
Pröstur Thorarensen (throstur11@ru.is)

June 1, 2016
Abstract

Generally intelligent robots and systems should be evaluated based on their ability to learn new tasks over a wide range of domains. Few if any of the available evaluation methods for artificial intelligence (AI) systems address this need, and most even leave out important aspects of intelligence, such as a system’s ability to learn. As a result, ad-hoc methods of evaluation are commonly used, and no standardized evaluation methods have been accepted. Furthermore, evaluation of controllers in physically realistic task-environments has been left mostly unaddressed. In short, there are vast opportunities for improvement in the way AI systems are evaluated. However, not all AI systems are alike or created equal. This could be addressed if we had a toolkit where developers could easily construct appropriate tasks for evaluating and comparing their systems on a variety of tasks. To be generally applicable such a toolkit should provide answers about the efficiency, both in time and energy, of various control systems, so that they could be ordered with respect to their practical utility in the most general way possible.

In this thesis we present a prototype framework that allows modular construction of task-environments, rooted in physics, and its early-state implementation, the Framework for Modular Task-Environment Construction (FraMoTEC). Simulation is used to evaluate control systems’ performances in terms of expended time and energy. In our approach tasks are dissected into dimensions to be controlled by the system to be evaluated; simpler tasks contain only a few dimensions to be controlled sequentially; more complex tasks have a large number of dimensions, some of which must be controlled simultaneously to achieve the task. In FraMoTEC components can be flexibly modified and changed through the inherent modularity, allowing evaluating control systems on a single or multiple tasks, as well as on a family of tasks.

The utility of FraMoTEC as an AI evaluation framework was demonstrated by evaluating the performance of various controllers (such as SARSA reinforcement learners) on a collection of task-environments, using both simple tasks and a suite of scalable N-dimensional tasks. The results show that FraMoTEC allows both simple and complex state of the art controllers to be flexibly evaluated on a family of physical tasks in a straightforward manner. The evaluation can be along the dimensions of efficiency (time, energy, or both), failures, learning rate, etc. and any combination thereof. Further theoretical analysis of N-dimensional tasks indicates the approach can scale to be suitable for advanced controllers with higher levels of intelligence, learning capacity, etc., making the approach a promising direction to pursue in the context of AI and robotics evaluation.
## Contents

1 Introduction .................................................. 5  
  1.1 Background .............................................. 5  
  1.2 Motivation .................................................... 7  
  1.3 Summary ..................................................... 8  

2 Related work ................................................. 10  
  2.1 Systems and Control ........................................... 10  
  2.2 Simulation ..................................................... 11  
  2.3 Evaluation Methods for AI Systems ........................... 12  
  2.4 Foundations for a Task Theory ............................... 14  
  2.5 Candidate Tools .............................................. 15  

3 Components of a Framework for Evaluating Task Performance 17  
  3.1 The Framework ............................................... 17  
     3.1.1 Building blocks ........................................ 17  
  3.2 Task Construction ............................................ 19  
     3.2.1 Examples ............................................... 20  
  3.3 Object Physics ............................................... 22  
  3.4 Simulation .................................................... 24  
  3.5 Summary and Discussion ..................................... 24  

4 Software Implementation ..................................... 26  
  4.1 Framework for Modular Task Environment Construction .... 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 Construction and (de)composition</td>
<td>26</td>
</tr>
<tr>
<td>4.2.1 Building Block Implementations</td>
<td>27</td>
</tr>
<tr>
<td>4.2.2 Simulation</td>
<td>29</td>
</tr>
<tr>
<td>4.3 Interface to controllers</td>
<td>29</td>
</tr>
<tr>
<td>4.4 Modeling Examples</td>
<td>30</td>
</tr>
<tr>
<td>4.4.1 Environments</td>
<td>30</td>
</tr>
<tr>
<td>4.4.2 Custom Systems</td>
<td>31</td>
</tr>
<tr>
<td>4.5 Supporting Tools</td>
<td>32</td>
</tr>
<tr>
<td>4.6 Summary and Discussion</td>
<td>32</td>
</tr>
<tr>
<td>5 Use Cases</td>
<td>35</td>
</tr>
<tr>
<td>5.1 Q-Learning / SARSA</td>
<td>36</td>
</tr>
<tr>
<td>5.1.1 Agent Implementation</td>
<td>36</td>
</tr>
<tr>
<td>5.1.2 Task-Environments</td>
<td>36</td>
</tr>
<tr>
<td>5.1.3 Evaluation</td>
<td>38</td>
</tr>
<tr>
<td>5.2 Agent Comparison</td>
<td>42</td>
</tr>
<tr>
<td>5.2.1 Controller</td>
<td>42</td>
</tr>
<tr>
<td>5.2.2 Results</td>
<td>46</td>
</tr>
<tr>
<td>5.3 N-Dimensional Task Comparison</td>
<td>46</td>
</tr>
<tr>
<td>5.3.1 Task-Environment Domain</td>
<td>46</td>
</tr>
<tr>
<td>5.3.2 Controller</td>
<td>46</td>
</tr>
<tr>
<td>5.3.3 Results</td>
<td>47</td>
</tr>
<tr>
<td>5.4 Summary</td>
<td>51</td>
</tr>
<tr>
<td>6 Conclusions and Future Work</td>
<td>52</td>
</tr>
<tr>
<td>6.1 Conclusions</td>
<td>52</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>53</td>
</tr>
<tr>
<td>6.2.1 Simulation</td>
<td>53</td>
</tr>
<tr>
<td>6.2.2 High-level Description Language</td>
<td>53</td>
</tr>
<tr>
<td>6.2.3 Supporting Tools</td>
<td>53</td>
</tr>
</tbody>
</table>
Acknowledgments

I would like to thank my thesis adviser Dr. Kristinn R. Thórisson at Reykjavik University and his Ph.D. student Jordi Bieger. We met for numerous team sessions (discussing a theoretical task theory, among other things), during which much of the work and many of the ideas leading up to this thesis were discussed.
Chapter 1

Introduction

1.1 Background

The world is full of tasks of all kinds. Children are often assigned simple tasks such as taking out the trash or feeding the family pet; as they get older they learn about other tasks such as loading and unloading the dishwasher and moving the car out of the driveway. The general rule of thumb is that younger individuals are assigned simpler tasks than older individuals, as young individuals may not have the necessary experience to complete a task considered more suitable for an older individual. This has to do with the life-long learning of the individuals as they gain various problem-solving abilities and as their physical capabilities increase. It is relatively easy to teach a 6 year old to take a bag from mother’s hand and place it in the trash (assuming no physical limitations), but a 6 year old will probably not have the necessary skills required to meticulously detail grandfather’s sports car — a task more suitable for adolescents.

This begs the question on how can we assess whether or not someone (— or something) is capable of doing some task — one way would be to simply put the individual to the test, a potentially risky approach (imagine tasks like nuclear safety inspection or firefighting). We could instead test the individual in a more controlled environment — whether that be a written examination, a virtual demonstration or something else. In this way we do not place any individuals in danger despite the target task being inherently dangerous.

Ultimately, artificially intelligent control systems will be used for controlling real-world systems (e.g. robots). These systems will need to account for resource expenditure, as they will be expected to solve tasks in a limited amount of time without exceeding some energy allowance.

In this thesis we will address the construction of environments in which artificially intelligent systems can be evaluated (both in terms of time and energy) with simulation. We call such environments task-environments\(^1\) and those interacting with it agents.

Humans are an example of an intelligent system that we have studied for a long time, and we already have some general ideas on how to evaluate a human’s capabilities. Evaluating humans is greatly simplified by the intuitive understanding that humans have of their own sensors and effectors, but this is a luxury that does not necessarily carry over to artificially intelligent systems.

\(^1\)We use the term task-environment to refer to the pairing of a task with the environment in which it must be accomplished; in some cases simply task or environment could be used in its place.
Despite the existence of several methods that can be used to evaluate intelligent systems, all existing methods have fundamental problems. The Turing Test\(^2\) was proposed by Alan Turing in 1950 and was the first proposal for how to evaluate intelligence in a machine [Turing, 1950]. Unfortunately, the machines most likely to come close to passing the Turing test do not come close to being intelligent. Passing the Turing Test can say nothing about a system’s ability to learn, as it could have learned the solution to every sub-goal prior to taking the test or had the knowledge programmed in — although the Turing Test can be useful, it should not be used as a benchmark. Other evaluation methods may be slightly more suited for benchmarking, AI evaluation has focused on checking whether machines can do tasks well instead of evaluating whether the AI is intelligent [Hernández-Orallo, 2014]. The Piaget-MacGyver Room problem [Bringsjord and Licato, 2012], Lovelace Test 2.0 [Riedl, 2014] and Toy Box problem [Johnston, 2010] all come with the caveat of being defined very vaguely — these evaluation methods may be likely to come up with a reasonable evaluation for intelligence, but it is very difficult to compare two different agents (or controllers) that partake in the their own domain-specific evaluations, which is what frequently happens when agents are tailored to pass specific evaluations.

In essence, most of the evaluation methods currently at our disposal leave out important aspects of intelligence, such as evaluating a system’s ability to learn. Evaluation methods should evaluate controllers without changing the task-environment so that the performance of the controller performing the task can be quantified. Quantifiably evaluating different systems in the same task-environment allows us to compare the systems critically and to judge the effects of slight adjustments to the systems more easily. Being able to create abstract versions of various classes of tasks enables yet another level of control in evaluating automatic learners and artificial intelligence systems [Thórisson et al., 2016].

In this thesis we present a tool for constructing task-environments and evaluating adaptive control systems regulating those task-environments, which we will call FraMoTEC: a Framework for Modular Task-Environment Construction. This framework will simulate control systems in task-environments to evaluate the time and energy expenditure of the control system solving the task.

AI systems interact with “environments” that contain all relevant existing objects and the rules by which they interact, while “tasks” assigned to an agent describe specific environment states that should be brought about or avoided. The term task-environment will often be used when we refer to the tuple of a task (which can itself be a set of tasks) and the environment in which it must be accomplished [Thórisson et al., 2016]. An important consideration here is that although the bodies of agents are *embedded* in the environment, their control systems (their “minds”) are not. This implies that any evaluation actually evaluates the collective control strategy of all agents. Figure 1.1 demonstrates the difference between these terms and their relationships. An environment is constructed with small atomic elements such as objects, transitions, sensors and motors, and some containers such as what we simply call “systems”. One could argue that only objects and transitions are necessary to construct environments but convenience components such as sensors and motors also have advantages such as facilitating interfacing — we could for instance imagine a MAPE-K\(^3\) controller interfacing with an environment created by FraMoTEC simply by connecting the sensors and motors of each software. In addition to this, the system component aides task designers in designing tasks at varying levels of abstraction.

---

2 Alan Turing’s intention of the Turing Test was not to answer whether machines can think (or whether they are truly *intelligent*), but rather digital machines could conceivably make humans *believe* they are intelligent by succeeding in “the imitation game”.  
3 [IBM et al., 2006].
1.2 Motivation

A strong initial motivating factor for constructing a framework for evaluating AI and AGI systems is the potential to push AGI in the right direction — adequate evaluation methods will facilitate optimization of cognitive architectures and AGI-aspiring systems. When AGI systems become feasible, we can start imagining a myriad of uses for them that could improve human quality of life significantly.

Here we view agents, AGI-aspiring systems, and other regulators as “controllers”, as we are interested in evaluating the control system of the agent and not the specifics of their embodiment, for example, whether the agent has prongs or suction cups; the embodiment of the controllers we want to evaluate are considered part of the task environments, and thus the controllers can be viewed as being “embedded” in their task-environments.

