A SUBSTRATE FOR ACCOUNTABLE LAYERED SYSTEMS

by

Bo Morgan

BS Electrical Engineering and Computer Science, 2004
MS Media Arts and Sciences, 2006
Massachusetts Institute of Technology

SUBMITTED TO THE PROGRAM IN MEDIA ARTS AND SCIENCES,
SCHOOL OF ARCHITECTURE AND PLANNING IN PARTIAL
FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY IN MEDIA ARTS AND SCIENCES
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

SEPTEMBER 2013

©2013 MIT. All rights reserved.

Signature of Author: _____
Bo Morgan

Certified by: _____
Joseph Paradiso, PhD
Associate Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by: _____
Patricia Maes, PhD
Alex W. Dreyfoos Professor of Media Technology
Associate Program Head
Program in Media Arts and Sciences

A Substrate for Accountable Layered Systems

by
Bo Morgan

Submitted to the program in Media Arts and Sciences
School of Architecture and Planning

in partial fulfullment of the requirements for the degree of

Doctor of Philosophy in Media Arts and Sciences
at the
Massachusetts Institute of Technology

September 2013

ABSTRACT

A system built on a layered reflective cognitive architecture presents many novel and difficult software engineering problems. Some of these problems can be ameliorated by erecting the system on a substrate that implicitly supports tracing the behavior of the system to the data and through the procedures that produced that behavior. Good traces make the system accountable; it enables the analysis of success and failure, and thus enhances the ability to learn from mistakes.

This constructed substrate provides for general parallelism and concurrency, while supporting the automatic collection of audit trails for all processes, including the processes that analyze audit trails. My system natively supports a Lisp-like language. In such a language, as in machine language, a program is data that can be easily manipulated by a program, making it easier for a user or an automatic procedure to read, edit, and write programs as they are debugged.

Constructed within this substrate is an implementation of the bottom four layers of an Emotion Machine cognitive architecture, including built-in reactive, learned reactive, deliberative, and reflective layers. A simple natural language planning language is presented for the deliberative control of a problem domain. Also, a number of deliberative planning algorithms are implemented in this natural planning language, allowing a recursive application of reflectively planned control. This recursion is demonstrated in a fifth super-reflective layer of planned control of the reflective planning layer, implying N reflective layers of planned control.

Here, I build and demonstrate an example of reflective problem solving through the use of English plans in a block building problem domain. In my demonstration an AI model can learn from experience of success or failure. The AI not only learns about physical activities but also reflectively learns about thinking activities, refining and learning the utility of built-in knowledge. Procedurally traced memory can be used to assign credit to those thinking processes that are responsible for the failure, facilitating learning how to better plan for these types of problems in the future.

Thesis Supervisor: Joseph Paradiso, PhD
Associate Professor of Media Arts and Sciences

A SUBSTRATE FOR ACCOUNTABLE LAYERED SYSTEMS

by

Bo Morgan

BS Electrical Engineering and Computer Science, 2004
MS Media Arts and Sciences, 2006
Massachusetts Institute of Technology

SUBMITTED TO THE PROGRAM IN MEDIA ARTS AND SCIENCES,
SCHOOL OF ARCHITECTURE AND PLANNING IN PARTIAL
FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY IN MEDIA ARTS AND SCIENCES
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

SEPTEMBER 2013

©2013 MIT. All rights reserved.

Certified by: _____

Marvin Minsky, PhD
Professor of Media Arts and Sciences, Emeritus
Professor of EECS, Emeritus
Thesis Reader

Certified by: _____

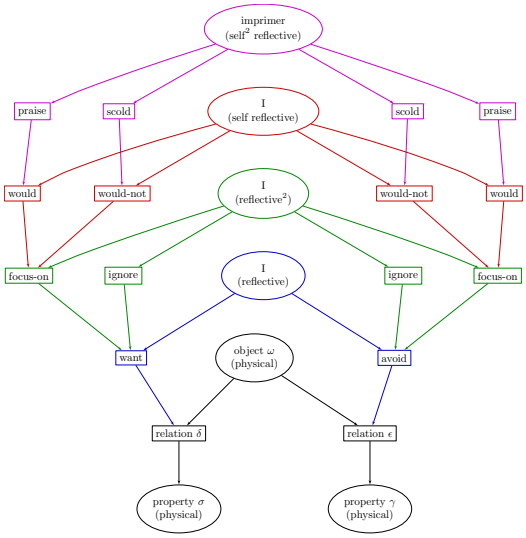
Gerald Jay Sussman, PhD
Panasonic Professor of Electrical Engineering
Thesis Reader

Certified by: _____

Michael T. Cox, PhD
Visiting Associate Research Scientist
University of Maryland
Thesis Reader

A SUBSTRATE FOR ACCOUNTABLE LAYERED SYSTEMS

BO MORGAN



PhD in the Media Arts and Sciences

Media Lab
Massachusetts Institute of Technology

June 2013

Bo Morgan: *A Substrate for Accountable Layered Systems,* PhD in the Media Arts and Sciences, © June 2013

Though there be no such thing as Chance in the world;
our ignorance of the real cause of any event
has the same influence on the understanding,
and begets a like species of belief or opinion.

— Hume (1748)

Dedicated to the loving memory of Pushpinder Singh.
1972 – 2006

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

Morgan, B.; “Moral Compass: Commonsense Social Reasoning Cognitive Architecture”; http://em-two.net/about; Commonsense Tech Note; MIT Media Lab; 2011 January

Smith, D. and Morgan, B.; "IsisWorld: An open source commonsense simulator for AI researchers"; AAAI 2010 Workshop on Metacognition; 2010 April

Morgan, B.; “A Computational Theory of the Communication of Problem Solving Knowledge between Parents and Children”; PhD Proposal; MIT Media Lab 2010 January

Morgan, B.; “Funk2: A Distributed Processing Language for Reflective Tracing of a Large Critic-Selector Cognitive Architecture”; Proceedings of the Metacognition Workshop at the Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems; San Francisco, California, USA; 2009 September

Morgan, B.; “Learning Commonsense Human-language Descriptions from Temporal and Spatial Sensor-network Data”; Masters Thesis; Massachusetts Institute of Technology; 2006 August

Morgan, B.; “Learning perception lattices to compare generative explanations of human-language stories”; Published Online; Commonsense Tech Note; MIT Media Lab; 2006 July

Morgan, B. and Singh, P.; “Elaborating Sensor Data using Temporal and Spatial Commonsense Reasoning”; International Workshop on Wearable and Implantable Body Sensor Networks (BSN-2006); 2005 November

Morgan, B.; “Experts think together to solve hard problems”; Published Online; Commonsense Tech Note; MIT Media Lab 2005 August

Morgan, B.; “LifeNet Belief Propagation”; Published Online; Commonsense Tech Note; MIT Media Lab; 2004 January

Eternity has nothing to do with time. Eternity is that dimension of here and now which thinking and time cuts out. This is it. And if you don't get it here, you won't get it anywhere.

— Campbell (1988)

ACKNOWLEDGMENTS

Push Singh for being a strong memory of a heroic friend and advisor.

Joseph Paradiso for his advice, encouragement and support at all times.

My readers for both being inspirations as well as tirelessly correcting countless drafts: Gerry Sussman, Michael Cox, and Marvin Minsky.

My immediate family: Greg Morgan, Carolyn Spinner, Paul Bergman, Virginia Barasch, and Leaf Morgan.

Those who worked on the project: Dustin Smith for the physical social commonsense kitchen simulation. Radu Raduta for the PID control loops for robot balance and movement. Jon Spaulding for the EEG amplifiers and pasteless electrodes. Gleb Kuznetsov for the first version of a 3D commonsense kitchen critic-selector layout planner. Jing Jian for being so good at finding critical bugs in Funk2 and for writing all of the learned-reactive resources for following recipes in the kitchen. Panupong Pasupat for the error-correcting subgraph isomorphism algorithm and the Winston difference-of-difference analogy algorithm. Mika Braginsky for the visualization of mental resources, agencies, and layers.

For their encouragement, advice, and inspiration: Walter Bender, Henry Lieberman, Patrick Winston, Whitman Richards, Aaron Sloman, Ed Boyden, Sebastian Seung, Ted Selker, Hugh Herr, and Rebecca Saxe.

The Responsive Environments Group: Joe Paradiso, Amna Cavalic, Josh Lifton, Mat Laibowitz, Mark Feldmeier, Jason LaPenta, Matt Aldrich, Nan-Wei Gong, Mike Lapinski, Gershon Dublon, and Nan Zhao.

The Common Sense Computing Group: Push Singh, Henry Leiberman, Walter Bender, Marvin Minsky, Felice Gardner, Betty Lou McClanahan,

Catherine Havasi, Rob Speer, Hugo Liu, Barbara Barry, Ian Eslick, Jason Alonso, and Dustin Smith.

The Mind Machine Project: Newton Howard, Gerry Sussman, Patrick Winston, Marvin Minsky, Neil Gershenfeld, Forrest Green, Kenneth Arnold, Dustin Smith, Catherine Havasi, Scott Greenwald, Rob Speer, Peter Schmidt-Nielsen, Sue Felshin, David Dalrymple, and Jason Alonso.

The Popes of GOAM: Dustin Smith, Scotty Vercoe, Dane Scalise, and Kemp Harris.

My adoptive thesis finishing family: Lindsey and baby Sudbury, Andrew Sudbury, Jon Sudbury, and Todd Rautenberg.

The following friends for laughing with me over the long haul: Dustin Smith, Grant Kristofek, Rhett Nichols, Mat Laibowitz, Nan-Wei Gong, Scotty Vercoe, Simon LaFlame, Dane Scalise, Hannah Perner-Wilson, Clio Andris, Nick Dufour, Jeff Lieberman, Mark Feldmeier, Matt Aldrich, Edwina Portocarrero, Mike Lapinski, Nan Zhao, Mako Hill, David Cranor, and Emöke-Ágnes Horvát.

This research was supported by an allocation of advanced computing resources provided by the National Science Foundation. Some of the computations were performed on Nautilus at the National Institute for Computational Sciences (`http://www.nics.tennessee.edu/`).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

An intelligent system thinks about a problem domain, learning the effects of its actions, constructing and executing plans to accomplish its goals. I will refer to these types of thinking about a problem domain as deliberative thinking. A reflective intelligence extends the deliberative intelligence by learning to accomplish goals in its own deliberative thinking process. Reflective thinking is sometimes referred to as "thinking about thinking" or "metacognition." A reflective intelligence can learn to select between different types of thinking that are appropriate for generating plans toward different types of goals. In this thesis, I present the Substrate for Accountable Layered Systems (SALS), an open-source software platform for the development of experimental reflective Artificial Intelligences (AI). A deliberative AI system consists of three processes: (1) perceptual data are generalized and categorized to learn abstract models, (2) abstract models are used to infer hypothetical states, i.e. states of future, past, or otherwise "hidden" variables, and (3) actions are chosen based on considerations of hypothesis dependent inferences. There are many approaches to machine learning that focus on this abstract 3-step closed-loop process of learning to control, such as: reinforcement learning (Kaelbling et al. 1996, Džeroski et al. 2001), game theory (Bowling & Veloso 2000, Rapoport 2001), and control theory (Simon 1982, Bertsekas 1995). The discipline of computational metacognition (Cox & Raja 2008, 2010) focuses on making at least two layers of closed-loop systems. Organizing the 3-step architecture within the metacognitive framework, deliberative thinking is modeled as a closed-loop learning algorithm that perceives and learns to control the external world, while reflective thinking is modeled as a second closed-loop learning algorithm that perceives and learns to control the deliberative thinking process. In this thesis, I present a tractable approach to reflective thinking that implies a linear scaling of time-complexity when extended to N layers of reflective control.

1.1 CONTRIBUTIONS

The four contributions of this thesis are:

1. *Emotion Machine Cognitive Architecture*: A computational implementation of the bottom four layers of the *Emotion Machine* six-layered theory of mind (Minsky 2006). The implementation contains a physical simulation that is controlled by a deliberative physical object-level reasoning layer with another reflective meta-level reasoning layer that learns to control the deliberative problem solving resources. The architectural primitives include resources that are organized into agencies that are organized into layers. The implementation includes five layers that are described in chapter 2: (1) built-in reactive thinking, (2) learned reacting thinking, (3) deliberative thinking, (4) reflective thinking, and (5) super-reflective thinking. The implementation leaves self-reflective and self-conscious layers of the Emotion Machine theory as future extensions of this research.

2. *Learning from Being Told Natural Language Plans*: A system needs a plan library. Plan libraries can be authored by humans as sequences of simple natural language sentences. The implementation includes the ability to interpret and imagine executing natural language commands by using analogies to natural language plans that it already knows. In this case, "being told" means that a natural language plan is programmed into the AI by the user. Interpreting a natural language plan involves parsing the English and generating a compiled program, along with imagined hypotheses about the program's effects. These hypotheses are based on what partial states the plan checks for in the control domain as well as rules learned from previous executions of similar actions in the past.

