EDA, however, could change that. An EDA would be able to develop such a weapon from first principles without the need for much external information. Its solution could even be used by someone who would otherwise have no knowledge of these first principles.

Those who study engineering are taught mathematics, a problem-solving mindset, and plenty of domain-specific knowledge. With this comes the ability to misuse that knowledge to dangerous ends, as the same knowledge that can create tools can also create weapons. This knowledge can be considered a sort of data hazard: "specific data, such as the genetic sequence of a lethal pathogen or a blueprint for making a thermonuclear weapon, [that,] if disseminated, create risk." (Bostrom, 2011). As it is, this knowledge is largely sequestered within the works and communities of engineers and is often not easily accessible to the general public. Efforts are even made to actively restrict the dissemination of this kind of information, the most notable example being the extreme secrecy surrounding nuclear arms development. A complete EDA, however, could sidestep all of these limitations and allow any user to easily apply any and all engineering knowledge to any given end. In this way, improperly handled EDA research would not constitute a direct hazard but would facilitate the creation of a data hazard.

With this in mind, it is imperative that there be some consideration with regard to limiting the potentially harmful impacts of this technology, even though the EDA created in this thesis is far too simple to present any form of threat. As it is believed that this research is important and that the potential benefits of EDAs outweigh the risks, a compromise must be found between academic openness and the responsible release of information. For the moment, the EDA architecture, implementation code, and all associated research are public and open source. This also includes basic knowledge bases capable of solving demonstration problems like the one presented in Chapter 6. This may not be the case for more advanced EDAs, however, as these may be restricted to only certified or approved users depending on how potentially dangerous these systems are deemed. One could even foresee a future in which the use of EDAs is regulated by governmental bodies in line with the growing calls to regulate facial recognition technologies in the United States (Smith, 2018). Regardless, more advanced knowledge bases will always be kept private and available only upon request and review. Additionally, no knowledge bases are to be created at all for the purposes of weapons development; EDAs in general will not be allowed to develop weapons or other dangerous technologies. This is not just an EDA issue, the AERA system that lies at the core of this thesis' demonstration system is bound by the *HUMANOBS Open-Source BSD License with CADIA Clause v. 1.0* (Icelandic Institute for Intelligent Machines and Center for Analysis and Design of Intelligent Agents, 2023). This license includes the following text:

> The software may not be used in furtherance of: (i) intentionally causing bodily injury or severe emotional distress to any person; (ii) invading the personal privacy or violating the human rights of any person; or (iii) committing or preparing for any act of war.

As such, there is no legitimate use by individual, corporate, or state actors in the development of weapons technologies with an EDA.

While these solutions are far from perfect, suppressing knowledge bases and prohibiting the engineering of weapons will hopefully be a start to keeping EDAs safe. The ultimate goal of this research project is a design agent with the ability to work alongside human engineers in solving some of the most difficult problems of our time and enabling the creation of some truly beneficial engineering solutions. It is imperative that any developments of EDA technology keep these goals in mind and that efforts are made to refine and improve these safeguards as the technology progresses.

Appendix B

NARS Toothbrush Example Problem

This is the original example that Appendix C's program is based on. It is written in NARSese, the language of the Non-Axiomatic Reasoning System or NARS (Wang, 2004, 2013). It is reproduced here from the Open NARS for Applications GitHub repository (Department of Computer and Information Sciences, Temple University, 2020) where it was originally published as `toothbrush.nal`.

NARS' goal in this example is to use a toothbrush to unscrew a screw, it does this by melting and reshaping the toothbrush with a lighter. To accomplish this, NARS is provided with four operations and a series of facts that describe how those operations can be used to alter an object's state. After these facts, NARS' goal of removing the screw is repeatedly injected with the expectation that, each time, NARS will take the next step towards a solution. It starts by using the lighter to heat the toothbrush and make it pliable. Then, it reshapes the toothbrush until it is shaped like the head of a screw. After this, it waits until the plastic cools and hardens. Finally, it uses its newly-created screwdriver to remove the screw. Given that this is a sort of engineering and problem-solving process, it seemed natural to be the example EDA-0 was tested on in Chapter 6.