Microsoft’s Malmo can be used to implement tasks within Minecraft; it has been used to evaluate and compare several control strategies in challenging tasks such as visual hill climbing. Abel et al., 2016. Projects such as Malmo demonstrate a need for easy task-environment construction to facilitate evaluations. However, the focus is on improving upon reinforcement learning algorithms and experimenting in visual domains, not on task-environment construction. The contribution outlined in this paper aims to improve the status quo of evaluation methods, supporting faster research and development in the field by focusing on facilitating task-environment construction.

A prerequisite to evaluating systems in task-environments is the construction of task-environments. Conant and Ashby, 1970 elegantly illustrates that a successful regulator implies that a sufficiently similar model must have been built (implicitly or explicitly). Therefore, any AGI-aspiring system and any agent that are able to successfully solve the problems presented in a task-environment must have
built some kind of model of the environment. By simulating controllers in environment models, we can compare the performance of different controllers in similar (or different) environments. Furthermore, environment models should be complete representations of the target environment while retaining some grounding with the physical world. Tasks and environments in the physical world are composed of smaller parts, often spanning 2 or 3 orders of magnitude (on the time and/or space dimensions), and thus a toolkit for modeling task-environments in simulation would better support that kind of modularity.

As highlighted in Hernández-Orallo [2014], there is still a huge margin of improvement in the way AI systems are evaluated. Ad-hoc methods of evaluation are commonly used and no standardized evaluation methods have been employed over a vast range of AI-aspiring agents. Furthermore, evaluation of controllers in physically realistic task-environments has not been properly addressed. A major motivation for the work outlined in this thesis is the need for a framework that would allow multiple kinds of agent controllers to be evaluated in continuous task-environments. Hopefully, future extensions will allow the generation of a varied set of task-environments that will even allow for certain cognitive abilities to be probed. Cognitive abilities are defined as a property of individuals that allows them to perform well in a range of information-processing tasks [Hernández-Orallo, 2014].

Toolkits exist for evaluating reinforcement learning algorithms, such as rllab. Such toolkits facilitate reinforcement learning development by enabling benchmark evaluations such as the suite described in Duan et al. [2016]. OpenAI’s Gym is a new toolkit for developing and comparing reinforcement learning algorithms which contains a large collection of various kinds of environments. These toolkits demonstrate the need for better evaluation tools.

Thórisson et al. [2015] lists a collection of properties of flexible intelligence evaluation frameworks considered to be important, putting emphasis on easy construction of task-environments and variants. Being able to compose or decompose environments, as well as scale or tune them facilitates construction and increases potential for procedural generation of task-environments. Different task-environments could be compared by decomposing them into the smallest structures and analyzing the relationships between them, but this requires some task theory as a guide. This is something that FraMoTEC aims to facilitate by enabling construction based on building block composition.

Thórisson et al. [2016] contains further motivations for this thesis, touching on evaluation of intelligent systems and the importance of a task theory. A task theory would enable critical comparison of task-environments and with some grounding in physics we can formalize simulable tasks by constructing them from simulable granules. This would also help us expand, simplify or modify task environments in systematic ways — see section 2.4 for related work on task theory.

Expanding on this, there is great motivation for a tool that would allow the evaluation of agents in task-environments by simulation. Such a tool could allow for plotting of evaluation results that visually demonstrates time and energy requirements for agents in certain task-environments. The plot should contain information about the maximum time and energy and the agents’ resource management. Section 4.5 covers plotting in more detail.

1.3 Summary

There is still a huge margin of improvement in the way AI systems are evaluated [Hernández-Orallo 2014]. All current evaluation methods have fundamental problems, leaving out important aspects of intelligence. AI systems will ultimately be used to control real-world systems that will need to manage resource expenditure. This is something that has not been addressed. This thesis is motivated by
the need for a tool that can be used to construct task-environments and evaluate controllers’ resource management in solving those task-environments. Thórisson et al. [2015] provides an excellent reference as a guide towards the creation of such a tool.

The structure of this thesis is as follows: Chapter 1 covers introduction and motivations. Chapter 2 presents related work, touching on some foundations for a task theory. Chapter 3 outlines some applications for task theory, and proposes a prototype for a framework that can be used to evaluate and compare agents, environments and tasks. Chapter 4 focuses on the prototype framework introduced in chapter 3 going into details about the implementation and a discussion about its current state. Chapter 5 demonstrates use cases for the proposed framework, comparing various controllers in several task-environments and diving deeper into practical applications. Finally, chapter 6 concludes the thesis and outlines future work.
Chapter 2

Related work

We will look at the relevant prior work from three main angles — cybernetics, including modeling, controllers and regulation, simulation and Newtonian Physics, evaluation methods and lastly recent work on a task theory for AI.

2.1 Systems and Control

Cybernetics is concerned with those properties of systems that are independent of their concrete material or components. This is at the very heart of modeling and abstraction and allows cybernetics to describe physically different systems with the same concepts and to look for isomorphisms between them [Heylighen and Joslyn 2001]. Heylighen and Joslyn [2001] further discusses entropy and modeling dynamics, but goes into great detail on circular processes and control theory, concluding on the topic of cognition a discussion on the relationship between learning and model building. As we are interested in being able to construct any task-environment from the set of simulable task-environments, we should ensure that all mechanisms of regulation are possible in task-environments constructed with the framework to fully cover the set of all possible regulators for any given task-environment.

As illustrated in [Scholten 2011] which revisits the work of [Conant and Ashby 1970], models are everywhere in science, using examples ranging from linear algebra to abstract concepts. Conant and Ashby prove a theorem stating that all good regulators must model the systems that they regulate, more specifically that the simplest optimal regulator of a system models the system it regulates. The theorem can also be interpreted as saying that optimal regulators that are not models of their regulands are either unnecessarily complex or simply not good regulators [Conant and Ashby 1970]. Scholten [2011] goes even further claiming that every good solution must be a model of the problem it solves.

An important consideration in our modeling is the connection to physics. Learning agents such as [Hutter 2001]'s AIXI may be promising, but can not be realistically expected to solve problems in task-environments with time and energy restrictions. This may be due to AI research being derailed by premature formalization and reliance on Turing machines, which does not account for time or energy at all. This oversimplification has decoupled AI research from the physical world and shifted the focus towards control theory with no regard to time or energy limitations. [Thörisson 2013] addresses this specific issue.

Our work aims to create practical time- and energy sensitive task-environment models that can be
used to evaluate control systems by simulation and thus the approach taken here rests on several of
the assumptions made by cybernetics, namely that natural and artificial intelligence systems are likely
to be based on similar organizational principles which to a large extent determine their exact nature
as control systems.

2.2 Simulation

[Rosen 1993] raises the question of whether mathematical systems are simulable, claiming that most are
not, but covers the special case for mathematical systems which are already models with an interesting
corollary for simulable models of mathematical systems. By ensuring that a model of some system
is simulable, we can assert that running a simulation of that model is equivalent to a simulation of
that system. This allows us to transmute the Church-Turing thesis into an assertion leading to the
reductionist’s dream [Rosen 1993]. Furthermore, this tells us that all constructed task-environment
models can be made simulable by simply enforcing models to be constructed of smaller simulable
models all the way down to the smallest building blocks.

By constructing models with systems composed of smaller sub-parts, we can factor different parts
of the model (e.g. we could simulate multiple parts of a system in parallel, provided that we wish
to evaluate the current state of the evaluated agent and not the learning progress). We consider de-
composition a necessary feature for the reasons previously outlined and to encourage the construction
and use of sub-environments (or prefab systems to place in environments) [Simon 2000][Thórisson and
Nivel 2009].

The Model-View-Controller architecture is a software programming paradigm that applies a three-
way factoring of an application. The model is responsible for describing the application domain, the
view represents the interface between applications (or the user) and the controller governs operations
within the application (more specifically, operations on the model which in turn affect views). In
this way all possible views in an application can be imagined as the application’s state space while
the controller dictates how changes can be made to the model (which propagate to the view). The
meaningful decoupling of components suggested by the MVC architecture encourages code re-use and
simplifies pluggability (generally by employing standardized interfaces). [Krasner et al. 1988]

The prototype framework proposed in this thesis does not use the MVC architecture, although
section 6.2.1 hints at why implementing the framework with a model-view-controller architecture could
be a good idea. The prototype framework presented in this thesis describes an implementation with a
more model-driven architecture — the controller is embedded in the top-level model (i.e. the “largest”
model in its own task-environment).

Advantages to simulating task-environments have been discussed in the context of cognitive research
in humans in [Gray 2002]. Although our focus is on evaluating artificial cognitive systems, most of
the same principles apply — it is just as useful to put an AI into a flight simulator before letting it
loose in an actual cockpit. [Gray] also discusses three dimensions on which simulated task-environments
differ: (i) Tractability (ii) Correspondence (iii) Engagement. Tractability is an important but relative
dimension. Important considerations like the training, usability and expense of running the simulation
can affect tractability differently based on the research questions being asked. Correspondence can
vary from one aspect of many systems to many aspects of one system as the research goal becomes
more specialized. The engagement dimension arises from the needs of the participant [Gray 2002]. It
is not always clear what exactly the needs of an adaptive control system are and thus the engagement
dimension is not easily mappable from human participants to machine participants.
Saitta and Zucker's book on abstraction in AI and complex systems touches on simulation and how exploiting generality in parts simulations can greatly enhance performance, using commonly recognized examples such as LOD (level-of-detail) control [Saitta and Zucker, 2013]. This ties in with the idea of (de)composability, in an environment of multiple agents whose interactions do not affect each other in any way for some time period, each agent can be simulated in its own sub-environment in parallel until any interactions in the sub-environments propagate outward, at which point the simulations would need to be synchronized (or merged). The same principle applies for multiple non-interacting systems within an environment.

Due to the nature of performing complex tasks, we insist on simulating (models of) complex systems as our main approach, in a way that takes into consideration how exactly this can be achieved in a practical manner. We can begin considering some of the simplest mathematical representations of complex systems such as cellular automata, but we are already limited by their discrete nature. We aim to simulate complex systems whose behavior can be described by continuous dynamics (as opposed to discrete dynamics) while coupling the passage of time and expenditure of energy into the simulation with Newtonian physics.

2.3 Evaluation Methods for AI Systems

Toolkits such as rllab and OpenAI’s Gym implement various kinds of task-environments for evaluating reinforcement learning algorithms. However, these toolkits are focused on reinforcement learning and algorithm development. Implemented tasks include classic control tasks like Mountain-Car and Cart-Pole Balancing, locomotion tasks, algorithmic tasks, board games and more. [Duan et al., 2016]. Microsoft’s Malmo takes things a step further, using Minecraft to implement challenging tasks for comparing control systems. [Abel et al., 2016].

Hernández-Orallo [2010] defines an environment class Λ that allows procedural generation of task-environments requiring a suite of abilities that can be appropriately sampled and weighed. This environment class uses a cellular spatial approach and implements rewards as a function of the evaluated agent’s position and the positions of two agents “Good” and “Evil”. Hernández-Orallo and Dowc [2010] introduces an intelligence test that can be applied to any kind of intelligent system in any amount of time (although more time leads to more reliable results). The environment class Λ is used in Insac-Cabrera et al. [2011] to show that the approach outlined in Hernández-Orallo and Dowc [2010] is feasible by evaluating a Q-learning algorithm. The environment class Λ does not cover environments with time and energy restrictions, as there is no connection to physics.