3. *Learning Asynchronously from Experience*: Executing plans in the external problem domain gives the system better expectations for what plans will actually end up doing when they are interpreted (section 3.2), imagined (section 3.6) and executed (section 4.1). I refer to this form of learning the effects of actions in the problem domain as learning from "experience." Many types of failures can occur when interpreting, imagining and actually executing ambiguous natural language plans, such as: expectation failures, interpretation failures, type failures, activation failures, and a variety of low-level system failures. These experiences of different types of failures inform reflective learning algorithms to subsequently predict these types of plan failures. Learning from experience is executed concurrently and asynchronously with the currently ex-

ecuting plan. The learning algorithm receives a stream of frame mutation trace events from the currently executing plan and uses this stream to learn abstract causal rule-based models. In this way, effects of physical and mental actions are learned through experience without slowing down the primary plan execution speed. Such failures are input to the reflective layer, which can learn to predict and avoid these failures in the future.

4. *Virtual Machine and Programming Language*: A concurrent and parallel virtual machine and low-level Lisp-like programming language provide the foundation upon which all of the above contributions have been implemented. The virtual machine includes native procedural tracing features that facilitate the automatic monitoring and control of many concurrently executing tasks. The virtual machine takes advantage of multiple CPU and multiple-core CPU hardware configurations. The programming language is a bytecode compiled language and all code for all contributions are open source (Appendix A).

1.2 A STORY OF REFLECTIVE LEARNING

Before getting into abstract generalizations of reflective thinking, let us consider the advice of Seymour Papert: "You can't think about thinking without thinking about thinking about something." Following this advice, consider the simple physical block stacking world, depicted in Figure 1, which is similar to the *Blocks World* planning domain (Winograd 1970). Imagine that the robot arm in this simple scenario is an entirely deliberative (non-reflective) AI that wants to accomplish the deliberative goal of a block being on a block. This AI has the capability of learning both from experience as well as from being told knowledge. The deliberative AI has been told a number of natural language plans for how to pick up and move around blocks in this problem domain. What types of thoughts might be going through the mind of this deliberative block stacking AI? The following

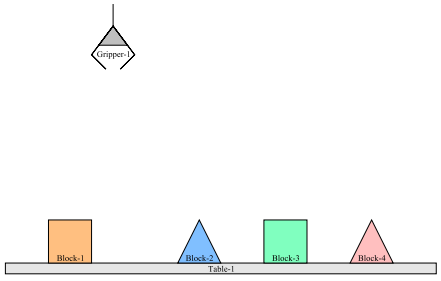


Figure 1: An example *Blocks World* problem domain.

story might be what this type of deliberative AI would be thinking in this block stacking domain:

> I want to accomplish the deliberative goal of a block being on a block. I must choose how to physically act. I have a number of plans that I have been told, but I don't know what they will do. I am focused on my most recently learned plan for how to physically act, which is called, "`stack a cube on a pyramid`." I have been told this plan in natural language, so I must interpret what it means if I am going to imagine executing it. If I can imagine a way that this plan could accomplish my deliberative goals, I will execute it. I will try interpreting and imagining the physical effects of my most recently learned plan for physical action, "`stack a cube on a pyramid`." I imagine that executing this plan will accomplish my goal of a block being on a block. I will stop imagining the rest of my plans and try executing this plan for physical action. I am picking up a cube and dropping it on a pyramid. The cube is falling off of the pyramid and onto the table. Oh no! A block is not on a block. My expectations have failed! A deliberative plan has failed. I will relearn the physical effects of my physical actions based on this new physical knowledge. The next time that I am dropping a block on a pyramid, I will expect the block to fall onto the table. Now, I must stop executing this plan and again choose how to physically act, given my new information.

In this story, the deliberative AI interprets and imagines executing plans based on reasoning by natural language analogies. Because the deliberative AI experiences a knowledge failure, the deliberative AI learns a better model of the effects of its physical actions. If this AI were reflective, it would be able to learn more from the deliberative failure of the AI in this story. A reflective AI not only learns the effects of physical actions but also learns the effects of deliberative actions, such as the effects of imagining the effects of a plan. How would a reflective AI approach thinking about accomplishing the same physical goal in the same problem domain? If the AI were reflective, the following story might be what it would think to itself as it tries to reflectively decide how to deliberate about plans for physical action:

> I want to accomplish the deliberative goal of a block being on a block. I also want to avoid plan failures. I have been told a number of reflective plans for deliberative action, but I

> don't know what they will do. I must choose a reflective plan for deliberative action. I am focused on my most recently learned reflective plan, which is called, "Find and execute a recently learned plan to accomplish my goals." I choose to imagine the effects of the various possible interpretations of this underspecified natural language plan on my deliberative knowledge. I can adapt other analogous plans to interpret this one, and I imagine that at least one interpretation will not lead to any deliberative plans having any execution failures, so I choose to execute this reflective plan to "find and execute a recently learned plan to accomplish my goals."

At this point in the story, the AI has decided on a plan of deliberative action. Notice that this story is very similar to the first story, except rather than deciding on a plan of physical action, the reflective planner is deciding on a plan for deliberative action. In this case, the "`find and execute a recently learned plan to accomplish my goals`" plan is an implementation of a planning algorithm within the natural planning language itself. This reflective plan is a sequence of mental actions rather than physical actions. The fact that the deliberative planning algorithm is written in the reflective planning language is one key aspect to the recursive nature of this approach to reflective learning and control. The reflective AI has a number of reflective plans for how to deliberatively plan, the method that the AI chose in this case tries to find a plan that it has been told most recently. So, the AI begins executing this reflective plan, which becomes the deliberative planning process that tries to find a plan for acting in the physical problem domain:

> I will try interpreting and imagining the physical effects of my most recently learned plan for physical action, "`stack a cube on a pyramid`." I imagine that executing this plan will accomplish my goal of a block being on a block. I will stop imagining the rest of my plans and try executing this plan for physical action. I am picking up a cube and dropping it on a pyramid. The cube is falling off of the pyramid and onto the table. A block is not on a block. Oh no! My deliberative expectations have failed! A deliberative plan has failed. Oh no, I was reflectively trying to avoid deliberative plan failures! My reflective expectations have failed! A reflective plan has failed.

At this point in the story, the AI has encountered two failures that will lead to two opportunities for learning the effects of both its physical

actions as well as its deliberative actions. If we consider what the AI might learn from this story, the AI might think:

> I have new support for the hypothesis that dropping a block while being over a pyramid leads to two blocks being on the table. I also have new support for the hypothesis that executing plans that try to accomplish the goal of a cube being on a pyramid may lead to an expectation failure when executed.

The fact that the AI failed at both the deliberative and reflective levels allowed the AI to learn two new sets of hypotheses: (1) about physical actions, and (2) about deliberative actions. The fact that more can be learned by adding a reflective layer to a learning algorithm is inspiration for researching reflective machine learning in those domains where physically acting is relatively costly while thinking is relatively cheap. Now, see how the reflective AI approaches reflectively thinking differently after it has learned from this initial experience:

> I still want to accomplish the deliberative goal of a block being on a block. I still also want to avoid my negative reflective goal by keeping the deliberative layer from having plans that have failed. I must choose another reflective plan for deliberative action. I am focused on my most recently learned reflective plan, which is called, "`find and execute a recently learned plan to accomplish my goals`." When I imagine executing this plan, I use my learned hypothesis that predicts that executing this reflective plan will lead to a failure in the deliberative knowledge. I choose to not execute this plan because it does not avoid my negative reflective goal. I focus on my next plan, "`find and execute an old plan to accomplish my goals`." I can adapt other analogous plans to interpret this reflective plan, and I have no hypotheses that predict that this plan will lead to any deliberative failures, so I choose to execute this reflective plan to "`find and execute an old plan to accomplish my goals`."

After this second round of reflective reasoning, the first reflective plan is considered and again imagined, but this time the reflective layer predicts that this reflective plan will lead to a failure in the deliberative layer because the deliberative conditions are so similar. For example, executing the same reflective plan in the context of the same deliberative goals is hypothesized to cause a deliberative failure. Because this conflicts

with the negative reflective goal to avoid deliberative failures, the first reflective plan is bypassed. The AI ends up considering another plan and selecting it for execution. The difference between these two plans is in how they organize their search through possible plans. The first reflective plan considers deliberative plans that it has learned most recently, while the second reflective plan considers deliberative plans that it has learned furthest in the past. In general, plans may be reflectively organized in more complicated mental structures, but in order to simply demonstrate my point, plans are organized in a doubly-linked list structure that goes forward and backward in time. One could imagine organizing plans by location, goal, or other equally important metrics in more complex data structures. Now that the AI has reflectively chosen a different way to deliberatively plan, the AI executes this reflective plan, which becomes the deliberative reasoning process:

> I will try interpreting and imagining the physical effects of my oldest plan for physical action, "`stack a pyramid on a cube`." I imagine that executing this plan will accomplish my goal of a block being on a block. I will stop imagining the rest of my plans and try executing this plan for physical action. I am picking up a pyramid and dropping it on a cube. The pyramid is now sitting on the cube. A block is on a block. Yay! Executing my oldest plan for physical action has accomplished my deliberative goal! Yay! I have also avoided my negative reflective goal to avoid deliberative plan failures!

So, finally, the reflective AI is happy to accomplish its deliberative goal. Note that the deliberative algorithm would have eventually found and executed the correct deliberative plan, if it had gone through all of its plans and imagined all of their effects, finally getting to its oldest plan, which happened to be the successful one. The advantage of the reflective learning algorithm is that it allows learning different ways of planning for dealing with different types of deliberative goals. Reflective planning allows learning how different plan representations, planning algorithms, and other deliberative knowledge is relevant to creating plans toward different types of deliberative goals.

1.3 LAYERS OF KNOWLEDGE

One tricky aspect of programming reflective learning algorithms is keeping clear distinctions between different layers of knowledge in the AI. Table 1 shows a few examples of physical, deliberative and reflective knowledge. Note that there is a strict hierarchy in the knowledge

Physical Knowledge	• A pyramid is on a cube. • A gripper is moving left. • A gripper is above a cube. • A cube is to the left of a pyramid.
Deliberative Knowledge	• Deliberative-Goal-1 is for a cube to be on a pyramid. • Deliberative-Plan-1 is to stack a pyramid on a cube. • Deliberative-Plan-1 fails for Deliberative-Goal-1. • Deliberative-Goal-2 is for a pyramid to be on a cube. • Deliberative-Plan-1 succeeds for Deliberative-Goal-2.
Reflective Knowledge	• Reflective-Goal-1 is to avoid a deliberative planner being focused on a deliberative plan that has failed in execution. • Reflective-Plan-1 is to find a *recent* deliberative plan to acheive one of my positive deliberative goals. • Reflective-Plan-1 fails for Reflective-Goal-1. • Reflective-Plan-2 is to find an *old* deliberative plan to acheive one of my positive deliberative goals. • Reflective-Plan-2 did not fail for Reflective-Goal-1.

Table 1: Examples of physical, deliberative and reflective knowledge.

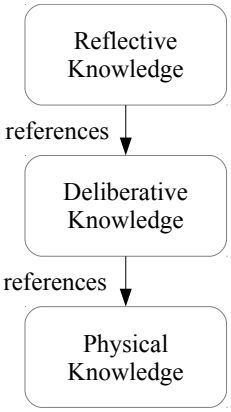


Figure 2: Three knowledge layers.

references between these layers. Physical knowledge cannot reference knowledge in other layers. An example of physical knowledge is: "`a pyramid is on a cube`." Physical knowledge is the representation of the problem domain. Deliberative knowledge cannot reference reflective knowledge but can reference physical knowledge. An example of deliberative knowledge is: "`a deliberative planner has the goal for a cube to be on a pyramid`." Deliberative knowledge includes positive and negative goals that specify which partial states of the physical knowledge should be sought or avoided. In addition to goals, deliberative knowledge includes plans, a planner, and potentially failures as well. Reflective knowledge can reference deliberative knowledge, which allows indirect reference to some deliberatively referenced physical knowledge as well. An example of reflective knowledge is: "`a reflective planner has the reflective goal for a deliberative planner to be focusing on a deliberative plan that is hypothesized to cause a cube to be on a pyramid`." Reflective knowledge is analogous to deliberative knowledge, but instead of being about accomplishing goals in physical knowledge, reflective knowledge is about accomplishing goals in deliberative knowledge. Figure 2 shows the hierarchical relationship between the physical, deliberative and reflective knowledge layers in the Substrate for Accountable Layered Systems (SALS), the open-source software platform for the development of experimental reflective Artificial Intelligences (AI) that I present in this thesis. In general, one can imagine that the recursive nature of SALS allows for any number of reflective layers to be added to the top of the reflective AI, resulting in *super-reflective* layers of learning planned control.