Listing B.1: The original NARSese toothbrush example that Appendix C is based on

```
*motorbabbling=false
*volume=0
*setopname 1 ^lighter
*setopname 2 ^reshape
*setopname 3 ^wait
*setopname 4 ^unscrew

//If something is made of plastic, applying the lighter on it will make it heated
<(&/,<(*,{$1},plastic) --> made_of>,<(*,{SELF},{$1}) --> ^lighter>) =/> <{$1} -->
     [heated]>>.
//If it's heated it will be melted
<<{$1} --> [heated]> =/> <{$1} --> [melted]>>.
```

```
//If it's melted it will be pliable
<<{$1} --> [melted]> =/> <{$1} --> [pliable]>>.
//If it's pliable and reshape is applied, it will be screwlike
<(&/,<{$1} --> [pliable]>,<(*,{SELF},{$1}) --> ^reshape>) =/> <{$1} --> [screwlike
    ]>>.
//If it's screwlike and we wait, it will be hardened
<(&/,<{$1} --> [screwlike]>,<(*,{SELF},{$1}) --> ^wait>) =/> <{$1} --> [hardened
    ]>>.
//If it's hardened, we can use it for unscrewing to remove the screw
<(&/,<{#1} --> [hardened]>,<(*,{SELF},{#1}) --> ^unscrew>) =/> <{screw1} --> [
    removed]>>.
<plastic --> [bendable]>.

//let's try an example:
//toothbrush1 is madeof bendable material
<(*,{toothbrush1},[bendable]) --> made_of>. :|:
10
//remove the screw1!
<{screw1} --> [removed]>! :|:
10
//expected: ^lighter executed with args ({SELF} * {toothbrush1})
30
//some time later:
//toothbrush1 is pliable
<{toothbrush1} --> [pliable]>. :|:
//remove the screw1!
<{screw1} --> [removed]>! :|:
10
//expected: ^reshape executed with args ({SELF} * {toothbrush1})
30
//some time later:
//toothbrush1 is screwlike
<{toothbrush1} --> [screwlike]>. :|:
//remove the screw1!
<{screw1} --> [removed]>! :|:
10
//expected: ^wait executed with args ({SELF} * {toothbrush1})
30
//some time later:
//toothbrus1 is hardened
<{toothbrush1} --> [hardened]>. :|:
//remove the screw1!
<{screw1} --> [removed]>! :|:
10
//expected: ^unscrew executed with args ({SELF} * {toothbrush1})
```

Appendix C

Toothbrush Problem Seed Program

This is the full seed program used in the toothbrush problem demonstrated in Chapter 6. It is programmed in Replicode with the POODL shown in the comment at the top of the file.