Hernández-Orallo argued that in order to assess general intelligence, that the assessment should cover the testing of a range of abilities required for a range of tasks. The need for ability-oriented evaluation as opposed to task-oriented evaluation is highlighted in Hernández-Orallo [2014]. Task-oriented evaluation methods may be suitable for some AI systems such as those operating in a fixed domain, but ability-oriented evaluation methods are more suitable for more general systems that we might expect to solve previously unseen tasks. [Hernández-Orallo, 2014]

Hernández-Orallo and Dowc [2010] includes an entire section on time but focuses on the intelligence and not the speed of the evaluated agent. Hernández-Orallo [2015] revisits the concepts of task and difficulty in the context of universal psychometrics. The notion of asynchronous-time stochastic tasks is introduced, paving the way for new ways to evaluate task difficulty and instance difficulty. Synchronous environments are described as a special subclass of asynchronous environments, highlighting the need for asynchronous-time tasks for universal evaluation methods.
(A) Offering easy construction of task-environments, and variants with a wide range of features and complexity dimensions. This would include the ability to (a) compose and decompose desired task-environments and parts thereof, and (b) to scale and tune them, in part and in whole, along various parameters and properties, with predictable effects, especially for increasing and decreasing their complexity along known dimensions.

(B) Ability to specify, at any level of detail, the procedural generation of task-environments with specific features, constraints, etc., and how they should (automatically) grow, possibly depending on the progress of the system under evaluation.

(C) Facilitation of analysis in terms of parameters of interest, including task complexity, similarity, observability, controllability, etc.

1. Determinism: stochasticity should be controllable
2. Ergodicity: reachability of states (or sub-states)
3. Controllable Continuity: discretization granularity
4. Asynchronicity: any action may operate on arbitrary time scales
5. Dynamism: environments can range from static to dynamic
6. Observability: what the agent can perceive
7. Controllability: agent control
8. Multiple Parallel Causal Chains: supporting co-dependent objectives
9. Number of Agents: should not be limited (1 → many)
10. Periodicity: repeating patterns
11. Repeatability: experimental repeatability

Arguments for why artificial intelligence research is in dire need of a task theory can be found in Thórisson et al. [2016]. Other fields have strong theories of tasks in their domain that allows them to thoroughly evaluate their design by methodical manipulation of well understood parameters of known importance. A task theory would allow tasks — however similar or different — to be measurably compared with appropriate formalization and classification techniques. This could be used not only to compare tasks but also to systematically construct tasks for specific purposes such as for training machine learning algorithms [Thórisson et al., 2016].

The current thesis is an attempt at taking the first step in the direction described by Thórisson et al. [2015]. Our method is based on a fusion of object-orientation and agent-based simulation — by focusing on (de)composability, we aim to facilitate the procedural construction of ability-oriented task-environments while retaining a connection to physics.
2.4 Foundations for a Task Theory

A task theory provides methods for the representation and analysis of tasks, environments and their relations. There does not currently exist a proper task theory. This section outlines the work leading up to what will hopefully become proper task theory.

Russell and Norvig [1995] contains a chapter on intelligent agents which goes into the relationship between agents and their environments. There, environments are defined in a variety of flavors depending on their properties. Environments are defined to be either accessible or inaccessible, deterministic or non-deterministic, episodic or non-episodic, static or dynamic (or semi-dynamic if time affects the agent’s performance but not the environment itself) and finally, discrete or continuous. The “hardest” environments are described as inaccessible, non-episodic, dynamic and continuous and that for all practical purposes, the most complex environments must be treated as non-deterministic. More recent publications have adopted the term “task-environment”, defining a task-environment as a pair of an environment and a specification $\Psi$ that needs to be satisfied in order for the task to be considered successfully completed [Wooldridge, 2009].

However, despite this existence of terminology, no proper task-theory has been presented. Although task theory in itself is not the topic of this thesis, it does build on the assumption of some task theory. Building on task theory can greatly facilitate comparison of different task-environments, as highlighted in [Thórisson et al., 2016]. Thórisson et al. [2016] addresses the difficulty for comparing agents built for and evaluated in different tasks by pointing out issues with the status quo, what we might want from a task theory and its various applications.

The definition of task-environments as the pair $(\text{Env}, \Psi)$ (where $\Psi$ represents a task and $\text{Env}$ an environment) is a good starting point, but lacks explicit information about time and energy — it would make sense for $\Psi$ to contain limitations on time and energy expenditure. The environment $\text{Env}$ can be further defined as the set $\text{Env} = (V, T)$ where $V$ is a set of all variables and $T$ is a set of all transitions describing how the variables in $V$ can change. We can elaborate further and re-define $\Psi$ as a set of Problems (Wooldridge defines $\Psi$ as a specification), each of which is a set of Goals with all relevant constraints that need to be accomplished in order to consider the task to be successfully completed. A task-environment needs to include at least one Problem in order to be considered a task-environment, otherwise it is simply an environment [Thórisson et al., 2016]. See Thórisson et al. [2016] for further details and elaborations on task theory conceptualization ideas.

In Wooldridge’s book on multiagent systems, tasks are classified into achievement tasks and maintenance tasks. Achievement tasks are self-explanatory — the regulator should achieve some goals (make some goals true). Maintenance tasks are those in which some specification $\Psi$ must be maintained (i.e. the environment needs to be regulated to prevent any goals from becoming unreachable). For how long must they be regulated? It is tempting to simply say “Infinity!” — but it is in fact unreasonable to assume that a task must be regulated for infinite time, especially in the context of evaluation.

Following the assumption of insufficient knowledge and resources (AIKR) [Wang, 2011] [Thórisson, 2013], it follows that tasks have a limit on time and energy. If there were no limit on time or energy, there would be no need for intelligence, as a random search for the correct solution would be successful without resource restriction [Thórisson et al., 2016]. From this we can assert that task-environments must not only have a minimum time and energy associated with them, but that it would be reasonable to assume that the Problems in the task-environment (in $\Psi$) place some restrictions on maximum time and energy as well. A task like “move my car out of the driveway” would probably contain a Problem with the maximum time representing the urgency of the task and the maximum energy representing the efficiency — the task can certainly not be solved if the car runs out of fuel, unless the parameters allow the agent to go and fetch more fuel and refill the gas tank before the time limit has been reached.
2.5 Candidate Tools

Many of the questions we address have been the subject of prior research in one form or another. One question we must answer before starting the work is whether we can make use of some prior tools. The work described in this thesis is somewhat interdisciplinary between fields such as theoretical computer science, simulation and physics. As such we may be interested in examining tools such as verifiers, simulators, proof tools etc. There are several considerations to be taken into account in selecting suitable candidates — we want to be able to construct a task theory that supports modular task construction with such a tool and would need to be able to tie in some physics.

We are interested in addressing how to formulate tasks in a convenient way so that it scales from simple tasks, such as e.g. opening a money safe (turning a few dials) towards much more complex tasks (the proto task theory from the previous section), e.g. traversing complex disaster areas to retrieve survivors, and finding convenient ways of implementing such a task theory in a powerful tool.

We can start by considering tools such as OMNeT++ and Rebeca. OMNeT++ is a C++ class library providing a modular component architecture similar to that of FraMoTEC [Varga et al., 2001]. Rebeca is an actor-based language with a formal foundation for modeling concurrent and distributed systems, designed to bridge the gap between formal verification approaches and real applications [Sriram, 2007], so it might seem quite appropriate for modeling complex task environments and although these tools are not meant specifically to deal the concerns of our work, we should still take them into consideration. Unfortunately, many such tools, including these two, are discrete in nature. Even if we could extend them to support continuous variables, we would still hit a wall due to state space limitations and would be restricted from many of the kinds of tasks intelligent machines might do. It is also questionable whether they could scale to the kinds of complex tasks we target. Imagine an agent confronted with a simple combination bicycle lock: Turning a number half-way could be modeled (e.g. with Rebeca) as an “invalid number” state or it could be discretized. For a 4-value combination lock this would only produce 10000 states, but what about a slightly more difficult lock, such as one found on a bank safe, and what if the task is to crack three different safes? Some complication could easily bring the state space billions higher. In order to accommodate for state space limitations we would most likely have to resort to manual fiddling for such complex (yet simple) tasks, as simply flattening the state space or downsampling would not be sufficient to surpass practical limitations. It seems clear that just a slightly more complex task would bring these tools well beyond what is practically possible.

Among other functionalities we need are simulation and physics. Many physics simulators exist, such as NEWTON, Bullet or Golems. However, the simulation techniques we are interested in do not necessarily have to target accurate physics simulations, we are more interested in being able to simulate modularly constructed tasks interactively. Before we might leverage such physics simulators, we have to understand how we want to express physics with task theory. Building on pre-existing tools may be a bit premature in this area since they don’t help us answer the questions we are most interested in, but this is something that can be revisited in the future.
FraMoTEC offers a mapping from a proto task theory to a tool, along with utilities for easy modular construction of task-environments. The simulation component is not very mature but provides a connection to physics and facilitates rudimentary analysis and visualization. While simple, it is sufficiently functional and — more importantly lightweight and integrated — to help us explore the key research questions and topics targeted, e.g. help us experiment with how to conveniently represent task theory.

Ptolemy II is a rather comprehensive set of Java packages supporting heterogeneous, concurrent modeling and design. It includes a suite of domains, each of which realizes a model of computation. Examples of models of computation include discrete-event systems, dataflow, process networks, synchronous/reactive systems, and communicating sequential processes. Ptolemy II takes advantage of the actor model and comes with a large array of components that can be used to build models of complex systems and includes a number of support packages [Ptolemaeus 2014].

It is reasonable to ask whether we could use this tool, which is quite powerful and mature. One focus for Ptolemy II is on the assembly of concurrent components — a focus shared by FraMoTEC. However, Ptolemy II builds on a key underlying principle: the use of well-defined models of computation (or domains). The model of computation employed by FraMoTEC aims to be as general as possible to facilitate the maintenance of a mapping between a theoretical task theory and a practical tool. Also, and more importantly, Ptolemy II does not at this point help with many of the key questions we focus on.

In conclusion, although tools exist that could potentially be of use to us, using them prematurely is more likely to introduce technical debt than to build one from scratch. In doing so we can more easily examine the applications of task theory during its development. In the future, tools such as Ptolemy II might prove indispensable, but at this point in time using such tools would be overkill.
Chapter 3

Components of a Framework for Evaluating Task Performance

3.1 The Framework

It should be apparent that research in the vast field of artificial intelligence could benefit from a tool that allows flexible construction of tasks, automation of evaluation and analysis of performance [Thórisson et al., 2015]. Since all AI systems will ultimately perform something in the physical world, and a large part of future AI systems will be deployed to control physical devices, whether it be robots, automatic doors, electricity and airflow in skyscrapers, or distributed interconnected things (internet of things), it is imperative to include time and energy in any tool we might construct for this purpose. This way, the tool would be applicable well beyond the decision-centric AI systems of the past.

We propose a framework for evaluating controllers (or agents, system regulators) based on their time and energy requirements for solving a task. A draft of such a framework has been implemented in Python with an object-oriented approach, using a layered composition of small building blocks to describe entire environments. Chapter [covers framework-specific details for the prototype. The base constructs used by the framework are Objects, Transitions, Systems, Motors and Sensors, as these components can be arranged to create models at varying levels of abstraction.

The building blocks come together, forming what is finally called the model — i.e. the complete representation of the task-environment (and by extension, the natural system that the model represents). The model is responsible for maintaining any “bookkeeping” operations required by the framework, such as incrementing the simulation clock and inspecting the energy usage. It should provide references to every building block within the model in an organized manner.