1.4 NATURAL LANGUAGE PLANS

SALS includes a simple but powerful planning language that is based on the interpretation of natural language plans. Plans are sequences of commands that can be created, mutated, and executed by a planner in order to accomplish goals. Deliberative plans are sequences of physical actions, while reflective plans are sequences of mental actions. The following is an example of a definition of one of the deliberative plans that the AI in the story could consider executing:

```
[defplan 'move slowly until over a cube'
  [plan-call [plan 'if a cube is to my left, move slowly
                   left until over a cube, otherwise if a
                   cube is to my right, move slowly right
                   until over a cube']]
  [plan-call [plan 'assert that a cube is below
                   me']]]
```

This expression defines a new deliberative plan. The `defplan` command is shorthand for "define plan." The first argument to the defplan expression is the name of the plan: "`move slowly until over a cube`." The body of the plan is the remaining sequence of expressions. The first expression in the body of this plan is to interpret and execute the natural language phrase beginning with "`if a cube`..." The second expression in the body of this plan is to interpret and execute the natural language phrase beginning with "`assert that`..." This plan attempts to position the AI over a cube and fails if a cube is not finally below the AI. If the planner wanted to find a plan to position the AI over a pyramid, this plan would not help unless it was slightly modified, replacing all mentions of "`cube`" with "`pyramid`." To help the planner to know what parts of plans might be analogously replaceable in this way, the SALS planning language includes optional natural language pattern matching templates and default frame variable bindings that can be specified for each plan definition. The following example shows how this simple plan can be generalized to allow positioning the AI over a range of shapes:

```
[defplan 'move slowly until over a cube'
  :matches ['move slowly until over a [? shape]']
   :frame [[shape 'cube']]
    [plan-call [plan 'if a [? shape] is to my left, move
                     slowly left until over a [? shape],
                     otherwise if a [? shape] is to my
                     right, move slowly right until over
                     a [? shape]']]
    [plan-call [plan 'assert that a [? shape] is below
                     me']]]
```

This generalized form of the original plan uses natural language variables that are specified with a question mark expression, "`?`" Note

that there are two optional arguments to the defplan expression in this example: (1) "`:matches`" and (2) "`:frame`." The optional "`:matches`" argument specifies a list of potential patterns that this plan may match as it is being interpreted. In this case, the variable expression "`[? shape]`" is allowed to replace the word "`cube`" from the original name of the plan. The optional "`:frame`" argument specifies the default natural language variable bindings. In this case, the "`shape`" variable is assigned the natural language phrase "`cube`" by default. In the body of the generalized form of the plan, all occurrences of cube have been replaced with the variable expression "`[? shape]`". Given this generalized form of the original plan, the planner can create a new analogous plan as an interpretation of the natural language phrase "`move slowly until over a pyramid`." In this way, plans can be communicated to the AI in a natural language form. The AI has been told a total of approximately one-hundred simple natural language plans, which can be adapted by analogy to provide interpretations for a variety of complex possible natural language plans, including recursive interpretations. The details of the planning language will be described in chapter 3.

1.5 LAYERS OF LEARNING

SALS includes an efficient relational learning algorithm in each layer of planned thinking. Relational learning in SALS is handled concurrently with the plan execution in that layer. In this way, as plans are executed at full speed, a trace of changes are produced and sent to a parallel event consuming algorithm that induces abstract partial states that are used to train a rule learning algorithm. The hypotheses that the rule learning algorithm creates are used to provide explanations as to the parts of the plan that have caused the traced changes. I have found that the separation of learning and planning into concurrent algorithms gives the learning algorithm the time to work more slowly, while the plan execution can be performed in bursts at near full speed.

The deliberative layer makes deliberative plans composed of physical actions to accomplish deliberative goals, while the reflective layer makes reflective plans composed of deliberative actions to accomplish reflective goals. Deliberative learning allows the AI to better predict the physical effects of deliberative plans for physical action, while reflective learning allows the AI to better predict the deliberative effects of reflective plans for deliberative action. The AI is learning at a reflective level when it thinks to itself, "I also have new support for the hypothesis that executing plans that try to accomplish the goal of a cube

being on a pyramid may lead to an expectation failure when executed." This newly supported hypothesis is a reflective hypothesis about deliberative actions and objects. The AI hierarchically assigns credit to the responsible parts of the executing plan, "`find and execute a recently learned plan to accomplish my goals`," that was currently executing at the time of the unexpected failure. This precondition for the execution of the plan is hypothesized to lead to the effects of this action in deliberative knowledge, "`a deliberative planner is focused on a plan that has failed`." The next time that the reflective layer is deciding whether or not the deliberative planner should execute a given plan, it can consider this new knowledge and predict whether or not executing the current plan will put the deliberative planner into a negative or positive deliberative goal state. I will discuss the details of SALS' parallel relational learning algorithm in chapter 4.

1.6 DOCUMENT OVERVIEW

Each of the four contributions of this thesis will be discussed in one of the following four chapters. My implementation of the bottom four layers of an Emotion Machine cognitive architecture is discussed next in chapter 2. In chapter 3, I will describe the natural language planning language that allows natural language plans to be told to the AI. This chapter will also discuss how the planning process interprets natural language plans by finding analogies to plans that the AI already knows. In chapter 4, I will describe how the AI learns from the experience it gains from actually executing its plans. This chapter will describe the necessary procedurally reflective components that have been used to attach complex time-intensive learning algorithms to quickly executing plans of action. To describe my last contribution, I will describe the SALS virtual machine and reflectively traced programming language in chapter 5.

Chapter 6 relates my AI to other contemporary cognitive architectures, approaches to reflective thinking, metacognition, and learning to plan, as well as other massively multithreaded computer systems. Chapter 7 evaluates the run-time performance of the SALS AI and shows a sub-linear increase in time-complexity for each additional reflective layer. In chapter 8, I discuss promising directions of future research for extending this architecture to learn at the top two layers of the Emotion Machine theory, the self-reflective and self-conscious layers. Finally, I discuss approaches to overcoming some of the current limitations in the SALS cognitive architecture.

Part I

CONTRIBUTIONS

2

EMOTION MACHINE COGNITIVE ARCHITECTURE

The SALS cognitive architecture is inspired by the bottom four layers of the *Emotion Machine* theory of human commonsense thinking described by Minsky (2006). SALS includes architectural primitives for defining different types of reflective AIs. The most basic component of the SALS architecture is an object called a *resource*. In general, a resource is any compiled procedure that can be executed by activating the resource. A resource is said to be *activated* if it is currently executing. A resource may be activated as an action in a plan. If a resource is currently executing, a *duplicate activation failure* results from an attempted additional activation of the resource. If a resource is not currently activated, a resource may be *suppressed*, which is a logical internal state of the resource that causes any attempts to activate the resource to result in a *suppressed activation failure*. Resources are usually assigned simple functions that do not do complicated or intelligent reasoning tasks themselves, but instead, resources are generally designed so that they perform primitive activities that can be combined in various ways to perform various resultingly complex tasks. For example, the simplest resources in the SALS AI that interact directly with the physical simulation do simple tasks like: "`start moving left`," "`start moving right`," "`stop moving`," "`reach`," and "`grab`." Resources may be activated or suppressed by plans in the layer above those resources. If a resource is suppressed and activated at the same time, this results in an activation failure that causes a plan to stop executing, so that it may be reflectively debugged at a higher layer. Some resources in SALS are referred to as *vital resources* because they are activated when the AI is initially created and they never complete execution. For example, one vital resource monitors any changes to a visual knowledge base and attempts to create a stable physical model of the world from these changes. Vital resources are usually used for processing streams of reflective trace events that monitor and learn from the effects of other executing resources.

Resources are grouped into collections called *agencies*. Agencies tend to be used for combining resources that are used for accomplishing similar types of goals. For example, the low-level resources that control the physical simulation are referred to as a *physical agency*. Resources that can be activated and used for solving problems are usually grouped into agencies that separate them from the vital resources. For example,

the vital resources that process low-level visual knowledge trace events are separated into a *sensory agency*.

Agencies are further grouped into collections called *layers*. The SALS AI consists of five layers of reflective control. These layers are:

1. *The Built-In Reactive Layer*
2. *The Learned Reactive Layer*
3. *The Deliberative Layer*
4. *The Reflective Layer*
5. *The Super-Reflective Layer*

Figure 3 shows an overview of the five layers of the reflective AI. The layers of the AI form cascaded control loops, where each layer controls the layer below. One would expect that in a real human there are cases where lower layers activate and suppress upper layers, such as hunger suppressing rational deliberation. In SALS, this is implemented as a reflective process that executes a deliberative plan that periodically checks a specific negative physical goal state exists and fails if it does, which would cause the reflective process to suppress the appropriate deliberative resources.

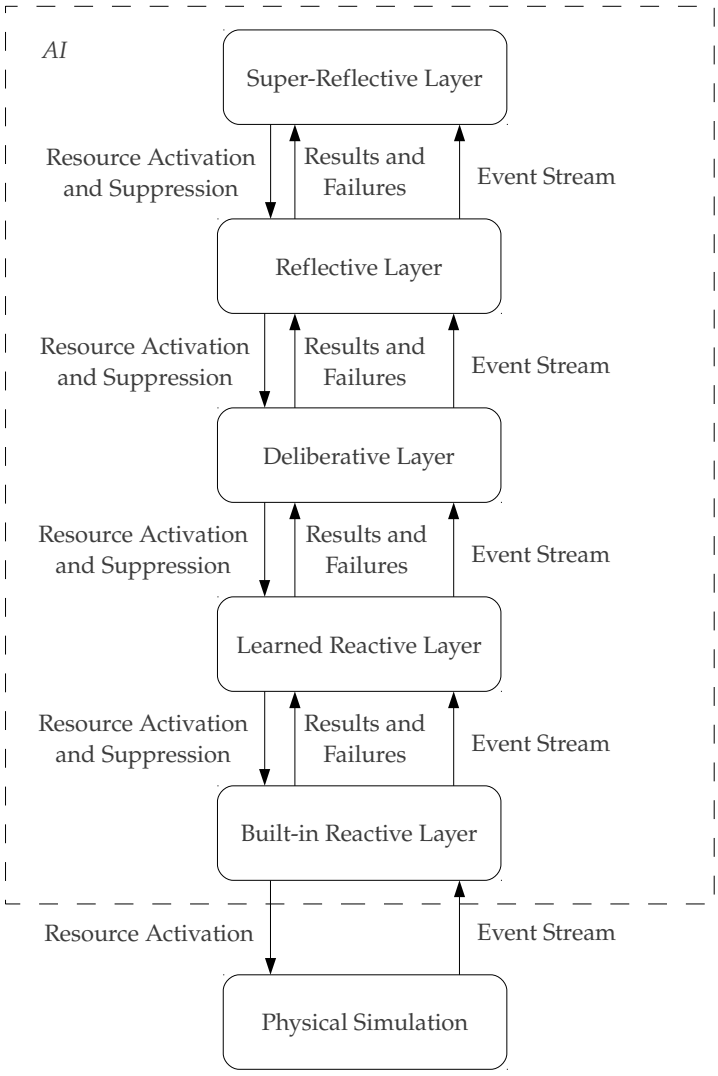


Figure 3: The five layers of the AI in relation to the physical simulation.

2.1 THE PHYSICAL SIMULATION

The physical simulation that is used to demonstrate the SALS AI, depicted in Figure 4, is similar to the *Blocks World* planning domain (Winograd 1970). The physical simulation is a 2-dimensional, deterministic, rigid-body physical simulation based upon a floating-point implementation of Newtonian physics ($F = ma$). The simulation is stepped with a time step of 0.1 seconds. The gripper can be controlled by the SALS AI to move left or right at one of two different speeds, fast (1 m/s) or slow (0.25 m/s). The simulation is meant to be a model of a continuous-time domain, rather than the logical propositional type of domain that is often used in other Blocks World simulations. The dexterous manipulation problem of the gripper picking up a block is simplified by simply allowing the gripper to magically grab a block when it touches the top of the block. When a square block is dropped on top of a triangular block, the square block will fall to the table in the direction of the center of mass of the square relative to the triangle. The physical simulation is programmed to recognize spatial relationships between objects, such as "`left-of`," "`below`," and "`inside-of`." As the physical simulation is stepped, the SALS AI receives a stream of relationship changes from the physical simulation.

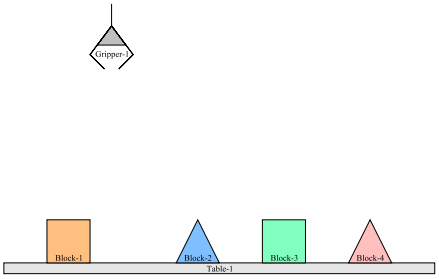


Figure 4: An example *Blocks World* problem domain, duplicated from Figure 1.