Listing C.1: The full seed program for Chapter 6's toothbrush example

```
; AERA Melts a Toothbrush
; Chloe Schaff 2023-09-27
;
; Based on the toothbrush-twostep program. This includes the full sequence through
     heating,
; melting, forming, and then unscrewing. it is a replica of the NARS toothbrush.
    nal example
; but implemented in Replicode with O&O-1 and POODL as guidelines. In POODL, the
    solution
; should look like this:
; (unscrew "screw" WITH "toothbrush" UNTIL "unscrewed" [
; (wait "toothbrush" WITH "pause" UNTIL "cooled" [
; (reshape "toothbrush" WITH "form" UNTIL "screwdriver-shaped" [
; (heat "toothbrush" WITH "lighter" UNTIL "heated" [])
; ])
; ])
; ])
;
; *Note the "pause" tool in the wait command. Every operation requires a tool in
    POODL
; so even an operation like waiting around for something to cool needs a
    placeholder
; tool like a pause even if a pause isn't a physical thing you can manipulate.
; There's probably a better way to do this.
;
; The POODL prompt for this would be something like this (where "?" refers to
    steps or
; objects the agent may fill in):
```

```
; (unscrew "screw" WITH "?" UNTIL "unscrewed" [?])

; Set up the objects and ontologies
screw:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
toothbrush:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
lighter:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
form:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
pause:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
state:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
s:(ent 1) [[SYNC_ONCE now 1 forever root nil]] ; A screw
t:(ent 1) [[SYNC_ONCE now 1 forever root nil]] ; A toothbrush
l:(ent 1) [[SYNC_ONCE now 1 forever root nil]] ; A lighter
f:(ent 1) [[SYNC_ONCE now 1 forever root nil]] ; A form
p:(ent 1) [[SYNC_ONCE now 1 forever root nil]] ; A pause (an imaginary tool that
     takes up time but doesn't do anything)

; Set some facts about the objects
s_is_a_screw:(mk.val s essence screw 1) |[]
(fact s_is_a_screw 0s:0ms:0us GIGASEC 1 1) [[SYNC_AXIOM now 1 forever stdin nil]]
t_is_a_toothbrush:(mk.val t essence toothbrush 1) |[]
(fact t_is_a_toothbrush 0s:0ms:0us GIGASEC 1 1) [[SYNC_AXIOM now 1 forever stdin
     nil]]
l_is_a_lighter:(mk.val l essence lighter 1) |[]
(fact l_is_a_lighter 0s:0ms:0us GIGASEC 1 1) [[SYNC_AXIOM now 1 forever stdin nil
     ]]
f_is_a_form:(mk.val f essence form 1) |[]
(fact f_is_a_form 0s:0ms:0us GIGASEC 1 1) [[SYNC_AXIOM now 1 forever stdin nil]]
p_is_a_pause:(mk.val p essence pause 1) |[]
(fact p_is_a_pause 0s:0ms:0us GIGASEC 1 1) [[SYNC_AXIOM now 1 forever stdin nil]]

; Models and CSTs
; S3: I have a screw and a cooled toothbrush
; S2: I have a toothbrush that's screwdriver-shaped
; S1: I have a toothbrush that's heated
; S0: I have a toothbrush that's shaped like a toothbrush

; UNSCREW: If we use a cooled toothbrush on a screw, it will become unscrewed
; UNSCREW_req: We can use UNSCREW if we satisfy S3
; WAIT: If we use wait (use nothing) on a screwdriver-shaped toothbrush, it will
     become cooled
; WAIT_req: We can use WAIT if we satisfy S2
; FORM: If we use a form on a heated toothbrush, it will become screwdriver-shaped
; FORM_req: We can use FORM if we satisfy S1
; HEAT: If we use a lighter on a toothbrush, it will become heated
; HEAT_req: We can use HEAT if we satisfy S0

; === Set up the CSTs ===
; S3: I have a screw and a cooled toothbrush
S3:(cst [] []
  (fact (mk.val T: essence toothbrush :) T0: T1: : :) ; There exists a toothbrush
  (fact (mk.val T: state "cooled" :) T0: T1: : :) ; that is cooled
  (fact (mk.val S: essence screw :) T0: T1: : :) ; and there exists a screw
  (fact (mk.val S: state "screwed in" :) T0: T1: : :) ; that is screwed in
|[]
|[]
[stdin] 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; S2: I have a toothbrush that's screwdriver-shaped
S2:(cst [] []
  (fact (mk.val P: essence pause :) T0: T1: : :) ; The tool we're using is a pause