3.1.1 Building blocks

Constructing task-environments requires using the building blocks provided by the framework. These building blocks are designed to be as basic as possible to allow for a wide variety of behaviors to emerge from the different combinations of organization of the blocks. Most of the organizational complexity emerges from the creation of systems of objects with custom transitions. The following building blocks have been designed for the framework: (i) Objects (ii) Transitions (iii) Systems (iv) Goals (v) Motors.
Sensors (vii) The model itself, which acts as a container for other building blocks.

In terms of composability, the model is the outer layer of the task-environment and defines the task in the task-environment in terms of solution goals. Objects (and transitions\footnote{Transitions are the basic building block that allow object behavior and interaction to be changed/introduced.}) can be viewed as the most basic constructs, followed closely by goals, sensors and motors.

Object Physics

The framework implements basic one-dimensional kinematics individually for each object. Objects have physical properties like velocity, mass, friction and might even contain values for gravitational acceleration at some angle. This allows the object to naturally transition: the velocity after \texttt{delta\_time} seconds is computed based on the current velocity and the input power (and direction) from any actuators, then the value is updated based on the velocity. Although the framework does not currently implement other object behavior, it should be simple to envision extending the framework. We could for instance extend objects to radiate some of their mass over time and use such an extension to create tasks that are time-sensitive internally. \textsection{3.3} covers the natural transition of objects in more detail.

Transitions

Transitions come in two forms, the \textit{natural} form and the \textit{designed} form. Designed transitions are specified by the task-environment designer. Designed transitions expand upon the natural behavior of an environment by adding custom logic to it without requiring the framework to be extended specifically. Natural transitions are not specified by the task designer but are provided by the framework automatically, like the natural transition of an object or system. A transition will only affect objects it is designed to affect but there is currently no reason to prevent a transition from modifying another transition’s list of affected objects (this can be done to produce dynamic behavior). Transitions serve to modify the behavior of an object or system — the transition, when fired, will perform some computation that might result in some modification of the object (or system). We could for example implement a transition that causes some objects to radiate some of their mass over time instead of extending the framework.

Systems

Systems facilitate composition immensely by acting not only as a container for objects but also as an encapsulator of the objects’ behaviors. We can for example model a particle whose mass decays at a rate of 0.01 kg per second by placing that object in a system that contains a transition like \texttt{var\_mass -= 0.01 * delta\_time}. The natural transition of a system is to apply all transitions within. Since transitions affect only the objects they are designed to affect, one could even stuff thousands of objects, each with their respective transition functions into a single system — if it made sense to do so.

The composability of systems allows for all kinds of behaviors in the environment model. One could conceive of a controller of a system within an environment, modeled with systems of objects, transitions and motors.
Goals

Goals are necessary to describe tasks and ultimately form the solution to a given task-environment. A goal specifies a target object along with a goal value and tolerance. Goals can also depend on other goals to be satisfied (i.e. the goal can require other goals to be met before it can consider itself satisfied).

Once a goal has been satisfied, it is forever considered satisfied unless it is reset, in which case both the goal itself and any prerequisites will be reset recursively. This allows the state of the task to be evaluated based on the number of achieved goals with disregard to the current environment state.

In an example task in which the task is to “press button X, but not button Y”, we would set a goal for button X as “X has a value of \( X_p \pm X_{p\_tol} \)” where \( X_p \pm X_{p\_tol} \) represents all states in which the button is considered to be pressed. Likewise for the goal for button Y, we would set the goal as “Y has a value of \( Y_{np} \pm Y_{np\_tol} \)” where \( Y_{np} \pm Y_{np\_tol} \) represents all states in which the button is considered NOT to be pressed.

Motors

Motors can be considered the “effectors” or “actuators” in environments, that can be utilized to solve the task. They are foundation for being able to account for energy expenditure in task-environment simulations.

Motors are currently the only object actuator. Motors have properties relating to the maximum power and whether or not it can run in the opposite direction (and if so, at what power). Activating a motor with more power than it is capable of outputting results in energy being wasted — it will be recorded as expended but a motor can never output more than its maximum power.

Motors can be placed in systems of objects with custom transitions to create “new” behavior. For example, we can imagine that we must control the 2D position of some objects in a system, but we only have access to a control motor that we can rotate with a selection motor. From this description it should be clear that this control task is feasible, but what is not necessarily clear is how to construct such a mechanism. A specific example is outlined in 4.4.2.

Sensors

Standardized read access to objects’ values can be provided via sensors. A Sensor reads an object’s value, optionally applies some distortion and rounds the value to a certain number of digits. Sensors can also read other sensors in multiple layers, overlaying distortions or rounding at any level.

3.2 Task Construction

Tasks are constructed modularly using the smallest possible constructs, in the spirit of Thórisson and Nivel’s publication on peewee granularity. The most simple environment conceivable is a single system containing one object and one motor capable of affecting that object. The simplest task-
**environment** contains a solution system consisting of the aforementioned **environment** (as a System) and a Goal, with the goal value and tolerance as variables in the solution system. The model of such a task-environment contains only this solution system and some limits on maximum time and energy.

We can construct more complicated tasks by adding more objects and/or motors to the environment. We can enforce systems behavior as desired by implementing appropriate transitions. We can for example create a system in which $X$ and $Y$ move independently except if $Y < 20$ by implementing a transition that checks if $Y < 20$, locking $X$ in place (freezes its state) if so, and unlocking it otherwise.

Although objects in task-environments can be interacted with directly, it should be preferable to create a Sensor that provides access to the reading of some object. The reading on the sensor can be rounded and distortion can be added to the value before rounding.

Constructing tasks modularly with composition not only allows the model to be factored, but allows other models to make use of its sub-models. Sharing sub-models between task-environments can facilitate comparisons of task-environments despite a lack of proper task theory. Factoring task-environment models into smaller sub-models enables sub-models to be simulated independently (even in parallel) until changes within the sub-model propagate outward (at which point synchronization is necessary to continue). As an added bonus, this approach makes it much easier to compare agent embodiments (or measuring the effects of various body features).

An interesting result of this composition is that implementing relationships such as orthogonality between two dimensions would require implementing the interactions resulting in orthogonality becoming a weakly emergent property of the environment.

### 3.2.1 Examples

**One-dimensional Drag Racing**

In 1D drag racing the **destination** is simply a point on a number line with some width **epsilon** in either direction. The racer is an object we will call **car**. Our task model then consists of the solution goal satisfied by $\text{destination} - \varepsilon < \text{car} < \text{destination} + \varepsilon$. In order for this goal to be attainable, a motor must be attached to either **car** or **destination**. For this particular example, it doesn’t make much sense to move the destination, so a motor for **car** is created and configured to the liking of the task designer. The task-environment is now complete for the purpose of being solvable, but still requires a Sensor for **car** (and optionally for **destination**) so that a controller can interface with it without cheating.

Since this task is being modeled in a single dimension, the condition $\text{destination} - \varepsilon < \text{car} < \text{destination} + \varepsilon$ will become true as long as the **car** passes by the destination, since the goal becomes satisfied immediately as the **car** falls between $\text{destination} \pm \varepsilon$. Notably, the intermediate value theorem tells us that even if the car overshoots the destination, it will still have satisfied the goal at some point. Unfortunately, it is possible to overshoot goals with the framework as it is and rectifying this error (however unlikely) is left as future work.

---

2 Modeling a 'solution system' allows the goal values to be changed at runtime, but hard-coding the goals into the task-environment is possible, in which case a solution system is unnecessary.

3 Figure 5.1 depicts an instance diagram of this environment.

4 Although we call our object **car**, the physics in the model more closely represents a block being pushed than a car rolling on wheels.
Figure 3.1: A visual representation of what 1D drag racing could look like. This representation shows us that the task is defined on a single independent dimension.

**One-dimensional Plotting**

In the previous example, the only requirement was for the car to pass by the destination, i.e. the task would be considered solved if at any time, the solution condition is satisfied. We can imagine a similar task more analogous to a printer, in which we redefine our car as the location of the ink-jet and the destination would then be the point at which ink is to be applied (although the car could be unbounded, it would make sense to bound the plotter’s position). We must add another object, indicative of whether or not ink is being actively applied at the plotter’s current position and a motor to enable some controller to ‘plot here’. The task-environment solution for a single-point plotting task would then require both the plotter’s location and ink application to be within the right ranges.

To make the task-environment stricter (and to prevent a controller from simply plotting ‘everything’), we could introduce an object acting as a counter for incorrect ink applications. This could be implemented with a transition that checks if the ink-jet is being applied while not in a goal position — then the solution would include the goal that this counter object must be equal to its starting value (or below some threshold if errors are acceptable).

Of course, it is hardly realistic of a task to require only a single point to be plotted, so in order to plot meaningful images we can consider the solution to be a set of goal points that need to be plotted and our error-counting object would have to be incremented only if plotting is done within none of those goals.

**Extending 1D tasks to N dimensions**

One dimensional tasks are inherently easy and therefore not particularly interesting. In the racing example, we can solve the task every time by simply applying maximum power to our only motor until the task is solved (as long as the direction is correct). In the plotting example, we can clearly see that by introducing new dimensions we can start to model printing of complete images and even 3D printing.

Fortunately, each dimension usually governs a single component in the physical world, so to extend 1D racing to ND racing we could simply replace each object with N similar objects, one for each component. Likewise, we would want to introduce additional motors to allow controllers to operate in all dimensions. We could also replace all motors with a single ‘Multi-Motor’ and some way of configuring the motor so that the controller could decide how much of the total input energy would go into each component. For the plotting task, we would probably prefer using separate motors for each dimension for more fine-grained control, but the transformation from one dimension to many remains the same.

— See figure 5.2 for an instance diagram.
The Goal of the Mole

The plotting task described above goes a long way in capturing the essence of more “realistic” tasks, such as the carnival game ‘Whack-a-Mole’. In Whack-a-Mole, the player must whack as many moles with a hammer as moles pop up through holes in the machine.

A simplified model of this game would model the agent’s body as the position of the hammer in space, it would in fact be quite similar to the plotting task. We could then either make hitting each mole a goal of its own, or make the goal to whack at least some threshold number of moles — or both! We could define one goal as “Solve more than X goals from the goal collection Moles” and then generate a goal for each mole that pops up in the environment.

3.3 Object Physics

The framework prototype currently implements a single natural transition for Objects, which describes the Object’s change in position and velocity based on its actuators. The only currently allowed actuator type is the Motor, although future expansions should allow other kinds of actuators (like Systems or other Objects for collision physics). See chapter 4 for details about the implementation. The physics formulae governing the laws of motion under constant power can be found in the 1930 Ohio Journal of Science. The laws provide a means to anchor power (and by extension, energy) into the framework.

We define an Object’s natural transition as the change occurring due to a passage of $dt$ seconds:

1. If the object is in a non-changing state (no velocity, zero angle towards gravity, no actuators), or if it is locked, then it does does not change during the transition.

2. Otherwise, first the velocity is updated:

   • If the velocity is zero, we calculate the forces due to actuators, gravity and friction, see equations [3.1]. If the force does not overcome static friction, velocity remains unchanged (any input energy is wasted).

   • If the velocity is nonzero, we calculate the change in velocity and update the velocity accordingly. See equations [3.2]. Note that if the sign changes, we can use the intermediate value theorem to determine that the object came to rest during the time period. The
simulator currently approximates this by setting the velocity to zero\(^6\).