2.2 THE BUILT-IN REACTIVE LAYER

The built-in reactive layer is the layer of the AI that connects to the problem domain. In the example, the problem domain is a physical block stacking world. The lowest level action and perception agencies are in the built-in reactive layer, such as physical and sensory agencies. The built-in reactive physical agency contains resources that send asynchronous commands to the physical simulation, which means that these resources do not receive any response from the physical world, and thus do not report any types of failure or success status messages after they have completed being activated. The built-in reactive physical resources

are for the primitive actions of the physical simulation. The primitive actions that are exported by the physical simulation directly change its state. For example, the resource, "`start moving left`," can be thought of as applying 5 volts to a DC motor. The motor may or may not start turning, but the application of the voltage cannot be sensed as a failure. The "`start moving left`" built-in reactive resource puts the physical simulation into the state of trying to move left. There is no way that this very basic state changing function can fail in this sense. Any failure to actually move the robot arm to the left must be detected as an expectation failure at a higher level of reasoning that correlates actions with changes in sensory perceptions. The built-in reactive sensory agency receives a stream of change events from the state of the physical simulation. A *change event* is a frame[1] that consists of a removed attribute or property to or from a given frame object at a given time. Details of change events will be discussed in chapter 4, section 4.2. From this stream of changes, a visual knowledge base is constructed in the built-in reactive layer.

2.2.1 *The Visual Knowledge Base*

Knowledge bases in SALS consist of collections of interconnected frame-based objects. The visual knowledge base consists of visual objects that have a number of property slots with different sets of possible symbolic values:

- *type*: {"`block`", "`table`", "`gripper`"}
- *shape*: {"`cube`", "`pyramid`"}
- *color*: {"`red`", "`green`", "`blue`", "`brown`", "`white`", "`black`"}
- *movement-command*: {"`move-left`", "`move-right`", "`move-left-slowly`", "`move-right-slowly`", "`stop`", "`reach`", "`grab`", "`recoil`"}

These properties are meant to be those aspects of the physical world that the AI can see about a single object, including the type, shape and color of a visual object. Also, the AI can see what the robot arm, or "`gripper`", is currently commanded to do. The current activity of a gripper object is stored in the "`movement-command`" property slot. In addition to each

1 *frame*: A frame is a memory object that contains attribute or property values (Minsky 1975). Attributes of frames can be frames themselves, allowing for the definition of recursive memories.

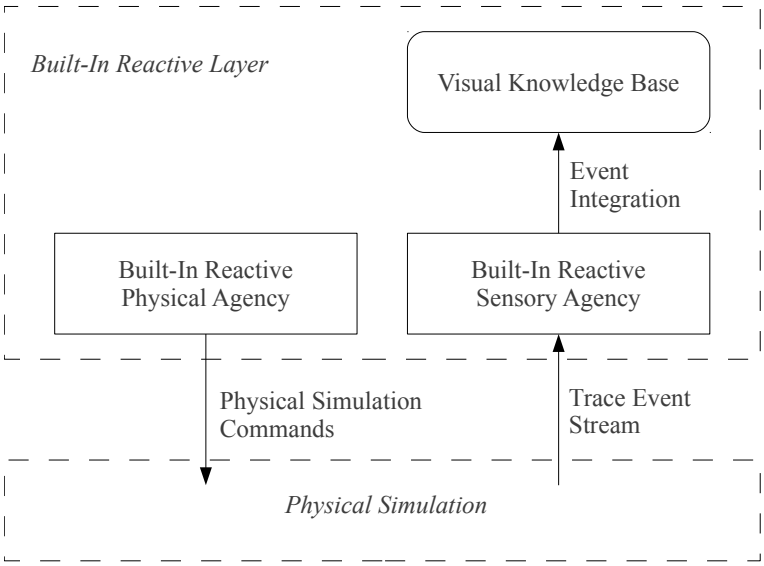


Figure 5: The built-in reactive layer communicates with the physical simulation.

visual object having properties, each visual object may have different types of relations to other visual objects that the AI can currently see.

- `on`
- `above`
- `below`
- `right-of`
- `left-of`
- `below-right-of`
- `below-left-of`
- `inside-of`

As can be seen in Figure 5, the built-in reactive physical and sensory agencies connect the AI to the physical simulation. Also shown is the visual knowledge base, the initial sensory knowledge base in the AI.

2.3 THE LEARNED REACTIVE LAYER

The learned reactive layer in SALS does not have direct access to the physical simulation. Instead, the learned reactive layer perceives the visual knowledge base in the built-in reactive layer and sends activation

or suppression commands to the built-in reactive physical agency resources. The learned reactive layer is similar to the built-in reactive layer because it also contains a *physical agency*. However, the learned reactive physical agency contains resources that execute compiled plans from the deliberative layer above. When these resources are activated, they execute plans that contain sequences of commands that end up either activating or suppressing built-in reactive resources in the layer below. Learned reactive physical resources can fail for a variety of reasons that will be introduced later when I present the details of plans and the planning process in chapter 3.

2.3.1 *The Physical Knowledge Base*

The learned reactive layer contains an agency called the *physical knowledge agency*. The physical knowledge agency contains resources that receive a trace event stream of any changes in the visual knowledge base. In partially observable environments, the physical knowledge base contains a representations of the physical world that is larger than the current visual knowledge base may contain as direct sensory knowledge. The block stacking domain is not a partially observable environment, so the physical knowledge base in this case is simply a reconstructed copy of the visual knowledge base. However, in IsisWorld (Smith & Morgan 2010), the partially observable physical problem domain, SALS utilizes the distinction between visual and physical knowledge as the physical knowledge base is larger than the partial view of knowledge provided by the visual knowledge base. In this dissertation, I will not describe the larger number of different types of objects with more complex relationships and properties that occur in the IsisWorld physical simulation. Because my focus is on learning to plan and learning to reflectively plan, the visual and physical details of the IsisWorld simulation would only confuse my point.

As can be seen in Figure 6, the learned reactive physical and physical knowledge agencies connect the higher layers of the AI to the built-in reactive layer. Also shown is the physical knowledge base, the focus knowledge base that the deliberative planning layer attempts to accomplish goals within.

The basic mechanism for perceptual abstraction in SALS is based on an object called a *partial state*. Because all knowledge in SALS is represented as frames with slots, these knowledge bases can be simultaneously represented as semantic graphs with objects and their symbolic properties as nodes of the graph, while slot names are consid-

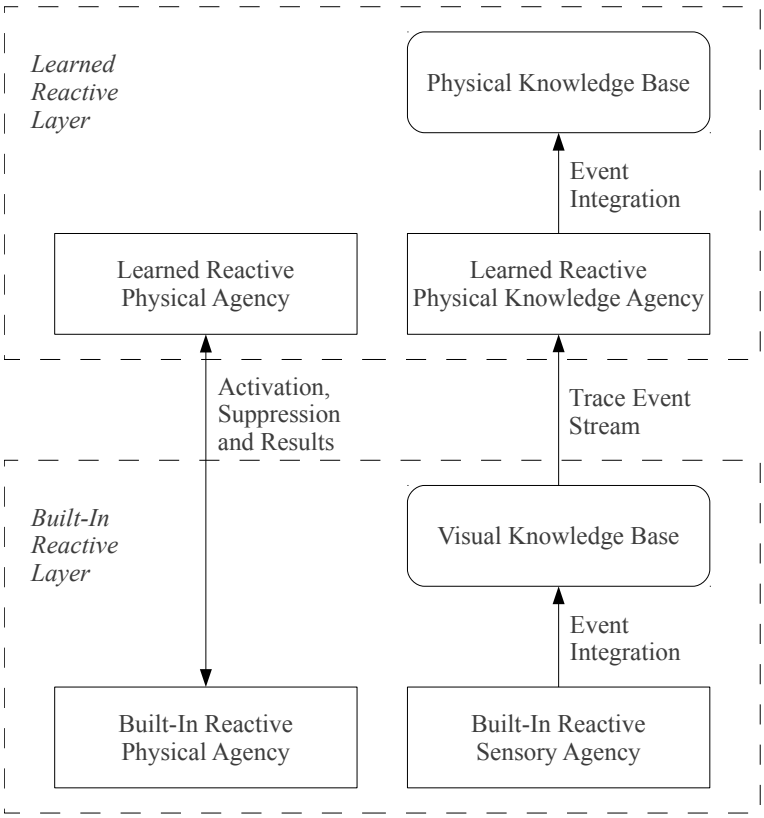


Figure 6: The learned reactive layer communicates with the built-in reactive layer.

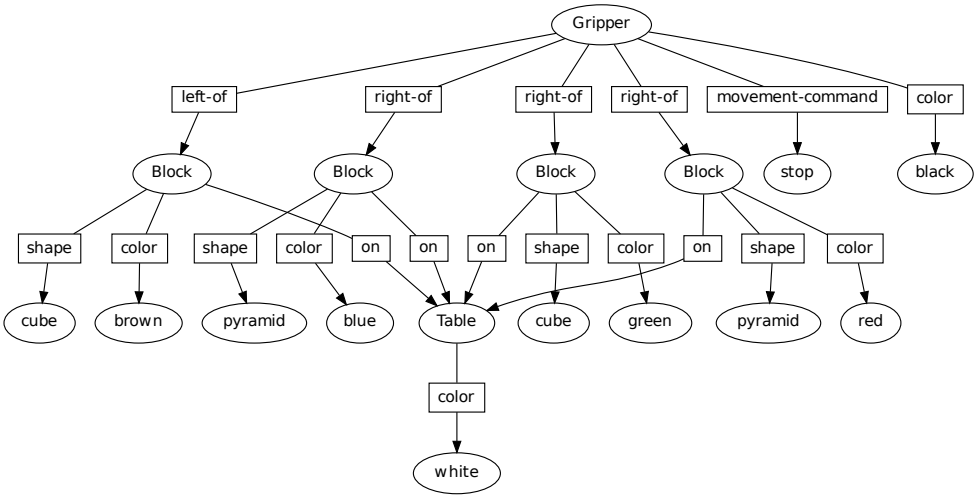


Figure 7: A graph representation of the physical knowledge base, where frame-based objects become interconnected collections of elliptical node labels and rectangular edge labels. This representation is consistent with the physical situation shown previously on page 38 in Figure 4.

ered the edges of the graph. Figure 7 shows a graph representation for the physical knowledge base. Given this graph representation of any SALS knowledge base, a partial state of a SALS knowledge base is any subgraph of one of these knowledge base graphs. The details of specific partial state object definitions and how the partial state abstraction process works, while avoiding the complexity of a general graph matching algorithm are described in chapter 4.

2.4 THE DELIBERATIVE LAYER

The deliberative layer is the first planning layer in the SALS cognitive architecture. The deliberative layer tries to accomplish goals that are partial states of the physical knowledge base in the learned reactive layer. Figure 8 shows an overview of the deliberative layer and its connections to the learned reactive layer below. The following are functional explanations of the labeled parts, A–G, in Figure 8:

A. Natural language plans can enter the *deliberative plan knowledge base* from outside of the AI by "being told" by a human user or

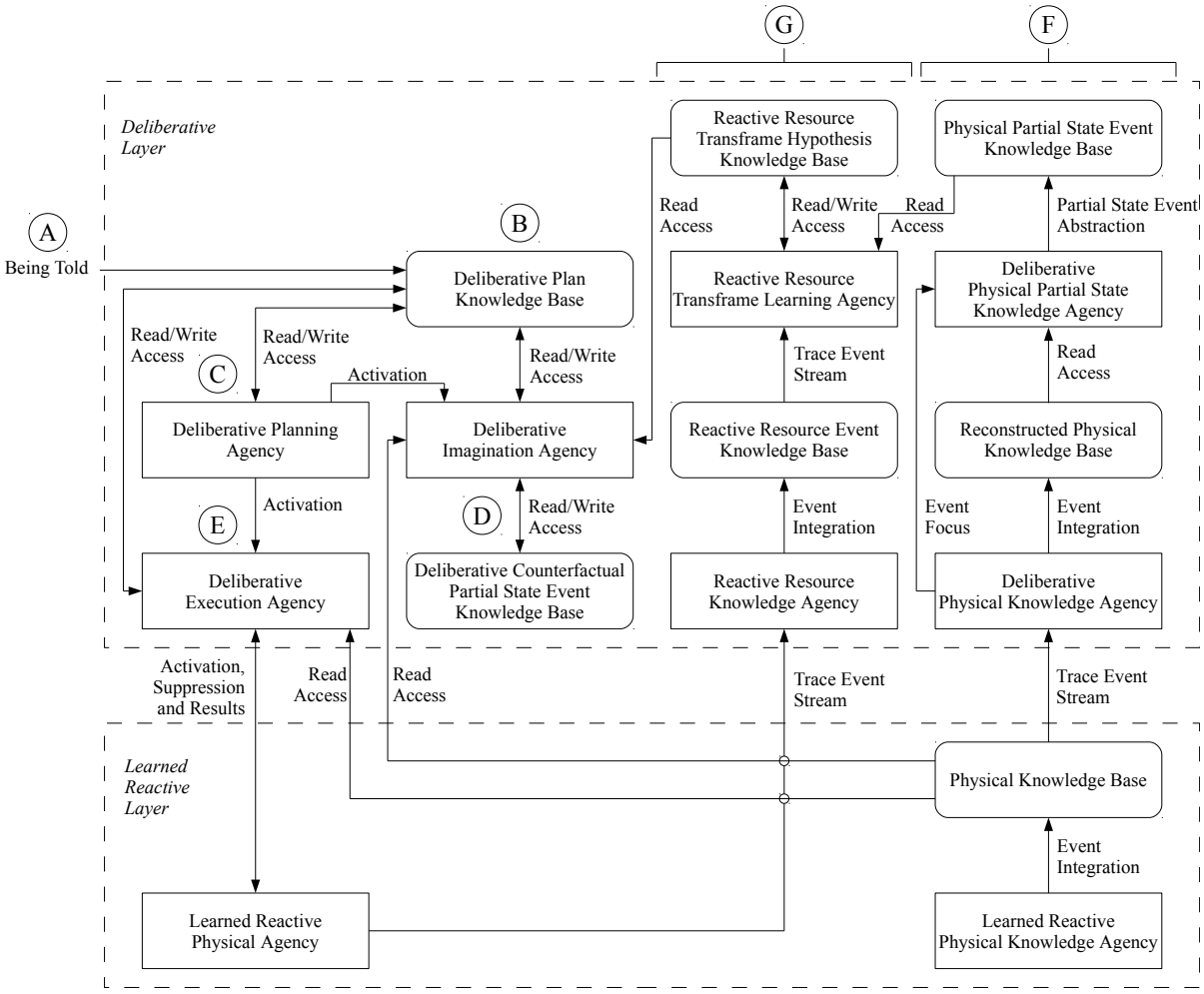


Figure 8: The deliberative layer and its connections to the learned reactive layer. See text in section 2.4 for descriptions of labeled functional areas, A–G.