  (fact (mk.val T: essence toothbrush :) T0: T1: : :) ; and there exists a
       toothbrush
  (fact (mk.val T: state "screwdriver-shaped" :) T0: T1: : :) ; that is
       screwdriver-shaped
|[]
|[]
[stdin] 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; S1: I have a toothbrush that's heated
S1:(cst [] []
  (fact (mk.val F: essence form :) T0: T1: : :) ; There exists a form
  (fact (mk.val T: essence toothbrush :) T0: T1: : :) ; and there exists a
       toothbrush
  (fact (mk.val T: state "heated" :) T0: T1: : :) ; that is heated
|[]
|[]
[stdin] 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; S0: I have a toothbrush that's shaped like a toothbrush
S0:(cst [] []
  (fact (mk.val L: essence lighter :) T0: T1: : :) ; There exists a lighter
  (fact (mk.val T: essence toothbrush :) T0: T1: : :) ; and there exists a
       toothbrush
  (fact (mk.val T: state "toothbrush" :) T0: T1: : :) ; that looks like a
       toothbrush
|[]
|[]
[stdin] 1) [[SYNC_ONCE now 0 forever primary nil 1]]


; === Set up the models ===
; UNSCREW: If we use a cooled toothbrush on a screw, it will become unscrewed
UNSCREW:(mdl [T: S: (ti T0: T1:)] []
  (fact (cmd use_tool [T: S:] :) T2: T1_cmd: : :)
  (fact (mk.val S: state "unscrewed" 1) T1_RHS: T3: : :)
[]
  T1_RHS:(+ T0 100ms)
  T3:(+ T1 100ms)
[]
  T2:(- T1_RHS 80ms)
  T1_cmd:(- T3 100ms)
```

```
   T0:(- T1_RHS 100ms)
   T1:(- T3 100ms)
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; UNSCREW_req: We can use UNSCREW if we satisfy S3
UNSCREW_req:(mdl [] []
   (fact (icst S3 [] [T: S:] : :) T0: T1: : :)
   (fact (imdl UNSCREW [T: S: (ti T0: T1:)] [T1_RHS: T3:] : :) T0: T1: : :)
|[]
|[]
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]


; WAIT: If we use wait (use nothing) on a screwdriver-shaped toothbrush, it will
     become cooled
WAIT:(mdl [P: T: (ti T0: T1:)] []
   (fact (cmd use_tool [P: T:] :) T2: T1_cmd: : :)
   (fact (mk.val T: state "cooled" 1) T1_RHS: T3: : :)
[]
   T1_RHS:(+ T0 100ms)
   T3:(+ T1 100ms)
[]
   T2:(- T1_RHS 80ms)
   T1_cmd:(- T3 100ms)
   T0:(- T1_RHS 100ms)
   T1:(- T3 100ms)
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; WAIT_req: We can use WAIT if we satisfy S2
WAIT_req:(mdl [] []
   (fact (icst S2 [] [P: T:] : :) T0: T1: : :)
   (fact (imdl WAIT [P: T: (ti T0: T1:)] [T1_RHS: T3:] : :) T0: T1: : :)
|[]
|[]
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]


; FORM: If we use a form on a heated toothbrush, it will become screwdriver-shaped
FORM:(mdl [F: T: (ti T0: T1:)] []
   (fact (cmd use_tool [F: T:] :) T2: T1_cmd: : :)
   (fact (mk.val T: state "screwdriver-shaped" 1) T1_RHS: T3: : :)
[]
   T1_RHS:(+ T0 100ms)
   T3:(+ T1 100ms)
[]
   T2:(- T1_RHS 80ms)
   T1_cmd:(- T3 100ms)
   T0:(- T1_RHS 100ms)
   T1:(- T3 100ms)
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; FORM_req: We can use FORM if we satisfy S1
```

```
FORM_req:(mdl [] []
   (fact (icst S1 [] [F: T:] : :) T0: T1: : :)
   (fact (imdl FORM [F: T: (ti T0: T1:)] [T1_RHS: T3:] : :) T0: T1: : :)
|[]
|[]
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]


; HEAT: If we use a lighter on a toothbrush, it will become heated
HEAT:(mdl [L: T: (ti T0: T1:)] []
   (fact (cmd use_tool [L: T:] :) T2: T1_cmd: : :)
   (fact (mk.val T: state "heated" 1) T1_RHS: T3: : :)
[]
   T1_RHS:(+ T0 100ms)
   T3:(+ T1 100ms)
[]
   T2:(- T1_RHS 80ms)
   T1_cmd:(- T3 100ms)
   T0:(- T1_RHS 100ms)
   T1:(- T3 100ms)
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]

; HEAT_req: We can use HEAT if we satisfy S0
HEAT_req:(mdl [] []
   (fact (icst S0 [] [L: T:] : :) T0: T1: : :)
   (fact (imdl HEAT [L: T: (ti T0: T1:)] [T1_RHS: T3:] : :) T0: T1: : :)
|[]
|[]
[stdin] 1 1 1 1 1) [[SYNC_ONCE now 0 forever primary nil 1]]


; === Emulator Setup ===
; Set up the emulator state and inject it
; emulator_state: [Is the screw unscrewed?, What is the state of the toothbrush?]
e1:(ent 1) [[SYNC_ONCE now 1 forever root nil]]