3. Finally, the position is updated:

\[ s = s_0 + v \cdot dt \]

If the velocity is zero, we calculate the forces due to actuators, gravity and friction. If the force does not overcome static friction, velocity remains unchanged (any input energy is \textit{wasted}). Otherwise, the velocity is updated as per equations\(^3.1\)

\[
\begin{align*}
a_{\text{power}} &= \text{sgn}(P) \cdot \sqrt{\frac{|P|}{2m \cdot t}} & \text{acceleration due to power from actuators} \\
a_{\text{move}} &= a_{\text{power}} - \sin(\theta) \cdot g & \text{acceleration due to gravity and actuators} \\
F_{\text{move}} &= a_{\text{move}} \cdot m & \text{total force in the direction of movement} \\
F_{\text{net}} &= \text{sgn}(F_{\text{move}}) \cdot (|F_{\text{move}} - F_{\text{kin}}_{\text{fric}}|) & \text{net force in direction } \text{sgn}(F_{\text{move}}). \\
a &= \frac{F_{\text{net}}}{m} & \Leftrightarrow F = ma \\
v &= a \cdot dt & \text{final velocity determined}
\end{align*}
\]

...where \(\text{sgn}(x)\) is the sign function and \(t\) is an actuator property.

If the velocity is nonzero, we calculate the change in velocity and update the velocity accordingly:

\[
\begin{align*}
m \cdot \frac{dv}{dt} &= \frac{P}{v} + F & (F = ma \text{ rewritten}) \\
\Rightarrow dv &= \left(\frac{P}{v} + F\right) \cdot \frac{dt}{m} & \text{solve for } dv \\
(\text{or}) \quad dv &= \left(\frac{P}{v} + F_0 + F_1 + \cdots + F_n\right) \cdot \frac{dt}{m} & \text{(as components)} \\
F_P &= \frac{P}{v} & \text{force due to power input} \\
F_G &= -mg \cdot \sin(\theta) & \text{force due to gravity} \\
F_{\text{kin}}_{\text{fric}} &= -\text{sgn}(v_0) \cdot mg \cdot \cos(\theta) \cdot \mu_k & \text{force due to friction} \\
F_{\text{net}} &= F_P + F_G + F_{\text{kin}}_{\text{fric}} & \text{net force} \\
dv &= \frac{F_{\text{net}} \cdot dt}{m} & \text{final change in velocity}
\end{align*}
\]

...where \(\text{sgn}(x)\) is the sign function.

If the sign changes (i.e. \(\text{sgn}(v_0) \neq \text{sgn}(v_0 + dv)\)), we can use the intermediate value theorem to determine that the object came to rest during the time period. The simulator currently approximates this by setting the velocity to zero.

This approach enables the physical properties of an agent and its environment to be taken into account when evaluating the efficiency with which a particular controller could achieve some physical task, both with respect to time and with respect to energy (or a combination).

\(^6\)This approximation may result in slight loss of energy, which may become apparent with very low time resolutions (large \(dt\)) or extreme forces.
3.4 Simulation

As previously established, time and energy usage are key metrics to consider when evaluating the performance of agents in a given set of task-environments. It goes without saying that an agent that spends 2 minutes and 20 KJ of energy to solve a specific task-environment is worse at solving the task than an agent that spends 30 seconds and 700 J in that same task-environment.

Naturally, we can continuously measure the time and energy expenditure of an agent to quantify the total amount of time and energy required to come up with a solution to some task. In this sense we are not evaluating an agent’s ability, but an agent’s ability to improve some ability (i.e. the agent’s ability to learn). Perhaps one agent spends more time learning about the environment while another happens upon a more favorable solution to some task early on. We can evaluate both agents multiple times and see how they progress for an idea about an agent’s learning speed. We can further extend both these approaches to a set of tasks in lieu of a single task, allowing for a more comprehensive evaluation and comparison of all kinds of agents.

The proposed framework should facilitate such comprehensive evaluations and comparisons as much as possible, which finally brings us to the importance of simulation. The designers of task-environments ultimately produce formalizable models — this is a natural implication of the framework building on simple, simulable causal processes (the building blocks and their interaction). As stated in Rosen [1993], a simulation of the model becomes a simulation of the natural system that the model represents, transmuting Church’s Thesis into an assertion (all systems that can be modeled by the framework are simulable).

The simulation component of the framework would ideally be truly continuous, but the nature of the Von Neumann architecture encourages stepwise integration. As such, every simulation step (or tick) regardless of length should optimally ensure that:

- All objects in the task-environment should naturally transition (if applicable)
- All custom transitions should fire on all applicable objects
- Goals should be asserted to evaluate whether success (or failure) conditions have been met
- The time passed during the frame must be recorded and added to an accumulator
- The energy used by any motor during that time frame should be recorded and added to an accumulator
- Current time and energy usage should be compared with task time and energy limits

See section 4.2.2 on simulation for details about the implementation of the simulator in the prototype framework.

3.5 Summary and Discussion

In this chapter we have covered the principle ideas of a framework for evaluating agents in task-environments while measuring time and energy expenditure. We have described a set of building blocks with which task-environments can be constructed and how they interact with each other. Section 3.2
covers task construction and provides examples on how to construct task-environments modularly. Finally, section 3.4 explains how simulation can be used to evaluate agents.

To summarize, starting with the three requirements for intelligence evaluation frameworks listed in Thórisson et al. [2015], the proposed framework provides a means for which construction can be made easy by exploiting modularity, including some limited variation. The same can be said for procedural generation of task-environments at different levels of detail (requirement (B)) — although the framework doesn’t directly implement procedural generation of task-environments, the groundwork has been laid out. The final requirement is the facilitation of analysis in terms of parameters of interest. Constructing tasks modularly by composing structures of smaller structures facilitates not only static analysis of task-environments, but dynamic analysis of agents’ interactions in task-environments as well. In this sense it is fair to conclude that the proposed framework does not currently fulfill all requirements fully, but considerations towards enabling the requirements to be easily implemented as features have been taken.

Alongside these three requirements, Thórisson et al. [2015] also lists 11 properties that should be controllable (i.e. tunable or can be gradually constructed). The principles of the framework ensure that none of these properties are impossible, although as discussed in chapter 4, the prototype does not implement the necessary features for each and every property. A detailed evaluation of FraMoTEC in light of the desired features is presented in section 4.6.

The flexibility of the framework allows it to fulfill other purposes than just evaluation of agents in task-environments and comparisons of task-environments. Using the proposed framework, it would be possible to create controlled settings (or models of settings) to not only evaluate agents, but for learning purposes. We could factor a complex task-environment into sub-task-environments representative of some fundamental systems of interactions (which would be the target for teaching) [Bieger et al., 2014].

Imagine an example competitive fish-eating task-environment. The agent is embodied as a small fish and must control its movement by rotating itself (with side fins) and going forwards (with the tail). In the “real” environment, there may be multiple competing agents and a top-level goal of accumulating a certain amount of mass. In order to accumulate such mass, the agent should either eat food pellets scattered around the environment or eat smaller fish. Throwing an intelligent agent into a sea of fish and pellets might sound like a good idea, until reality sets in and chance allows other agents to accumulate so much mass that the agent is unable to compete with them most of the time. This becomes a problem if the agent doesn’t manage to learn anything about its environment before being eaten, in this case it would make sense to let it learn in a sandbox rather than in the harsh ocean. The exact same embodiment can be situated in a fish tank in which there are only food pellets, this allows the agent to peacefully learn that it should eat food pellets to accumulate mass. The same can be done for an environment where competing fish are introduced only once the agent has accumulated X mass — repeating the exercise and decreasing X every time allows the agent to become gradually accustomed to “the real task”, moving away from the experimental setting.

By limiting the set of task-environments supported by the framework by the set of simulable environments, we can exploit the simulation engine to not only evaluate agents and task-environments, but also for experimentation which can also be used to e.g. evaluate agents’ learning speeds.
Chapter 4

Software Implementation

4.1 FraMoTEC: Framework for Modular Task Environment Construction

The framework described in chapter 3 has been implemented in Python. Implementation follows closely the layout described in the prior chapter — “objects” of the framework, motors, goals, etc., have been implemented as software objects and the physical constraints as methods that define how these objects interact. An intelligent controller connects to task-environment instances by interacting with sensors and actuators in the environment.

Section 4.2 discusses the modularity of task-environments that can be constructed with the framework. Section 4.2.1 outlines the mapping from the framework as it was described in chapter 3 to the FraMoTEC prototype, followed by section 4.2.2 on simulation and section 4.3 on controller interfacing. Section 4.4 demonstrates some task modeling examples followed by section 4.5 on supporting tools. Finally, the chapter is concluded by a summary and discussion in section 4.6.

FraMoTEC is implemented as a collection of object-oriented Python classes\(^1\). The most notable class in the framework is the Model class, which encapsulates the behavior necessary to run a simulation of the model. The Model owns a System representing the environment and a collection of Goals that should be satisfied representing the task. Simulation parameters are also maintained in the Model object, along with values for maximum time and energy expenditure during simulation.

4.2 Construction and (de)composition

Figure 4.1 depicts a high-level class diagram of the task-environment model as implemented in FraMoTEC. By exploiting the recursive power of modularity, we can describe a wide range of environments composed entirely of these depicted classes. Figures 5.1 and 5.2 depict examples of environments visualized as instance diagrams, demonstrating composition from base components.

Section 4.2.1 provides a nice overview of the building blocks available in the framework and how they interact with each other. What it does not mention are hidden building blocks. It is possible to

\(^1\)The FraMoTEC code is available at https://github.com/ThrosturX/task-env-model.
create building blocks that do not get counted in the Model’s bookkeeping operations. This could for example be a hidden Motor that gets activated by a custom Transition but is otherwise inacessible; such a Motor would need to be created as an actuator on some Object. A more explicit approach is also possible, which does not result in hidden building blocks. This approach involves creating a System and populating it with the building blocks that would have been hidden, adding that System to the environment. If the hidden block is a Motor, then that energy expenditure must not doubly counted. This can be accomplished by resetting the usage and wasted_power attributes of appropriate motors in the transition functions that the hidden motors are used in.

A key feature of the modular design of task-environments is the ability to re-use parts of environments by working at the System level of abstraction. In essence, everything can be a system. A 3D crate can be modeled as the 9 objects, one “core” object and 8 locations of boundaries, with all of its behavior encapsulated within a System with a custom transition. The crate system becomes re-usable and could even be placed in a system with a transition function that does simple collision detection between systems — now implement the agent as a collidable system and the agent can interact with all the crates!

Furthermore, being able to abstract at the System level allows systems to be replaced “from the inside out”. We can imagine an environment of amoebae, with each amoeba modeled as a System of center and radius objects. Perhaps bigger amoebae are able to ingest smaller amoebae (like in the fish example in 3.5), but a discovery is made that renders the current modeling technique obsolete — comparing the radii of the competing amoebae incorrectly determines the winner as this level of modeling does not account for the altered shape caused by extended pseudopods. Instead of re-constructing the entire environment model, we can simply replace the generic circle-based amoeba system implementation with an implementation of higher fidelity.