another AI. One such natural language plan that has been told to the SALS AI is to "`stack a cube on a pyramid`."

B. The *deliberative plan knowledge base* is where all deliberative natural language plans for physical action are stored along with the state of the deliberative planning machine and plan failures. When natural language plans are told to the deliberative layer of the SALS AI, the plan is stored in the deliberative plan knowledge base. When plans are manipulated and new plans are created, these new plans are also stored in the deliberative plan knowledge base. In the example story presented in chapter 1, the deliberative planning machine focuses on a plan to "`stack a cube on pyramid`." At this point in the example story, the fact that the deliberative planning machine is focused on this plan is also stored as knowledge in the deliberative plan knowledge base: "`a deliberative planning machine is focused on a plan to stack a cube on a pyramid`." Details of the internal representation of the deliberative plan knowledge base will be described in subsection 2.4.1. The state of the deliberative plan knowledge base is reflected upon by the reflective layer, which will be described in section 2.5.

C. The *deliberative planning agency* contains the resources for planning activities that manipulate plans in the deliberative plan knowledge base as well as resources that in turn activate the resources in the neighboring deliberative imagination and execution agencies. The deliberative planning agency includes resources that cause the imagination of the effects of a plan in focus, change the planning focus, manipulate plans currently in focus, as well as cause the execution of plans currently in focus. The reflective layer, described in section 2.5, activates the resources in the deliberative planning agency to control the deliberative planning machine.

D. The *deliberative imagination agency* imagines the hypothetical future effects of executing deliberative plans for physical action. The *deliberative counterfactual partial state event knowledge base* is used as a scratchpad for storing these hypothetical future physical states. The current state of the physical knowledge base in the layer below is used as a starting point for the counterfactual knowledge created by the deliberative imagination agency. For example, when the deliberative planning agency focuses the planning machine on the plan to "`stack a cube on a pyramid`" and subsequently activates the deliberative imagination agency, the effects of the plan are imagined and the deliberative counterfactual partial state event

knowledge base subsequently contains the physical partial state for "`a cube to be on a pyramid`."

E. The *deliberative execution agency* executes plans by activating and suppressing resources in the learned reactive physical agency in the layer below. For example, when the deliberative planning agency focuses the planning machine on the plan to "`stack a cube on a pyramid`" and subsequently activates the deliberative execution agency, the body of the plan is executed, including activating resources in the physical agency to "`move right`," "`grab`," "`move left`," and "`drop`."

F. A column of agencies and knowledge bases abstract partial states from the physical knowledge base in the learned reactive layer below. Because partial state abstraction can be a slow process, this process is performed asynchronously based on a stream of change events. A detailed description of partial states and their asynchronous abstraction will be given in chapter 4, section 4.2. Abstracted partial state event knowledge is stored in the *physical partial state event knowledge base*. The abstraction of partial states is one of two types of asynchronous processing streams that constitute the SALS AI's ability to learn from the experience of executing plans.

G. A column of agencies and knowledge bases perform asynchronous learning of abstract causal rule hypotheses from physical agency resource execution preconditions. The advantage of an asynchronous learning algorithm is that it does not slow down the execution of plans in the deliberative layer. Historical versions of knowledge bases are reconstructed so that the slower learning algorithms can discover relevant patterns in this data for predicting the effects of actions. For example, when the deliberative execution agency executes the plan "`stack a cube on a pyramid`," the SALS AI learns that when "`a gripper is holding a cube`" and "`a pyramid is below a gripper`," the resulting state will *not* be "`a cube is on a pyramid`." The details of the asynchronous learning of abstract causal rules from the experience of executing plans will be described in chapter 4, section 4.3.

2.4.1 *The Deliberative Plan Knowledge Base*

The *deliberative plan knowledge base* is the heart of the deliberative layer, where all deliberative natural language plans for physical action are

stored along with the deliberative planning machine and deliberative plan failures. Natural language plans can enter the deliberative plan knowledge base from outside of the AI by "being told," which is a form of natural language programming, possibly in the form of an natural language expression from the user. The *deliberative planning agency* is the center of executive control in the deliberative layer. The deliberative planning agency manipulates plans and also activates the *deliberative imagination agency* and the *deliberative execution agency* when these plans should be imagined or executed. The deliberative imagination agency uses learned rules that map preconditions of learned reactive physical agency resource activations to changes that these resources cause to occur in the learned reactive physical knowledge base. The deliberative imagination agency uses the *deliberative counterfactual partial state event knowledge base* as a scratchpad that can store hypothetical future states of the learned reactive physical knowledge base if the appropriate learned reactive physical agency resources are activated. Once the deliberative planning agency has decided to execute a given plan, the deliberative execution agency is activated, which executes plans by activating and suppressing resources in the learned reactive physical agency in the layer below.

Similar to the visual and physical knowledge bases, the deliberative plan knowledge base also consists of collections of interconnected frame-based objects. The following are different possible property slots and symbolic values for the deliberative objects in the deliberative plan knowledge base:

- *type*: {"`plan`", "`planner`", "`execution-node`", "`failure`"}
- *has-been-imagined*: {"`true`", "`false`"}
- *default-slot-value*: {all natural language strings}
- *hypothesized-to-cause*: {all physical partial states}
- *positive-goal*: {all physical partial states}
- *negative-goal*: {all physical partial states}

These properties are examples of those aspects of the deliberative thinking layer that the AI can reflectively see about a single deliberative object, including the type, the default slot values of a natural language plan, and the goals of a planner. In addition to each deliberative object having properties, each deliberative object may have different types of relations to other deliberative objects that the AI can reflectively perceive. Some are these relations are as follows:

- *focus-plan*
- *execution-plan*
- *imagination-failure*
- *execution-failure*
- *start-execution-node*
- *previous*
- *next*
- *subnode*
- *supernode*

I will describe the details of the objects in the deliberative plan knowledge base in chapter 3, where I will describe learning from being told and the natural language planning process. A simplified graph representation of the deliberative plan knowledge base is shown in Figure 9. In this figure, knowledge labels A and B refer to the following different types of deliberative knowledge:

A. The state of the *deliberative planning machine* includes positive and negative physical goals as well as references to plans for physical action that the planning machine is currently focusing on or executing. For example, in the story presented in chapter 1, the plan

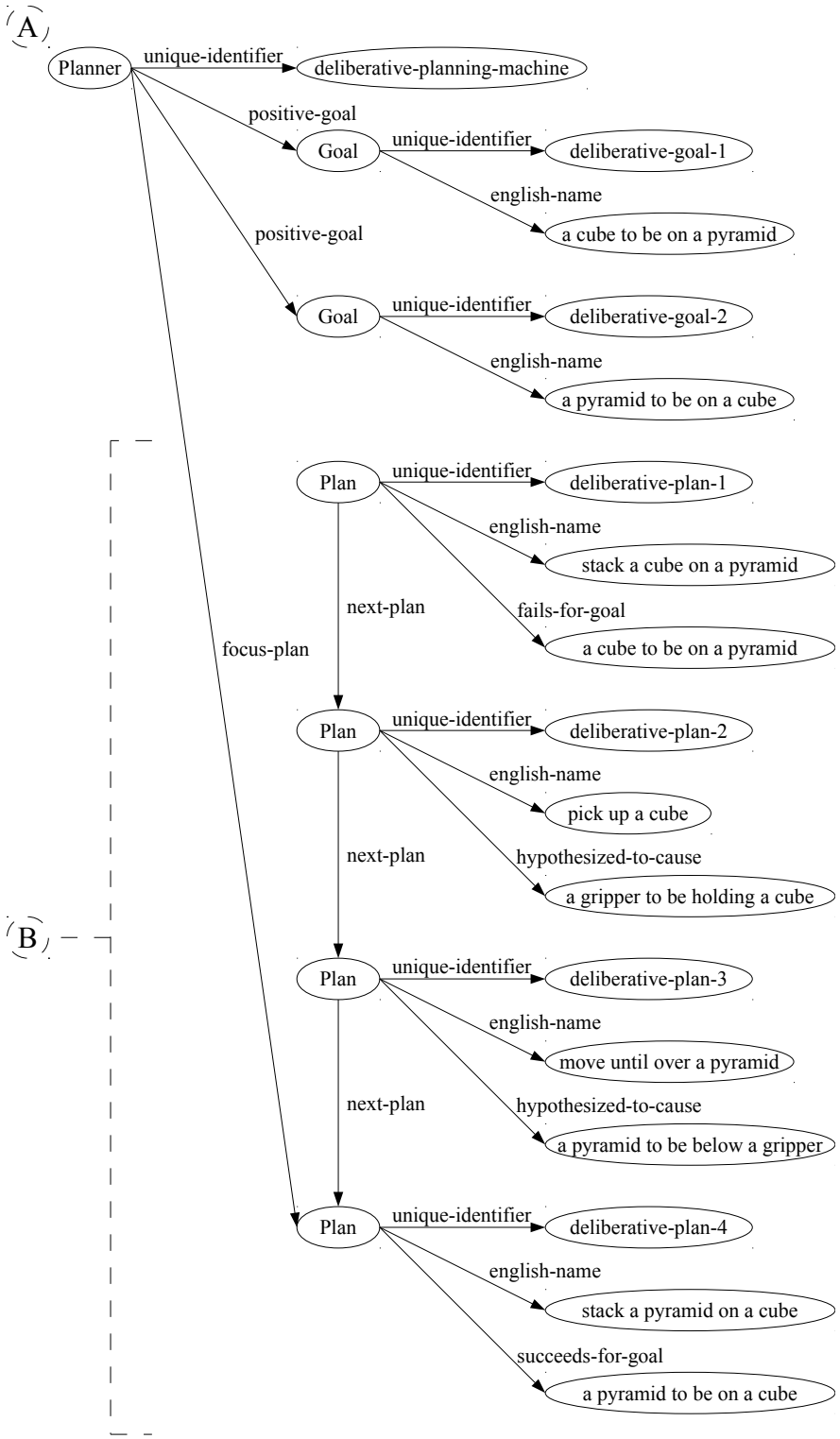


Figure 9: Part of the deliberative plan knowledge base represented as a graph. See text in subsection 2.4.1 on page 48 for descriptions of knowledge labels, A and B.

to "`want a block to be on a block`" is a reflective plan that has previously been told to the reflective layer of the SALS AI. The execution of this reflective plan causes the deliberative planning machine to be given two specific positive goals for either "`a cube to be on a pyramid`" or "`a pyramid to be on a cube`." At the end of the story, the reflective SALS AI successfully accomplishes its goal for "`a pyramid to be on a cube`" by first focusing on the last plan that it has been told and then executing this plan.

B. Representations of deliberative plans are organized into a linked-list structure that goes forward and backward in time. Plans that have been told to the SALS AI furthest in the past are at the beginning of the list. In the example story, the SALS AI uses this linked-list structure to organize its search through deliberative plans for physical action. Initially, the SALS AI considers plans from newest to oldest, which results in finding the plan, "`stack a cube on a pyramid`," which fails to accomplish its goal for "`a cube to be on a pyramid`." Finally, the SALS AI searches through deliberative plans from oldest to newest and this results in finding the plan, "`stack a pyramid on a cube`," which succeeds in accomplishing its goal for "`a pyramid to be on a cube`." At this point in the example, the SALS AI reflectively learns to apply a different planning method for the current goals of the deliberative planning machine.