emulator_state:(ont 1) [[SYNC_ONCE now 0 forever root nil]]
emulator:(std_grp 2 0 0 0 []) [[SYNC_ONCE now 0 forever root nil COV_OFF 0]]

; This program runs in the primary group to re-inject any (fact (goal (fact (cmd
     ::)))) into emulator.
pgm_inject_in_emulator_group:(pgm [] []
   (ptn f_G:(fact G:(goal (fact (cmd ::) ::) ::) ::) [])
[]
   ; Only re-inject non-simulation goals.
   (= (is_sim G) false)
[]
   (inj [f_G [SYNC_ONCE (now) 1 1 emulator nil]])
1) |[]
(ipgm pgm_inject_in_emulator_group [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever primary nil 1]
```

```
; === Drive Setup and Maintenance ===
!def DRIVE_START 300ms
start:(pgm [] [] [] []
   (inj [State:(mk.val e1 emulator_state [false "toothbrush"] 1) []])
   (inj []
      (fact State After:(now) (+ After sampling_period) 1 1)
      [SYNC_PERIODIC now 1 1 emulator nil]
   )
1) |[]
(ipgm start [] RUN_ONCE sampling_period VOLATILE NOTIFY 1) [[SYNC_ONCE now 0
     forever stdin nil 1]]

m_drive:(mdl [] []
   ; The goal target timings are the same as the drive timings.
   (fact (mk.val s state "unscrewed" 1) T0: T1: 1 1)
   (fact run T0: T1: ::)
|[]
|[]
[stdin drives] 1 1 1 0 1) [[SYNC_ONCE now 0 forever primary nil 1]]

pgm_inject_drive:(pgm [] []
   ; This fact repeats periodically. We use it as a "heartbeat".
   (ptn (fact (mk.val s essence screw :) After: Before: ::) [])
[]
   (>= After (+ this.vw.ijt DRIVE_START))
[]
   ; The end of the time interval will be used in m_drive as the end of the goal
        interval.
   (inj [f_run:(fact run After (+ Before 500ms) 1 1) []])
   (inj [G:(goal f_run self nil 1) []])
   (inj []
      ; Delay a little to allow predictions for this sampling period before
           injecting the drive.
      (fact G T0:(+ After 10ms) T0 1 1)
      [SYNC_ONCE T0 1 forever primary nil]
   )
   (prb [1 "print" "injected drive" []])
1) |[]
(ipgm pgm_inject_drive [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever primary nil 1]

; Before DRIVE_START, just re-inject the same values.
pgm_before_drive:(pgm [] []
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: Shaped:] :) After: Before: ::)
        [])
[]
   (< After (+ this.vw.ijt DRIVE_START))
[]
   ; All state variables keep the same value.
   (inj [Next_state:(mk.val e1 emulator_state [Unscrewed Shaped] 1) []])
   (inj []
      (fact Next_state (+ After sampling_period) (+ Before sampling_period) 1 1)
      [SYNC_PERIODIC (+ After sampling_period) 1 1 emulator nil]
   )
1) |[]
(ipgm pgm_before_drive [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]


; === Emulator Interaction ===
; Unscrew something with a screwdriver-shaped object
pgm_cmd_unscrew:(pgm [] []
   (ptn (fact G:(goal (fact Command:(cmd use_tool [t s] ::) Cmd_after: Cmd_before:
        ::) ::) ::) []))
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: State:] :) After: Before: ::)
        [])
[]
   (= (is_sim G) false)
   (= Unscrewed false)
   (= State "cooled")
   (< Cmd_after Before)
   (> Cmd_before After)
[]
   ; Inject the fact that the command was executed.
   (inj []
      (fact Command (+ After 20ms) Before 1 1)
      [SYNC_ONCE After 1 1 stdin nil]
   )
   ; The screw is now unscrewed
   (inj [Next_state:(mk.val e1 emulator_state [true "cooled"] 1) []])
   (inj []
      (fact Next_state (+ After sampling_period) (+ Before sampling_period) 1 1)
      [SYNC_PERIODIC (+ After sampling_period) 1 1 emulator nil]
   )
   (prb [1 "print" "pgm_cmd_unscrew from command:" [Command]])
1) |[]
(ipgm pgm_cmd_unscrew [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]

pgm_cmd_wait:(pgm [] []
   (ptn (fact G:(goal (fact Command:(cmd use_tool [p t] ::) Cmd_after: Cmd_before:
        ::) ::) ::) []))
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: State:] :) After: Before: ::)
        [])
[]
   (= (is_sim G) false)
   (= State "screwdriver-shaped")
   (< Cmd_after Before)
   (> Cmd_before After)
[]
   ; Inject the fact that the command was executed.