4.2.1 Building Block Implementations

The Model owns a System named environment. The purpose of this system is to act as container for everything that represents an environment. To understand the hierarchical composition explained in 4.2 one should examine the implementations of the building blocks implemented for task-construction with the framework. The Model also governs a list of Goals named solution — the task is considered solved if every Goal in solution is satisfied.
The **Object** class

The **Object** class represents the smallest *environment* building block, but despite this it implements a fair amount of logic. The logic is implemented in the **UnboundedObject** class, but the framework implements the **Object** class as an extension that places upper and lower bounds on the position of the **Object**. The **UnboundedObject** class inherits all the methods defined in the **AbstractVariable** class and implements the logic necessary for simulation by implementing a **natural_transition** function that updates the **Object** based on the formulae in [3.3](#) if the instance is not locked. **Object** instances can be locked to prevent the **natural_transition** from happening. The **AbstractVariable** class implements some syntactic sugar that allows **Objects**' values to be compared with implicit comparisons.

The **Transition** class

The **Transition** class contains a list of **Objects** and a **transition** function implemented in Python. It can optionally include a separate **precondition** function that must evaluate to **True** in order for the transition to be applicable. A **transition** can be applied on its affected objects and any additional arguments. If it does not have any affected objects, it must take some arguments instead (this is to defensively prevent transitions from being called that should have arguments if they are missing). When the **apply_transition** method is called, the **transition** function is executed.

The **System** class

Every instance of the **System** class owns a collection of (i) **Objects** (ii) **Transitions** (iii) **Motors** (iv) **Sensors** and (v) **Systems** with an implementation of the **natural_transition** function that applies every transition in the **transitions** collection. It also implements accessors to all **Objects**, **Motors**, **Sensors** and **Systems** in the context of the current **System** instance and any instances in its **systems** collection along with convenience methods to lock or unlock all items in the **objects** collection.

The **Goal** class

The **Goal** class inherits from the **AbstractVariable** class and implements methods to (i) add other goals as prerequisites (ii) inspect the current condition of the goal itself (iii) assess whether or not the current condition along with all prerequisites are satisfied ⇒ if they are, that instance has its **satisfied** property set to **True**; and (iv) set the **satisfied** property to **False** (in itself and any prerequisite goals).

**Motors** and **Sensors**

Every **Motor** instance has a set of properties governing whether the motor is reversible and the maximum output of the motor in each direction. Activating a **Motor** sets the **Motor's power_level** and maintains any bookkeeping regarding power usage and waste (energy is considered wasted if the **Motor** is activated for more power than its maximum power in that direction). Every **Sensor** instance allows read access to a single **Object's value** with some specified amount of random distortion and digit or decimal place rounding.
Implementation note: Although the Sensor class is the proper way to read object values, there is nothing to restrict access to “inaccessible” objects or other constructs. This facilitates the theoretical creation of Instructors that could then base instruction on non-observable environment data.

4.2.2 Simulation

The framework implements a class-level method `tick(self, delta_time)` on the Model. Section 3.4 on Simulation depicts an ideal implementation of such a method. The implementation differs slightly. One small difference is that current time and energy limits are not evaluated in the framework during each tick (although this can be easily done by adding a single statement somewhere in the function body). Additionally, the framework prototype requires that for all (non-natural) transitions, any transition that should be fired must be a part of a system containing both that transition and any objects on which it should fire. This may introduce some complexity in the form of inter-dependencies between systems for some environments, but greatly facilitates re-use of complex systems by being explicit in the formalization.

A high level description of the `tick` method is as follows:

1. For all systems: naturally transition for \( \Delta t \) seconds (recall that any system’s natural transition fires all transitions within)
2. For all objects: naturally transition for \( \Delta t \) seconds
3. Once all objects and systems have transitioned, check for solutions
4. For all motors: increment the motor’s usage by \((P_{\text{motor}} + P_{\text{wasted}}) \cdot \Delta t\)
5. Increment the clock by \( \Delta t \) seconds

Due to the rudimentary state of the framework with regard to setting up evaluations, the `tick` method does not check for failures to facilitate using constructed environments for learning or exploration. Hopefully, future incarnations of the framework will properly deal with conditions such as failure due to over-use of time or energy, but currently the responsibility to restart or end a simulation due to such conditions falls on the user.

4.3 Interface to controllers

The task-environment modeling framework allows controllers to manipulate any part of the model, it is up to the discretion of the programmer connecting the controller to the task-environment to allow only read access to Sensors and write-access to motor power levels that are a part of the agent’s body in the environment model. The framework does not provide any specific rewards but there is of course room for extending the framework in the future.

Evidently, some kinds of controllers require more assistance in others in functioning properly. A SARSA learning controller would require some method to evaluate the ‘current state’ relative to other states and custom logic might be involved in computing the reward for the any state in the task-environment. It is possible to use the framework to approximate such a value by checking how much energy is required to solve the task from the current state. It is limited however, as it doesn’t help
much in creating a reward function when it is implicitly necessary to “move away” from the goal in order to actually reach it.

An important consideration in using this framework is that the controller’s body is a part of the environment — the agent is already in the environment, it is just missing a control mechanism. Therefore, a controller would need to either learn what actions are available and what information it has access to, or have that knowledge programmed into it. In the general case, a controller will be able to read any sensor and adjust the power level of any motor. See figure 4.2 for an example of a controller in the 1D drag racing from 3.2.1. The TaskEnvironmentModel class provides access to all motors and sensors in the environment via the motors() and sensors() methods.

4.4 Modeling Examples

4.4.1 Environments

We consider simple environments to be those without custom transition functions and hidden objects. The 1D drag racing environment from 3.2.1 (also the first environment in section 5.1.2) would be considered a simple environment (in fact, the described environment is in the simplest environment class). These environments are considered simple since a solution to the environment should be attainable simply by inspecting the structure of the environment (if a solution exists). Adding custom transitions
not increases the level of complexity but also raises the bar for the domain of the task-environment.

The examples from 3.2.1 are described without custom transitions and are therefore simple. Chapter 5 makes use of the 1D drag racing example and some complex environments.

Environments with custom transition systems introduce some domain knowledge into the model of the environment. The second environment described in 5.1.2 contains a simple custom transition function that locks or unlocks the position object in place based on the value of the plot_it object. This transition function creates a causal relationship between position and plot_it that can be considered a part of the task domain but more importantly, that can be learned.

4.4.2 Custom Systems

In 3.1.1 on Motors, a system of a rotating motor was mentioned. In order to implement this system with the framework, a system must be created that encapsulates the behavior of the rotating motor, with respect to the variables it controls. Let us say that it must control two variable objects, pos_x and pos_y.

We define a new object angle and give it bounds like $[-\pi, \pi]$ to represent all angles in a circle. We must also create a motor rotator for this object to allow control over the angle and a motor main_motor to act as the accessible interface of the motor system’s power level.

We also create a dummy object, which exists only for the main motor. Since the main motor’s effect on pos_x and pos_y depends on angle, we create the dummy object main_power to serve as a reference to the motor that sets the power level. Now we create two motors, mx and my, each responsible for the movement along a specific axis (i.e. controlling pos_x and pos_y) and a transition function.

Note that we could also have implemented a complicated transition function in lieu of the two hidden motors, but for simplification purposes we will assume that we actually want to control two motors that can be controlled indirectly with the rotator and main_motor motors.

The transition function can access the power level of the main motor by querying the main_power object for any actuators (the main motor should be the only actuator). Now it can simply compute the powers for mx and my with basic trigonometry:

$$power_x = \cos(angle) \cdot total_power$$

$$power_y = \sin(angle) \cdot total_power$$

Now mx and my are ready to be activated with power_x and power_y respectively. The transition function should also reset the usage of mx and my to prevent counting the usage twice. Alternatively, the usage on the main motor can be reset instead.

The resulting system should contain the objects pos_x, pos_y, angle and main_power, the custom transition function, the motors mx, my, rotator and main_motor and any sensors (pos_x, pos_y and angle would be typical targets for sensors).
4.5 Supporting Tools

A plotting program is included with the framework. The plotter reads an input file and plots any valid data points. Any invalid data points are ignored, but the color of each plotted data point depends on the number of data points before it (invalid data points are counted for the color incrementor, so many fails followed by a few successful trials will look different than a few successful trials right from the get-go).

The plotting program also plots a curve for the task itself, indicating the maximum energy expenditure along with the points for minimum time and minimum energy. It is of course desirable to be as close to the minimum time and energy points as possible (the preference over time or energy is arbitrary and depends on the task).

The plotter uses an immature feature of the framework that generates time-energy profiles based on the physical properties of the task-environment. Task theory has not come far enough, for it is still unclear how the composition of objects in the task-environment should be represented in terms of time and energy — is the minimum time for a task-environment equal to the sum of the minimum times for each sub-task, or the maximum lowest time? These kinds of questions can be answered for specific cases, but can be hard to generalize. This may lead to some bizarre looking plots in environments with multiple dimensions or custom transitions.

In addition to plotting the default plots, the plotter can be used to plot time and energy progression over evaluations (in two separate plots). The plotter is currently the only supporting tool in the framework and was used to create the plots depicted in chapter 5.

4.6 Summary and Discussion

In this chapter we have discussed the implementation of FraMoTEC prototype, specifically covering architecture, simulation and supporting tools (and lack thereof). The architecture of FraMoTEC was covered by analysis of the task-environment structure. The simulator is coupled with the task-environment model, but future work discusses how the two can (and should) be separated.

Thórisson et al. [2015] lists a collection of properties as important for the construction of evaluable task-environments. The three requirements listed in 3.5 are largely isomorphic between the prototype implementation and the proposed implementation of the framework. We can however attempt to evaluate the prototype implementation of the framework by going through this list of properties:

To summarize, FraMoTEC addresses each of the desired features identified in Thórisson et al. [2015] in the following way:

1. **Determinism**
   Both full determinism and stochasticity must be supported. Non-determinism can be implemented with custom transition functions. The framework also provides the option of partial stochasticity out-of-the-box, such as in the creation of objects (start values can be randomized) and in sensor readings.

2. **Ergodicity**
   This controls the degree to which the agent can undo things and get second chances. The framework imposes no restrictions on this other than a fundamental rule: Expended time and energy cannot be un-expended. If the agent spends time or energy doing the wrong thing, that
time and energy will still have been spent and the task-environment needs to be reset in order to
give the agent a second chance with regard to the energy and time expenditure. Task-environment
designers have full control over what states are reachable (as long as time and energy are ignored).

3. **Controllable Continuity**
   This point notes that it is crucial to allow continuous variables, and that the degree to which
   continuity is approximated should be changeable for any variable. The nature of all objects in the
   framework are continuous variables, discretized only by floating-point inaccuracies by default. It
   is possible use the **Sensor** class to further discretize (or distort) any accessible variables. It is
   also possible to tweak the time resolution of the simulation.

4. **Asynchronicity**
   The framework description does not explicitly address **asynchronicity** but the modular design
   accounts for the possibility of asynchronous actions. Neither synchronicity nor asynchronicity
   is forced upon the user — the user is free to make controller interactions synchronous or asyn-
   chronous.

5. **Dynamism**
   The framework allows both static environments and dynamic environments, but all dynamic
   behavior must be programmed in by the designer of the task-environment.

6. **Observability**
   The observability of task-environments is determined by the interface between the environment
   and the controller interacting with it. **Sensors** are the primary control for observability in the
   framework. See [3.1.1 on Sensors. Sensors can be tuned to tune the observability of a task-
   environment by distorting the value and/or rounding the result off to a specified number of
   significant digits.

7. **Controllability**
   Controllability is the control that the agent can exercise over the environment to achieve its
   goals. The controllability of the task-environment is controlled with the exposure of **Motors** to
   the controller. See [3.1.1 on Motors. By modifying motor properties and interactions between
   motors (specifically in custom transition functions), the controllability of a task-environment can
   be tuned.