2.4.2 *Asynchronous Learning*

To imagine the effects of executing plans, the deliberative layer learns abstract hypothetical models of the effects of learned reactive physical agency resource activations. These learned models are generated by a rule learning algorithm that predicts a hypothetical *transframe* (Minsky 1975) given the preconditions for an action. Transframes represent changes between one collection of frames and another collection of frames. The details of transframes in the SALS AI will be discussed in chapter 4, section 4.3. Asynchronous learning is implemented as two stages of stream processing. The first stage abstracts partial state events from a trace event stream composed of any change events that occur in the learned reactive physical knowledge base. The second stage receives a stream of activation and completion events from the learned reactive physical agency resources. Because both of these asynchronous stages process different types of event streams at different rates, the resulting knowledge bases are accurate for different points of time in the past.

Although the timing of the two-staged asynchronous learning in SALS complicates the implementation, the advantage is simple: the execution of learned reactive physical agency resources and the manipulation of the learned reactive physical knowledge base can both operate at full speed, while the slower learning algorithm can operate at its own pace. In practice, resource activations and knowledge changes occur in high-speed bursts, followed by periods of deliberation, which generally gives the slower learning algorithms time to catch up. The details of the SALS asynchronous learning algorithm will be discussed in chapter 4.

2.5 THE REFLECTIVE LAYER

While the deliberative layer focuses on learning the effects of physical actions to make plans to control the physical knowledge base, the reflective layer focuses on learning the effects of deliberative planning actions to make plans to control the deliberative plan knowledge base. This similarity is abstracted in SALS and is called a *planning layer*. The deliberative, reflective, and super-reflective layers are all instantiations of these planning layer cognitive architectural objects. A planning layer is an extension that can be added to any paired combination of a knowledge base and an agency of resources to learn how to use to control the partial states within the knowledge base. Figure 10 shows the reflective layer and its connection to the deliberative layer. While the deliberative layer focuses on learning about making plans to control the physical knowledge base, the reflective layer focuses on learning about making plans to control the deliberative plan knowledge base. This similarity is abstracted in SALS and is called a *planning layer*. The deliberative, reflective, and super-reflective layers are all instantiations of these planning layer cognitive architectural objects.

A. Natural language plans can enter the *reflective plan knowledge base* from outside of the AI by "being told" by a human user or another AI. One such reflective natural language plan that has been told to the SALS AI in the example presented in chapter 1 is to "`find an old plan to accomplish a goal`." Plans in the reflective plan knowledge base include different methods for how to find or create plans to accomplish different types of physical partial states, or deliberative goals.

B. The *reflective plan knowledge base* is where all reflective natural language plans for deliberative action are stored along with the state of the reflective planning machine and reflective plan failures.

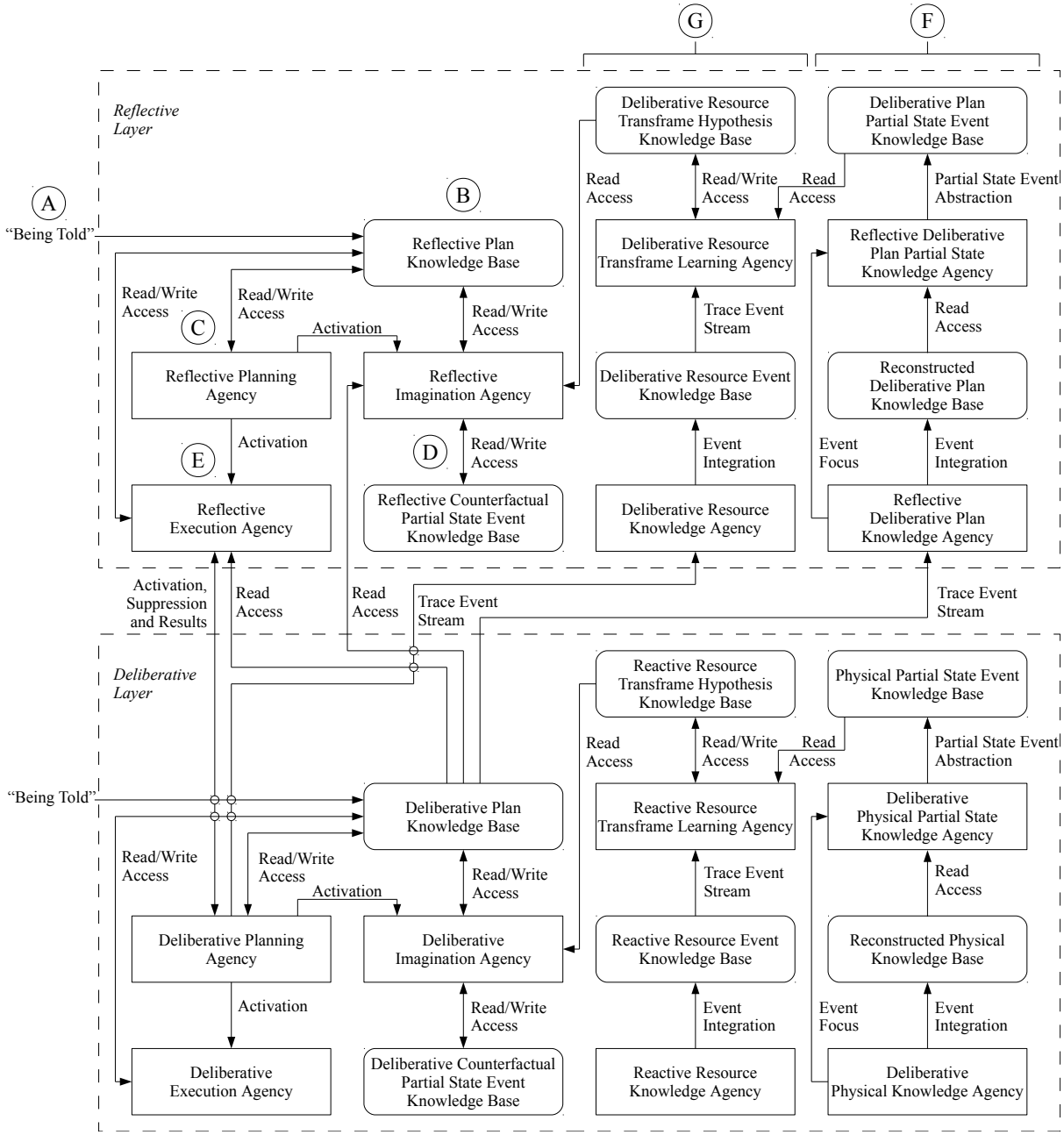


Figure 10: The reflective layer and its connection to the deliberative layer is analogous to the deliberative layer and its connection to the learned reactive layer, shown in Figure 7 on page 43. See text in section 2.5 for descriptions of labeled functional areas, A–G.

When natural language plans are told to the reflective layer of the SALS AI, the plan is stored in the reflective plan knowledge base. In the example story presented in chapter 1, the reflective planning machine focuses on a plan to "`find a recent plan to accomplish a goal`." At this point in the example story, the fact that the reflective planning machine is focused on this plan is also stored as knowledge in the reflective plan knowledge base: "`a reflective planning machine is focused on a plan to find a recent plan to accomplish a goal`." Details of the internal representation of the reflective plan knowledge base will be described in subsection 2.5.1. The state of the reflective plan knowledge base is further reflected upon by the super-reflective layer, which will be described in section 2.6.

C. The *reflective planning agency* contains the resources for reflective planning activities that manipulate plans in the reflective plan knowledge base as well as resources that in turn activate the resources in the neighboring reflective imagination and execution agencies. The reflective planning agency includes resources that cause the imagination of the effects of a reflective plan in focus, change the reflective planning focus, manipulate reflective plans currently in focus, as well as cause the execution of reflective plans currently in focus. The super-reflective layer, described in section 2.6, activates the resources in the reflective planning agency to control the reflective planning machine.

D. The *reflective imagination agency* imagines the hypothetical future effects of executing reflective plans for deliberative action. The *reflective counterfactual partial state event knowledge base* is used as a scratchpad for storing these hypothetical future deliberative states. The current state of the deliberative plan knowledge base in the layer below is used as a starting point for the counterfactual knowledge created by the reflective imagination agency. For example, when the reflective planning agency focuses the reflective planning machine on the plan to "`find a recent deliberative plan to accomplish a goal`" and subsequently activates the reflective imagination agency, the effects of the plan are imagined and the reflective counterfactual partial state event knowledge base subsequently contains the deliberative partial state for "`a plan has an expectation failure`." This prediction of plan failure does not require the deliberative layer to imagine the physical effects of executing physical actions; instead, plan failure is predicted

reflectively from the structure of the deliberative plan and its relationship to the deliberative planning machine, including the current deliberative goals.

E. The *reflective execution agency* executes reflective plans by activating and suppressing resources in the deliberative plan agency in the layer below. For example, when the reflective planning agency focuses the reflective planning machine on the plan to "`find a recent deliberative plan to accomplish a goal`" and subsequently activates the reflective execution agency, the body of the plan is executed, including activating resources in the deliberative plan agency to "`focus on most recent plan`," "`focus on previous plan`," "`imagine effects of plan in focus`," and "`execute plan in focus`."

F. A column of agencies and knowledge bases abstract partial states from the deliberative plan knowledge base in the deliberative layer below. Because partial state abstraction can be a slow process, this process is performed asynchronously based on a stream of change events. A detailed description of partial states and their asynchronous abstraction will be given in chapter 4, section 4.2. Abstracted partial state event knowledge is stored in the *deliberative plan partial state event knowledge base*. The abstraction of partial states is one of two types of asynchronous processing streams that constitute the SALS AI's ability to learn from the experience of executing plans.

G. A column of agencies and knowledge bases perform asynchronous learning of abstract causal rule hypotheses from deliberative plan agency resource execution preconditions. The advantage of an asynchronous learning algorithm is that it does not slow down the execution of plans in the reflective layer. Historical versions of the deliberative plan knowledge base are reconstructed so that the slower learning algorithms in the reflective layer can discover relevant patterns in this data for predicting the effects of deliberative actions. For example, when the reflective execution agency executes the plan to "`find a recent deliberative plan to accomplish a goal`," the SALS AI learns that when "`a planner has the goal for a cube to be on a pyramid`" and "`a planner has the goal for a pyramid to be on a cube`," the resulting state will be "`a plan has an expectation failure`." The details of the asynchronous learning of abstract causal rules from

the experience of executing plans will be described in chapter 4, section 4.3.

2.5.1 *The Reflective Plan Knowledge Base*

A simplified graph representation of the reflective plan knowledge base is shown in Figure 11. In this figure, knowledge labels A and B refer to the following different types of reflective knowledge:

A. The state of the *reflective planning machine* includes positive and negative reflective goals as well as references to reflective plans for deliberative action. For example, in the story presented in chapter 1, the reflective planning machine has the negative goal for avoiding "`a planner to be focused on a plan that has failed`." The reflective planning machine fails to avoid this deliberative partial state when it executes a plan to "`execute a recent plan to accomplish a goal`," which leads to a failure while executing the deliberative plan to "`stack a cube on a pyramid`."