   (inj []
      (fact Command (+ After 20ms) Before 1 1)
      [SYNC_ONCE After 1 1 stdin nil]
   )
   ; The screwdriver is now cooled
   (inj [Next_state:(mk.val e1 emulator_state [Unscrewed "cooled"] 1) []])
   (inj []
      (fact Next_state (+ After sampling_period) (+ Before sampling_period) 1 1)
      [SYNC_PERIODIC (+ After sampling_period) 1 1 emulator nil]
   )
   (prb [1 "print" "pgm_cmd_wait from command:" [Command]])
1) |[]
(ipgm pgm_cmd_wait [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]

pgm_cmd_form:(pgm [] []
   (ptn (fact G:(goal (fact Command:(cmd use_tool [f t] ::) Cmd_after: Cmd_before:
        ::) ::) ::) [])
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: State:] :) After: Before: ::)
        [])
[]
   (= (is_sim G) false)
   (= State "heated")
   (< Cmd_after Before)
   (> Cmd_before After)
[]
   ; Inject the fact that the command was executed.
   (inj []
      (fact Command (+ After 20ms) Before 1 1)
      [SYNC_ONCE After 1 1 stdin nil]
   )
   ; The toothbrush is now screwdriver-shaped
   (inj [Next_state:(mk.val e1 emulator_state [Unscrewed "screwdriver-shaped"] 1)
        []])
   (inj []
      (fact Next_state (+ After sampling_period) (+ Before sampling_period) 1 1)
      [SYNC_PERIODIC (+ After sampling_period) 1 1 emulator nil]
   )
   (prb [1 "print" "pgm_cmd_form from command:" [Command]])
1) |[]
(ipgm pgm_cmd_form [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]

pgm_cmd_heat:(pgm [] []
   (ptn (fact G:(goal (fact Command:(cmd use_tool [l t] ::) Cmd_after: Cmd_before:
        ::) ::) ::) [])
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: State:] :) After: Before: ::)
        [])
[]
   (= (is_sim G) false)
   (= State "toothbrush") ; A little unnecessary, we can always heat and re-form
   (< Cmd_after Before)
   (> Cmd_before After)
[]
   ; Inject the fact that the command was executed.
   (inj []
      (fact Command (+ After 20ms) Before 1 1)
      [SYNC_ONCE After 1 1 stdin nil]
   )
   ; The toothbrush is now heated
   (inj [Next_state:(mk.val e1 emulator_state [Unscrewed "heated"] 1) []])
   (inj []
      (fact Next_state (+ After sampling_period) (+ Before sampling_period) 1 1)
      [SYNC_PERIODIC (+ After sampling_period) 1 1 emulator nil]
   )
   (prb [1 "print" "pgm_cmd_heat from command:" [Command]])
1) |[]
(ipgm pgm_cmd_heat [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]


; === Inject emulator states ===
; Reinject that the screw is still screwed in (if that’s the case)
pgm_inject_mk_vals_screwedin:(pgm [] []
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: Shaped:] :) After: Before: ::)
        [])
[]
   (= Unscrewed false)
[]
   (inj [Val_state:(mk.val s state "screwed in" 1) []])
   (inj []
      (fact Val_state After Before 1 1)
      [SYNC_PERIODIC After 1 1 stdin nil]
   )
   (prb [1 "print" "s is screwed in" []])
1) |[]
(ipgm pgm_inject_mk_vals_screwedin [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]

; Reinject that the screw is unscrewed (if that’s the case)
pgm_inject_mk_vals_unscrewed:(pgm [] []
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: Shaped:] :) After: Before: ::)
        [])
[]
   (= Unscrewed true)
[]
   (inj [Val_state:(mk.val s state "unscrewed" 1) []])
   (inj []
      (fact Val_state After Before 1 1)
      [SYNC_PERIODIC After 1 1 stdin nil]
   )
   (prb [1 "print" "s is unscrewed" []])
1) |[]
(ipgm pgm_inject_mk_vals_unscrewed [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]

; Reinject the current state of the toothbrush
```

```
pgm_inject_mk_vals_toothbrush:(pgm [] []
   (ptn (fact (mk.val e1 emulator_state [Unscrewed: State:] :) After: Before: ::)
        [])
|[]
[]
   (inj [Val_state:(mk.val t state State 1) []])
   (inj []
      (fact Val_state After Before 1 1)
      [SYNC_PERIODIC After 1 1 stdin nil]
   )
1) |[]
(ipgm pgm_inject_mk_vals_toothbrush [] RUN_ALWAYS MAX_TIME VOLATILE NOTIFY 1) []
   [SYNC_ONCE now 0 forever emulator nil 1]
```