8. **Multiple Parallel Causal Chains**
   Multiple parallel causal chains are supported because any object could individually start a causal
   chain (provided that transition functions describing object interactions are present).

9. **Number of Agents**
   The framework does not restrict the number of agents nor what interactions can take place. Even
   if multiple agents have access to the same **motor** objects, the framework regards the most recent
   setting to be the current setting. **However, it should be noted** that until asynchronicity is fully
   integrated into the framework, there is significant external overhead in coordinating multiple
   agents in a single task-environment.

10. **Periodicity**
    The framework does not specifically tackle periodicity, which can be controlled implicitly by
    configuring the behaviors of task-environments. No explicit tuning is provided other than giving
    task-environment designers the ability to implement transition functions for this purpose.

11. **Repeatability**
    The framework comes very close to creating fully repeatable task-environments, as long as the
    same random seed is used. However, agents and sensors must use their own random number
    generators (and seeds) to avoid tampering with task-environment repeatability. This property
cannot be fully considered satisfied due to the possibility of inconsistent ordering in the data structures used by the framework. In order to circumvent this, strict iteration order must be enforced in the framework.

It seems clear that although the framework description does not explicitly cover all of the properties, the design of the framework lays a foundation for features supporting these properties to be easily implemented.
Chapter 5

Use Cases

Reinforcement learning is an area of machine learning in which software agents take actions in order to maximize reward. Since there are no currently existing AGI systems at our disposal, we limit our testing of the framework to evaluations of reinforcement learning algorithms and domain-specific controllers. This chapter will reveal that results are rather positive regarding potential for scaling, although future work is obviously needed.

Some basic agents were implemented to demonstrate FraMoTEC’s capabilities and to test the framework as an evaluation tool. In this chapter, we will show results from evaluating the current implementation of the framework that was discussed in chapter 4, examining domain specific agent evaluation and scaling in multi-dimensional task-environment evaluations.

In section 4.2, we discussed some motivations for the work outlined in chapters 3 and 4 and presented an example of a plot demonstrating time and energy requirements for agents’ solutions to some task. In this chapter, we will demonstrate some of the current features described in chapter 4 and show that we can already use the framework for rudimentary agent and task-environment evaluations and comparisons. The presented plots in this chapter also demonstrate the ease at which relevant data can be gathered and presented by the framework.

By demonstrating the implemented prototype of FraMoTEC and its current capabilities, we hope to show the potential power of a fully-implemented version of the framework. We will do this by first comparing a SARSA agent in two task-environments, followed by a comparison of two domain-specific agent implementations in one task-environment and finally a comparison of two task-environments that differ only in the number of dimensions. Although we will not delve into what the results of these specific examples represent, we will hint at what kind of information can be gauged from the evaluations.

Our early Task Theory allows dissection of tasks into dimensions that must be controlled; simpler tasks contain only a few dimensions to be controlled sequentially; more complex tasks have a large number of dimensions, some of which must be controlled simultaneously. We will begin evaluating reinforcement learning in simple task-environments, moving onwards to domain-specific controllers and finally we will evaluate a domain-specific controller in complex (multi-dimensional) task-environments.
5.1 Q-Learning / SARSA

A SARSA agent was implemented in order to test the framework. Although reinforcement learners are quite far from any AGI-aspiring systems, they still serve as some of the most basic learners that can be situated in environments such as those constructed with the framework. SARSA learners are generally better suited for online learning, which is why SARSA was favored over a more standard Q-Learner.

The agent was introduced to two similar environments. The first environment had the goal of moving the position object into goal_position with a tolerance of 5, with 5000J and 60 seconds as the maximum expendable time and energy (essentially, the 1D drag racing example in 3.2.1). The second environment expanded upon this environment, requiring a “plotter” to be activated when the position is correct — both goals needed to be satisfied to consider the task solved (essentially the plotting example in [3.2.1]). An additional transition in the second environment locked the position while the plotter was activated.

5.1.1 Agent Implementation

An agent was implemented with an underlying SARSA reinforcement learning algorithm. The state exposed to the agent was an n-tuple of all sensor readings along with the velocity of one of the objects in the model. A scoring function was implemented to determine reward.

Reinforcement learners generally learn slower as the state × action space increases, therefore the agent enumerates the available actions as the setting of a single motor at one of three power levels: (i) 0 (ii) $P_{\text{max}}$ and (iii) $-P_{\text{max}}$. We experimented with an agent that included the settings (iv) $\frac{P_{\text{max}}}{2}$ and (v) $-\frac{P_{\text{max}}}{2}$, but we found that these settings unnecessarily crippled the agent and removed them.

The agent implements a method perform(self, dt) that creates an experience for the agent by: (a) setting the current state (b) selecting and executing the reward-maximizing action (c) ticking the simulation by dt seconds (d) rewarding the learner based on the value of the scoring function and the new state. This method is called repeatedly in the evaluation, see section 5.1.3.

5.1.2 Task-Environments

Two simple task-environments were constructed in which to test the SARSA agent, the latter being an expansion on the first. These environments are described in 3.2.1. The first environment will be referred to as ’1D drag racing’ and can be described like so (see figure 5.1 for a visual representation):

- One object: position
- One fully reversible motor affecting position with 200W maximum input

1 This was implemented as either the last instance in the goal_vars collection or the first instance with a corresponding target goal value greater than 10.
2 Implemented as $-\sum_{i=0}^{N} |s_{\text{object}_i} - s_{\text{goal}_i}| - \epsilon_i$ where $s$ represents the position (value) of either the object or the goal associated with a Goal in the task-environment’s solution.
3 SARSA (and other Q-learning algorithms) stores its policy in a Q-matrix representative of state × action → reward mapping, increasing the number of actions significantly increases the search space (and number of actions required to fully populate the Q-matrix).
Figure 5.1: Instance diagram of the 1D drag racing task-environment. Objects are represented as circles, systems are represented by rounded rectangles and goals are represented by parallelograms.

- One sensor for position
- Goal: position within $\text{goal\_position} \pm \text{goal\_epsilon}$
- Max time and energy: 60 s, 5000 J

The second environment will be referred to as '1D locking plotter' and can be described like so:

- Two objects: position and plot_it
- One fully reversible motor affecting position with 200 W maximum input
- One non-reversible motor affecting plot_it with 5 W maximum output
- One sensor for each object
- New transition function: If $\text{plot\_it} > 0.5$: position is locked, otherwise it is unlocked.
- Goal prerequisite: position between $\text{goal\_position} \pm \text{goal\_epsilon}$
- Goal: $\text{plot\_it} = 1 \pm 0.1$
- Max time and energy: 60 s, 10000 J

4 One could imagine a solenoid with an off button.
The second task-environment increases the task difficulty when compared to 1D drag racing by adding a new object (complete with sensor and motor), changing the behavior (with the transition function that locks the position object) and by expanding on the original goal. Figures 5.1 and 5.2 depict visual representations of the two environments and their differences. Note that the position object is initialized in a random position within a specified range, introducing some variety for evaluating deterministic controllers.

5.1.3 Evaluation

The SARSA agent was evaluated in the two task-environments described in section 5.1.2. The evaluations were identical:

First, the agent is tasked to solve some training environments, which are copies of the target environment, except with a more favorable starting position. The training environments gradually get more difficult by increasing the distance between the starting position and the goal. Once this training is complete, the agent gets 200 chances to satisfy the goal(s) in the original task-environment. To visualize the data, we used the result plotting method mentioned in section 4.5. Figures 5.3, 5.5 and 5.6 show the results for the 1D drag racing task-environment. Figures 5.4, 5.7 and 5.8 show the results for the 1D locking plotter task-environment. Note that the agent continues to learn by creating
Figure 5.3: Resulting plot for 200 evaluations of a SARSA agent in the 1D drag racing task-environment after having received some basic training in easier task-environment variants. (The blue line represents the energy required to complete the task within $x$ seconds. The red line represents the maximum amount of expendable energy due to motor power limitations. Each data point represents a single evaluation. The shading (or coloring) of each data point indicates the order. Lighter shading (green) precedes darker shading (red)).

experiences during the evaluation (i.e. learning is not “switched off”). The evaluation works as follows:

- While the task is not solved:
  1. If the task has been failed, stop
  2. Invoke the agent’s perform method (with $dt$ set to 0.25s, see section 5.1.1).
- Finally, report time and energy usage (and indicate if the task failed).

Figures 5.3 through 5.8 indicate that the SARSA agent becomes increasingly capable of solving the 1D drag racing task-environment, but that adding a layer of complexity drastically decreases the number of successful evaluations. This is what we would expect knowing the limitations of Q-learning agents (and how badly they scale with added complexity).
Figure 5.4: Resulting plot for 200 evaluations of a SARSA agent in the 1D locking plotter task-environment after having received some basic training in easier task-environment variants.

Figure 5.5: Energy/time usage progression plot for 200 evaluations in the 1D drag racing environment. The size of the markers indicate the time taken to complete the task. Lighter markers represent failed evaluations.
Figure 5.6: Time/energy usage progression plot for 200 evaluations in the 1D drag racing environment. The size of the markers indicate the energy expended to complete the task. Lighter markers represent failed evaluations.

Figure 5.7: Energy/time usage progression plot for 200 evaluations in the 1D locking plotter environment. The size of the markers indicate the time taken to complete the task. Lighter markers represent failed evaluations.
5.2 Agent Comparison

5.2.1 Controller

In order to demonstrate how different agents can be compared just as different environments can be compared, a custom agent implementation was compared with the SARSA implementation in the 1D locking plotter environment. The custom agent roughly implements the following algorithm in the perform method:

- Compute distance between position and the corresponding goal
  - If the distance is small enough, deactivate the position motor and activate the plot_it motor.
  - If the distance is positive, maximum power to the position motor and deactivate the plot_it motor.
  - If the distance is negative, maximum negative power to the position motor and deactivate the plot_it motor.

- Tick the simulation by dt seconds

It should be obvious that the above algorithm is specifically tailored to outperform the SARSA agent, as it includes domain knowledge which the SARSA agent would need to come up with on its own. Two of these agents were evaluated, the difference between them being that the improved version deactivated the position motor prematurely to compensate for momentum.
Figure 5.9: Resulting plot for 200 evaluations in the 1D locking plotter environment using an agent with a domain-specific implementation. Data points are overlapping in this figure, and therefore some early data points are not visible. *(The blue line represents the energy required to complete the task within x seconds. The red line represents the maximum amount of expendable energy due to motor power limitations. Each data point represents a single evaluation. The shading (or coloring) of each data point indicates the order. Lighter shading (green) precedes darker shading (red)).*

Figure 5.10: Resulting plot for 200 evaluations in the 1D locking plotter environment using an agent with an improved domain-specific implementation. All 200 data points are within the visible area of the graph (overlapped by neighboring datapoints).
Figure 5.11: Energy/time usage progression plot for 200 evaluations of a domain specific implementation in the 1D locking plotter environment. The size of the markers indicate the time taken to complete the task. Lighter markers represent failed evaluations.

Figure 5.12: Time/energy usage progression plot for 200 evaluations of a domain specific implementation in the 1D locking plotter environment. The size of the markers indicate the energy expended to complete the task. Lighter markers represent failed evaluations.
Figure 5.13: Energy/time usage progression plot for 200 evaluations of an improved specific implementation in the 1D locking plotter environment. The size of the markers indicate the time taken to complete the task. Lighter markers represent failed evaluations.