B. Representations of reflective plans are organized into a linked-list structure that goes forward and backward in time. Plans that have been told to the SALS AI furthest in the past are at the beginning of the list. In the example story, the SALS AI uses this linked-list structure to organize its search through reflective plans for deliberative action. Initially, the SALS AI executes the reflective plan to "`execute a recent plan to accomplish a goal`," which results in a search through deliberative plans for physical action starting with most recently learned deliberative plans. This search method leads to a failure to accomplish a deliberative goal, one of the physical partial states for "`a block to be on a block`." The failure of the deliberative plan to "`stack a cube on a pyramid`" subsequently causes the failure of the reflective plan to avoid the reflective goal of avoiding "`a deliberative planner to be focused on a plan that has failed`." At this point in the example, the SALS AI reflectively learns to apply a different reflective plan given the state of the deliberative plan knowledge base, including the current deliberative goals. The reflective plan to "`execute an old plan to accomplish a goal`" initiates a search through deliberative plans from oldest to newest, which results in accomplishing a positive deliberative goal and avoiding the negative reflective goal. In this way, the SALS AI learns to apply different search strategies, or planning methods, for successfully accomplishing different types

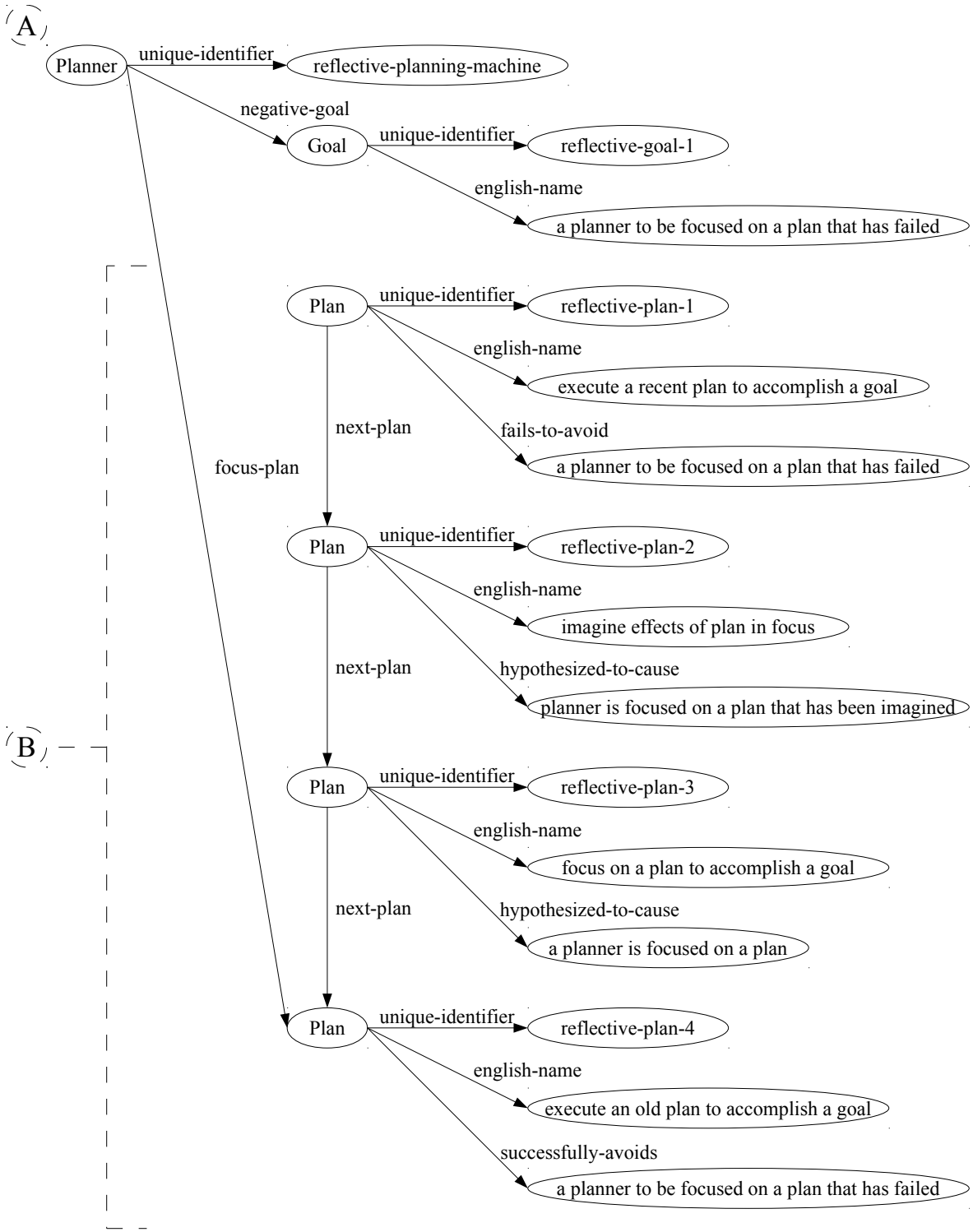


Figure 11: Part of the reflective plan knowledge base represented as a graph. See text in subsection 2.5.1 on page 55 for descriptions of knowledge labels, A and B.

of goals, given feedback from the experience of actually executing the plans that it has found.

Although the reflective layer of the SALS cognitive architecture is superficially similar to the deliberative layer, the type of knowledge that the reflective layer reasons about is categorically different. While the deliberative layer plans toward relatively simple physical goals, such as "`a cube to be on a pyramid`," the partial states in the reflective layer are the much more complex partial states of the deliberative layer below, such as "`a planner to be focusing on a plan that is hypothesized to cause a cube to be on a pyramid`." Because of the categorically different types of knowledge in different planning layers, each planning layer has a separate communication path for being told natural language plans. In general, natural language plans in the deliberative layer are about controlling the physical robot arm, while natural language plans in the reflective layer are about controlling the deliberative planner.

2.6 THE SUPER-REFLECTIVE LAYER

The super-reflective layer is the third planning layer in the SALS cognitive architecture, after the deliberative and reflective layers. The learning examples in this dissertation are focused on two layers of learning: (1) in the deliberative layer about the physical effects of physical actions, and (2) in the reflective layer about the deliberative effects of deliberative actions. To simplify the logistics of implementing the reflective planning process in the SALS AI, a super-reflective planning layer is included that contains natural language plans that when executed become the reflective planning process. The super-reflective plans are very similar to reflective plans because both of these layers control planning layers below: the reflective layer controls the deliberative planning layer, while the super-reflective layer controls the reflective planning layer. For example, in the story presented in chapter 1, the super-reflective layer is initially executing a natural language plan to "`execute a recent reflective plan to avoid all negative reflective goals`." An analogous plan exists in the reflective layer for finding a deliberative plan that avoids all negative deliberative goals, partial states of the physical knowledge base. This type of plan search can be used in general for any reflective planning layer that is controlling a planning layer below. The negative reflective goal that is being avoided in the example story is for "`a deliberative planner to be focused on a plan that has failed`." In this way, the relationship between the super-reflective layer and the reflective layer is analogous to the relationship between the reflective

layer and the deliberative layer. Although the example story does not include descriptions of super-reflective learning from experience, the super-reflective layer in the SALS AI is a completely functioning planning layer and learning from experience is implemented and does occur in this layer as well.

I have now completed my description of the Emotion Machine cognitive architecture included in the SALS AI. I have described how the bottom four layers of the Emotion Machine theory of mind have been implemented in terms of the example story presented in chapter 1. These four layers of the Emotion Machine that have been described are:

1. *Built-In Reactive Layer*
2. *Learned Reactive Layer*
3. *Deliberative Layer*
4. *Reflective Layer*

I have also described a fifth layer that has been implemented in the SALS AI: the *Super-Reflective Layer*. I see additions of super-reflective layers as a means to implementing the *Self-Reflective Layer* and the *Self-Conscious Layer* that exist as layers five and six of the Emotion Machine theory of mind, which I will describe briefly as future research in chapter 8, section 8.4 and section 8.5. In the next three chapters I will describe the remaining three contributions of this thesis:

- Chapter 3: Learning from Being Told Natural Language Plans
- Chapter 4: Learning Asynchronously from Experience
- Chapter 5: Virtual Machine and Programming Language

3

LEARNING FROM BEING TOLD NATURAL LANGUAGE PLANS

Every planning layer in the SALS cognitive architecture, including the deliberative, reflective and super-reflective layers, is capable of learning in two different ways: (1) from "being told" and (2) from "experience." Learning from being told occurs in terms of natural language plans that are programmed into the different layers of the AI by the user or potentially another AI. Figure 12 shows the information pathways in SALS that are involved in learning from being told as well as learning from experience. When a layer of the AI learns from being told, a natural language plan is communicated to that layer from a source external to the AI, such as a human user. When a layer of the AI learns from experience, two streams of trace events are received from the layer below that are asynchronously woven into hypothetical causal models of the effects of actions. In this chapter, I will focus on learning from being told natural language plans. I will describe the details of how each planning layer in SALS asynchronously learns from experience in chapter 4. The important point that I will elaborate upon in this chapter is that the SALS AI interprets natural language plans, while simultaneously considering syntax, semantics, current environmental context, learned hypothetical knowledge about the effects of actions as well as the current positive and negative goals of the AI. My approach is as opposed to linguistic traditions that focus on only one or two of these aspects of natural language understanding.

Natural language plans in SALS are in the form of a programming language with variables, conditionals, recursion, ambiguous values, an imaginative compiler, and the ability to use analogical patterns between collections of these plans to interpret natural language sentences and phrases. SALS natural language plans are sequences of commands that can be created, mutated, and executed by a planning layer to accomplish goals. The following is an example of a definition of one of the deliberative plans that the AI in the story could consider executing:

```
[defplan 'move left'
  [call-below 'move left']]
```

This expression defines a new deliberative plan. The "`defplan`" command is shorthand for "define plan." The first argument to the "`defplan`"

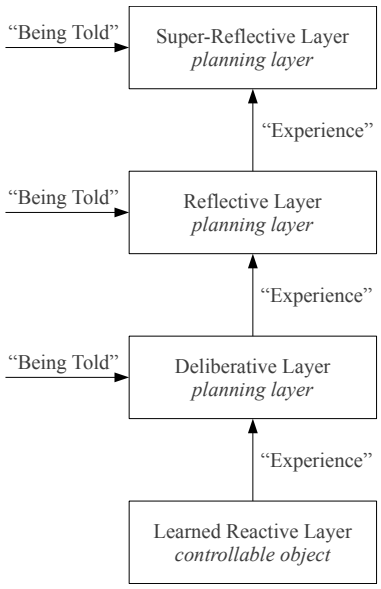


Figure 12: Learning from being told and learning from experience both occur in each SALS planning layer.

expression is the name of the plan: "`move left`." The body of the plan is the remaining sequence of expressions. The only expression in the body of this plan is the "`call-below`" expression with the "`move left`" argument. This expression activates a resource in the layer below, in this case, the "`move left`" resource, which is in the built-in reactive layer of the AI. The "`call-below`" expression not only activates a resource in the layer below but also waits for that resource to complete execution or fail. The "`move left`" plan defines a possible natural language interpretation for the "`move left`" phrase, stating that this phrase refers to the synchronous execution of the "`move left`" resource in the layer below.

Consider this analogous plan to the "`move left`" plan that defines an interpretation of the "`move right`" phrase:

```
[defplan 'move right'
  [call-below 'move right']]
```

The analogous similarity between the "`move left`" and "`move right`" commands can be abstracted into a new "`move left`" natural language plan that uses the following syntax:

```
[defplan 'move left'
  :matches ['move [? direction]']
   :frame [[direction 'left']]
    [call-below 'move [? direction]']]
```

This generalized form of the original "`move left`" and "`move right`" plans uses a natural language variable, "`direction`." Note that there are two optional arguments to the "`defplan`" expression in this example: (1) "`:matches`" and (2) "`:frame`." The optional "`:matches`" argument specifies a list of potential natural language patterns that this plan may match as it is being interpreted. In this case, the variable expression "`[? direction]`" is allowed to replace the word "`left`" from the original name of the plan. The optional "`:frame`" argument specifies the default natural language variable bindings. In this case, the "`direction`" variable is assigned the natural language phrase "`left`" by default. In the body of the generalized form of the plan, all occurrences of "`left`" have been replaced with the variable expression "`[? direction]`". Given this generalized form of the original plan, the planner can create a new analogous plan as an interpretation of either of the natural language phrases: "`move left`" or "`move right`."

3.1 CONDITIONALS AND PARTIAL STATES

The SALS natural planning language includes conditional branches that can change the course of plan execution based on the existence of partial states in the knowledge base that it is trying to control. For example, here is a more complicated SALS plan that shows a number of new SALS primitives that will be discussed next:

```
[defplan 'move toward a cube'
  [if [exists [relationship block property shape cube
                            preposition left-of
                            gripper property is-me true]]
      [call-below 'move left']
    [if [exists [relationship block property shape cube
                              preposition right-of
                              gripper property is-me true]]
        [call-below 'move right']
      [fail]]]]
```

This plan checks to see if there is a cube to the left of the gripper that the AI is controlling. If there is a cube to the left, this plan will activate the "`move left`" resource in the layer below. If there is not a cube to the left, this plan then checks to see if there is a cube to the right of the gripper that the AI is controlling. If there is a cube to the right, this plan will activate the "`move right`" resource in the layer below. At this point, if there is not a cube to the left or to the right, the plan fails. There are a number of new primitives that are introduced in this example of conditional branches:

- "`if`"
- "`exists`"
- "`relationship`"
- "`fail`"

The syntax for the SALS "`if`" expression is similar to the "`if`" expression in most lisp-like languages: the first argument is the conditional, the second argument is the true branch, and the remaining arguments are the optional body of the false branch. Unlike most lisp-like languages, the SALS "`if`" expression's conditional value must be a Boolean type object and will fail with any other value. The "`fail`" expression is a simple way for a plan to stop executing and mark the plan with the knowledge of a failure object.