Figure 5.14: Time/energy usage progression plot for 200 evaluations of an improved specific implementation in the 1D locking plotter environment. The size of the markers indicate the energy expended to complete the task. Lighter markers represent failed evaluations.
5.2.2 Results

The difference between the standard domain-specific implementation and the improved counterpart is only in the distance from the goal at which the controller deactivates its motor (this naturally implies that the “improved” implementation could be worse-off in task-environments with a shorter distance to travel). The plots clearly depict exactly what we should expect from comparing these implementations — figure 5.9 shows that the task-environment is consistently solved with similar time and energy expenditure. Figure 5.10 shows that the improvement to the domain-specific implementation results in a significant reduction of time and energy expenditure for tasks in this domain. As before, our plots give us a nice overview of the agents’ performance in task-environments from identical domains. Figures 5.11 through 5.14 are included to show detail, but since the agent does not use *learning*, the performance is the same for each evaluation (with only slight differences in results attributable to differences in the environment instances of the domains).

5.3 N-Dimensional Task Comparison

While flat 1- and 2-D tasks are suitable for simple reinforcement learners, more advanced learners and controllers would call for tasks with more dimensions, whether simultaneously or sequentially controlled. Here we consider every Motor that necessarily must be controlled (at least indirectly) to accomplish some goal a dimension (requiring only independence, not orthogonality); to elaborate on this — the 1D plotter environment from example 3.2.1 actually has two dimensions: One for the movement on what we would traditionally call the $x$ axis and one for the movement on the $z$ axis (note that the relationship between these imaginary axes has not been defined). Here we explore how task consisting of multiple dimensions can be constructed in the framework.

5.3.1 Task-Environment Domain

A generic N-dimensional task-environment generator is included in the samples as `sample_N_task`. The generator returns a task-environment with $N$ identical objects with a default starting position of $3 \pm 2$ (the difference is customizable) and a goal of reaching $10 \pm 0.5$ (the upper bound is set to 10, preventing any possibility of overshotting the goal). The environment system includes all $N$ objects (including a sensor for each one) and two systems: (i) a control system which contains two objects with associated motors and sensors and a transition function that sets the power level of some hidden motor to some value depending on the values of the objects in the control system. (ii) hidden motor system which ensures that activating the hidden motors for each of the $N$ variables results in that power usage being counted. The maximum time and energy specifications scale linearly with the number of dimensions.

The control system includes the motors that the agent should have direct access to. The main power motor determines how much power is input into the hidden motors while the selection motor determines which hidden motor is activated.

5.3.2 Controller

A simple controller was created to solve the class of task-environments described in the previous section. The algorithm is quite simple, the below should demonstrate the agents perform method ($dt$ was fixed
at 1):

- Activate the main power motor
- Determine the object that is furthest from the goal, call it \( \text{min}_o \)
- Determine the direction of power required to enable \( \text{min}_o \)'s actuator
- Activate the selection motor in the appropriate direction (towards \( \text{min}_o \))
- Tick the simulation by \( dt \) seconds

5.3.3 Results

Three variations of the N-dimensional task were attempted with 10, 20 and 50 dimensions respectively. Figures [5.15] and [5.16] show the trade-off plots for 10 and 20 dimensional task-environments. The trade-off plot was omitted for 50 the dimensional task-environment evaluation since the data points greatly shadow each other. Figures [5.17] through [5.22] show the time and energy usage of the implementations in the different domains along with any failures during the evaluations (colored orange in the plots). It should not come as a surprise that the task-environments with fewer dimensions were solved in less time and with less energy. Furthermore, we can see that increasing the number of dimensions also increases the number of failures, with zero failures in the 10 dimensional tasks, 19 in the 20 dimensional tasks and 41 failures in the 50 dimensional tasks. This tells us that the performance of this controller does not scale well with the number of dimensions in the task-environment.
Figure 5.16: Resulting plot for 200 evaluations in a generic 20-dimensional task-environment.

Figure 5.17: Energy/time usage progression plot for 200 evaluations in a generic 10-dimensional task-environment. The size of the markers indicate the time taken to complete the task. Lighter markers represent failed evaluations.
Figure 5.18: Time/energy usage progression plot for 200 evaluations in a generic 10-dimensional task-environment. The size of the markers indicate the energy expended to complete the task. Lighter markers represent failed evaluations.

Figure 5.19: Energy/time usage progression plot for 200 evaluations in a generic 20-dimensional task-environment. The size of the markers indicate the time taken to complete the task. Lighter markers represent failed evaluations.
Figure 5.20: Time/energy usage progression plot for 200 evaluations in a generic 20-dimensional task-environment. The size of the markers indicate the energy expended to complete the task. Lighter markers represent failed evaluations.

Figure 5.21: Energy/time usage progression plot for 200 evaluations in a generic 50-dimensional task-environment. The size of the markers indicate the time taken to complete the task. Lighter markers represent failed evaluations.
Figure 5.22: Time/energy usage progression plot for 200 evaluations in a generic 50-dimensional task-environment. The size of the markers indicate the energy expended to complete the task. Lighter markers represent failed evaluations.

5.4 Summary

In this chapter, we have outlined some use cases for the framework, demonstrating how it can be used to evaluate both agents and task-environments. We saw controllers compete against each other in the identical environments, but we also saw the same controller attempting to deal with different environments. By plotting the result data, we can see the relationship between the time and energy expenditure of an agent in the context of solving a task in an environment and determine the performance of the agent over time. This chapter also demonstrates some additional in-depth examples of task-environments.

The domain specific agents built on the SARSA agent, but did not make use of the learning component. Instead, the perform method was overwritten, leaving the rest of the agent code as it is. This facilitated connecting the agents to the framework, since that part of the framework is still immature.

It is clear that much of the work proposed in [2] has been done, although some work remains.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis we have presented ideas for a framework for modular task-environment construction along with a prototype implementation. FraMoTEC aims to facilitate the evaluation of intelligent control systems across the entire spectrum of AI sophistication on practical tasks. The implementation and supporting ideas rest on some assumptions about the ability to dissect tasks into small atomic components, that when put together in different ways can approximate physical tasks and task families of various kinds, as proposed in Thórisson et al. [2015].

Understanding fundamental properties of various types of task-environments and how they relate to each other would greatly facilitate comparisons of control systems along with quantifiable measurements of features such as learning speed, energy efficiency and scheduling.

Thórisson et al. [2015] outlines requirements for a hypothetical evaluation tool: facilitation of easy construction, procedural generation and in-depth analysis. FraMoTEC lays the groundwork for such an evaluation tool. The analysis capabilities of the framework are currently quite limited, although they could be expanded upon as task theory research advances. Easy construction is already possible although an intermediary modeling language would be desirable (see 6.2.2), as this would also pave the way for procedural generation of similar task-environments (the current implementation allows values to be randomized, but this does not make two randomly initialized instances fundamentally different).

Being a framework for task-environment construction and evaluation, FraMoTEC’s modular design allows for simple construction, composition, decomposition and scaling of task-environments. Adding, removing, or combining parts from task-environments enables complexity to be tuned such that it grows with the AI system under test. Section 4.6 covers an evaluation of FraMoTEC with respect to the requirements and desired tunable properties listed in Thórisson et al. 2015.

While a lot of work remains to be done, we believe that this framework will be able to eventually fulfill the requirements outlined in Thórisson et al. 2015 and significantly contribute to the field of AI evaluation, task theory and by proxy: AI itself.

Further steps towards a proper Task Theory have been taken in Thórisson et al. 2016.
6.2 Future Work

As mentioned throughout, FraMoTEC’s current implementation lays the groundwork for a task-environment construction and comprehensive evaluation toolkit that takes time and energy into account and meets the requirements in Thórisson et al. [2015]. This section discusses future work towards a fully implemented toolkit for comprehensive evaluation of AGI-aspiring systems.

6.2.1 Simulation

The current design of FraMoTEC incorporates the simulation engine into the task-environment model. Although this approach works very well for prototyping, future optimization can be facilitated by decoupling the simulator from the model at an early stage. The scope of simulation within the framework is currently quite limited — being used only for agent evaluation. Regardless, simulation is an important feature in the framework as it provides a means to evaluate and compare agents in task-environments.

The framework should allow high-fidelity modeling of 3D task-environments. Recall that the physics currently provided by the framework operate in single dimensions. Expanding the simulation engine to specifically cover 3D environments enables construction of task-environments with (hopefully) isomorphic real-world counterparts.

Finally, the considerations in 6.2.5 point towards potential for implementing an actor-based system to deal with asynchronicity, by for instance implementing the simulation engine as a reactive actor-based system operating on the model.

6.2.2 High-level Description Language

A highly useful addition would be a compact way to represent task-environments, such as with a task-environment description language. Coupled with the framework’s power of (de-)composability, task-units or ‘prefabs’ could be constructed and re-used between task-environments with ease. Thórisson et al. [2015]’s Proposal section (4) outlines the usefulness of such a description language.

The modular approach to modeling task-environments comes from the proto task theory partly discussed in section 2.4. A modular approach allows for easy construction and decomposition while facilitating analysis. Task-environments are created with python code which could be generated from a more high-level description language — although such a description language should support raw python code as well, in order to facilitate transition construction and design.

6.2.3 Supporting Tools

At this early stage, the software to supplement the framework has not been implemented (with the exception of the plotting software described in 4.5 which could use improvement). A suite of integrated supporting tools such as comprehensive data visualization and simulation bootstrapping software would go a long way in facilitating standardized evaluations of a wide range of AI systems.

Supporting tools related to the proposed high-level description language (see 6.2.2) would be indispensable. Tools for generating environment variants (possibly even with fixed agent body speci-
cations) and tools for static analysis of task-environment descriptions could be used to procedurally generate (and evaluate agents in) task-environments, the results of which could be used to comprehensively compare agents’ control strategies and profile their strengths and weaknesses in various domains.

### 6.2.4 Visualization

Directly following section 6.2.1, it would be highly desirable to visualize 3D task-environment models. Realtime monitoring provides a means to compare real world task-environments with their simutable counterparts.

We can also envision custom task-environments visualization techniques, perhaps implemented as filters on visible dimensions. A general approach such as this could be used to represent 3D task-environments without being tightly integrated with some underlying 3-D physics engine.

The framework currently includes plotting software to visualize agent evaluation data. Powerful data visualization can enable evaluation and/or comparison of various features “at a glance”, which might otherwise require careful data processing. See 6.2.3.

### 6.2.5 General Improvements

The framework currently suffers in its analytical capabilities due to a lack of a proper task theory. The ideas presented in Thórisson et al. [2016] are sufficient to prototype an evaluation framework, but more work needs to be done in this field in order to be able to comparatively analyze properties of tasks and how they relate to control systems.

A crucial improvement to the framework would be to properly implement asynchronicity on the framework side. The framework currently leaves asynchronicity to the user. Implementing proper asynchronicity opens the door for increasing the number of agents in arbitrary ways without issue, although it might cause problems in repeatability if random usage is not carefully controlled (although we have already mentioned that there are other problems with repeatability).

Another major concern is the facilitation of easy interfacing between controllers and task-environments. With task-environments embodying agents, a standardized method of interfacing with the environment is necessary to perform comprehensive comparative evaluation of control systems. Although some preliminary standards could theoretically be set, it might be prudent to base specifications for a standardized format on task theory (which, as mentioned in 2.4, has not been definitively fleshed out).
Bibliography


Daniel L Scholten. Every good key must be a model of the lock it opens, 2011.


Kristinn R Thórisson, Jordi Bieger, Thröstur Thorarensen, Jona S Sigurdardottir, and Bas Steunebrink. Why artificial intelligence needs a task theory — and what it might look like, 2016.