The "exists" expression accepts a partial state as its only argument and checks to see if this partial state exists in the knowledge base that the planning layer is trying to control. When the effects of a plan are being imagined, the return value of the "exists" expression are not known, so multiple possible ambiguous values are returned: (1) the result based on current hypothetical models of the effects of previous actions or the current state of the knowledge base that this planning layer is trying to control if no previous actions have been imagined, (2) a true value based on the possibility that learned models are incorrect, and (3) a false value based on the possibility that learned models are incorrect. The "relationship" expression is one of two special expressions in SALS that return partial state objects. The "relationship" expression accepts ten arguments, which map directly to the internal semantic graph representation of the knowledge base that the planning layer is trying to control. The following are the ten arguments to the "relationship" expression:

1. source-type
2. source-key-type
3. source-key
4. source-value
5. key-type
6. key
7. target-type
8. target-key-type
9. target-key
10. target-value

Figure 13 shows how the arguments to the "relationship" expression map to the frame types, slot values, and properties of a frame-based knowledge base that is represented as a graph. When an argument to the "relationship" expression is a symbol, this symbol is checked against a list of known symbols within the SALS AI. If an argument to the "relationship" expression is not a known symbol, this results in an interpretation failure, limiting the number of possible interpretations.

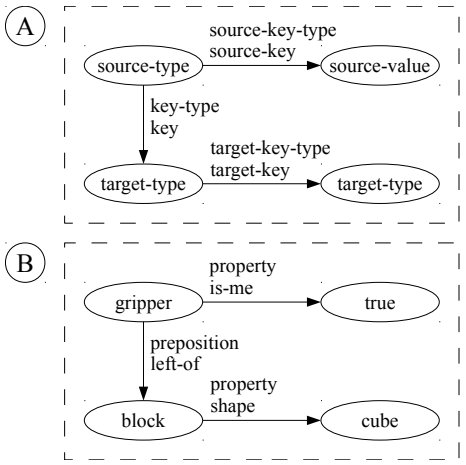


Figure 13: The SALS "`relationship`" expression returns a partial state object. The (A) top graph shows the ten argument names for the "`relationship`" expression, while the (B) bottom graph shows a potential partial state of the physical knowledge base that literally means, "`a cube shaped block to be to the left of a gripper that is me`."

Now, let us consider a slightly different type of partial state expression in the following example plan that attempts to control the gripper to grab a block:

```
[defplan 'attempt to grab block'
  [call-below 'grab']
  [wait-for [property gripper property is-me true
                    property movement-command
                    stop]]]
```

In this plan, two new types of SALS expressions are introduced:

- "`wait-for`"
- "`property`"

The "`wait-for`" expression takes one argument, which similarly to the "`exists`" expression, must be a partial state object, such as that returned by the "`relationship`" expression. Functionally, the "`wait-for`" expression puts the plan to sleep until the specified partial state exists in the knowledge base in the layer below that this plan is trying to control. The "`property`" expression is similar to the "`relationship`" expression in that it does return a partial state object, but the "`property`" expression only takes the following seven arguments:

1. `source-type`

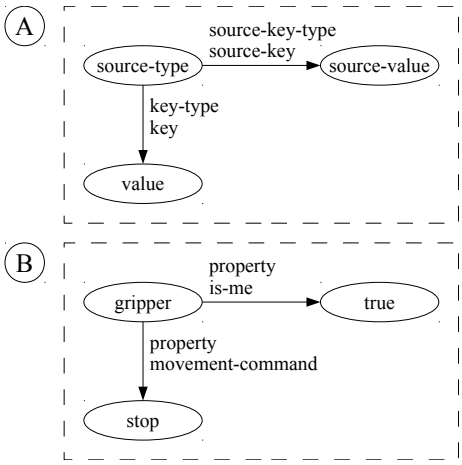


Figure 14: The SALS "`property`" expression returns a partial state object. The (A) top graph shows the seven argument names for the "`property`" expression, while the (B) bottom graph shows a potential partial state of the physical knowledge base that literally means, "`a gripper to be me and have a stop movement command`."

2. `source-key-type`
3. `source-key`
4. `source-value`
5. `key-type`
6. `key`
7. `value`

Figure 14 shows how the above arguments to the "`property`" expression map to the frame types, slot values, and properties of a frame-based knowledge base that is represented as a graph.

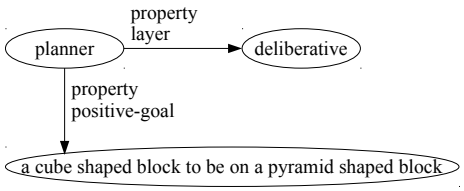


Figure 15: The SALS "relationship" and "property" expressions can be hierarchically combined in planning layers above the deliberative. Note that any partial states that are sub-expressions of other partial states become symbolically reified to maintain a frame-based graph structure for all knowledge.

While the deliberative layer may create plans that refer to partial states in the physical knowledge base, the reflective layer may create plans that refer to partial states in the deliberative plan knowledge base, which may in turn also refer to partial states in the physical knowledge base. Consider the following example of a reflective plan that includes this form of hierarchical partial state reference:

```
[defplan 'a deliberative planner to have a positive goal
         for a cube to be on a pyramid'
  [property planner property layer deliberative
            property positive-goal
            [relationship block property shape cube
                          preposition on
                          block property shape pyramid]]]
```

In this reflective plan, the "property" expression describes a partial state of the deliberative plan knowledge base, while the "relationship" expression describes a partial state of the physical knowledge base. For SALS to convert this expression to a purely deliberative form of knowledge, hierarchical partial states are reified in SALS so that they become a simple partial state that is purely of one layer's type of knowledge. When a partial state object is passed as an argument to another partial state object, the first partial state is converted to a symbolic form, so that the entire structure can continue to exist as a simple frame-based graph structure. Figure 15 shows an example of a hierarchical embedding of "relationship" and "property" partial states that may occur in any planning layer above the deliberative layer.

3.2 PLANS INTERPRETING PLANS

The most powerful capability of the SALS natural language programming language is the ability to find correct interpretations of ambiguous natural language plans. Let us first define the following simple natural language plan that returns a "`relationship`" partial state object:

```
[defplan 'a cube to be to my left'
  [relationship block property shape cube
                preposition left-of
                gripper property is-me true]]
```

This plan can be generalized to analogously work for any type of shape as in the following example:

```
[defplan 'a cube to be to my left'
  :matches ['a [? shape] to be to my left']
   :frame [[shape 'cube']]
    [relationship block property shape [? shape]
                  preposition left-of
                  gripper property is-me true]]
```

Now, consider the following plan that makes use of this previous plan definition and introduces two new SALS expression types for evaluating natural language plans:

```
[defplan 'a cube is to my left'
  [exists [plan-call [plan 'a cube to be to my left']]]]
```

This last plan returns a true or false value depending on whether or not the partial state returned by the plan, "`a cube to be to my left`," exists in the knowledge base that the planning layer is trying to control. Two new types of SALS natural language programming expressions are introduced in the last plan:

1. "`plan-call`"
2. "`plan`"

The "`plan`" expression can be thought of as taking one argument, a natural language phrase, and creating and returning one plan object that is analogous to a plan in the current planning layer. The returned plan is a procedural interpretation of the natural language expression. While this is the correct run-time functionality of the "`plan`" expression, the plan compiling process can often completely eliminate most plan expressions. Further, the "`plan`" expression is further complicated because of the

common occurrence that multiple analogous plans may match and be returned from the "`plan`" expression. In most cases, there are multiple possible matches for any given natural language phrase. In these cases, the SALS natural language plan compiler is responsible for imagining the effects of different interpretations on the knowledge base that the planning layer is trying to control, while avoiding natural language plan interpretation failures. The compiling stage in the SALS AI is responsible for removing all ambiguity from natural language phrases, so that there is only one procedural interpretation once a plan is in its final compiled form, ready to be executed. During the compiling stage, the "`plan`" expression returns an *ambiguous value* object. An ambiguous value object contains multiple possible values that each depend on different analogous natural language plan matches. To handle ambiguous value objects, each expression in the SALS natural planning language must separately consider each combination of ambiguous values that are used as arguments to those expressions. For example, the "`plan-call`" expression often takes as its first argument an ambiguous value object returned by a "`plan`" expression and must consider each of the possible ambiguous values during the compiling stage.[1]

The "`plan-call`" expression accepts one argument, a plan object, such as that returned by a previous "`plan`" expression. The purpose of the "`plan-call`" expression is to compile a plan object into the current location in the current plan that is being defined by the "`defplan`" expression. The "`plan-call`" expression is similar to a macro expansion operator in Lisp, which allows compiling plans into its own current location in another plan.[2] When the first argument is an ambiguous value object, the "`plan-call`" expression must choose one of the set of ambiguous values to compile into the current location. Plans are eliminated by simulating, or "imagining," their execution and eliminating those plans that encounter failures that can be detected at compile-time, such as type mismatch failures. If multiple possible plans still exist after this elimination process, the first successfully simulated plan is chosen.

1 The simulation of each of multiple ambiguous values during the compiling stage for interpreting natural language plans is performed in parallel on concurrent hardware by the SALS AI.

2 If the plan object is not knowable at compile-time, the SALS AI will delay this macro expansion process until run-time, but all plans in the example story presented in chapter 1 have been implemented to allow compile-time macro expansion of the "`plan-call`" expressions.

Now, consider the following redefinition of the previous "`move toward a cube`" plan that I defined previously:

```
[defplan 'move toward a cube'
  [if [plan-call [plan 'a cube is to my left']]
      [call-below 'move left']
    [if [plan-call [plan 'a cube is to my right']]
        [call-below 'move right']
      [fail]]]]
```

This version of the "`move toward a cube`" natural language plan is simpler because it only indirectly references the "`relationship`" and "`exists`" expressions through the "`plan`" and "`plan-call`" expressions that refer to the appropriate analogies to other natural language plans. Now, consider the following expression that defines an analogy for any natural language plans that use the "`if`" expression:

```
[defplan 'if a cube is to my left, move left,
          otherwise move right'
  :matches ['if [? condition], [? true-branch],
             otherwise [? false-branch]']
   :frame [[condition    'a cube is to my left']
           [true-branch  'move left']
           [false-branch 'move right']]
    [if [plan-call [plan [? condition]]]
        [plan-call [plan [? true-branch]]]
      [plan-call [plan [? false-branch]]]]]
```

Using this definition of an analogy for the "`if`" expression, the original "`move toward a cube`" natural language plan can be rewritten as follows:

```
[defplan 'move toward a cube'
  [plan-call [plan 'if a cube is to my left, move
                    left, otherwise if a cube is
                    to my right, move right,
                    otherwise fail']]]
```

Note that this last plan uses two "`if`" statements, the second is in the false branch of the first.

Before getting to the details of how ambiguity is searched through and eliminated in the SALS natural language plan compiler, consider the following definitions that include the SALS "not" expression:

```
[defplan 'a cube is not to my left'
  [not [plan-call [plan 'a cube is to my left']]]]
```

The SALS "not" expression is similar to the "not" expression in most lisp-like programming languages in that it takes one argument. Unlike most lisp-like languages, the SALS "not" expression only accepts a Boolean type of object. The "not" expression returns a new Boolean type object that represents the opposite value of the argument. The following is an analogous plan that can be used to generalize this natural language usage of the "not" expression:

```
[defplan 'a cube is not to my left'
  :matches ['[? subject] is not [? preposition]']
   :frame [[subject     'a cube']
           [preposition 'to my left']]
    [not [plan-call [plan '[? subject] is
                           [? preposition]']]]]
```

This plan allows many negative expressions to analogously have a correct interpretation, such as "`if a pyramid is not to my right, move left, otherwise fail.`"

3.3 PLANS WITH RECURSIVE INTERPRETATIONS

Another powerful component of the SALS natural programming language is the ability to compile natural language plans that include recursive references, which enable plans to describe looping functionality. SALS also has a primitive capability to imagine the possible effects of loops by imaginatively unrolling the loop only once. Plan objects have a slot-value Boolean property that keeps track of whether or not the plan "has been imagined." This property is false when a plan is first learned or created and becomes true once the effects of a plan are imagined in the current context, given the currect goals. A planning process must search through possible plans to find a plan whose effects accomplish positive goals and avoid negative goals. Part of this planning process must find plans whose effects have not yet been imagined. The following example is a definition of a reflective natural language plan that searches for a deliberative plan whose effects have not yet been imagined:

```
[defplan 'find next unimagined plan'
  [call-below 'focus on next object']
  [plan-call [plan 'if a planner is focusing on
                    a plan that has been imagined,
                    find next unimagined plan']]]
```

Notice that the name of this plan is "`find next unimagined plan`," which is the same as the true branch of the natural language "`if`" statement in the body of the plan. This plan checks to see if the plan currently in the focus of the deliberative planner has been imagined. If the plan in deliberative focus has been imagined, this plan calls itself recursively until the deliberative planner is focusing on a plan that has not been imagined.

3.4 PLANS REIFY HIERARCHICAL PARTIAL STATES

As a final example of a natural language plan interpreting a natural language plan, consider again the following hierarchical partial state construction from a reflective natural language plan from earlier in this chapter in Figure 15 on page 66:

```
[defplan 'a deliberative planner to have a positive goal
         for a cube to be on a pyramid'
  [property planner property layer deliberative
            property positive-goal
            [relationship block property shape cube
                          preposition on
                          block property shape pyramid]]]
```

This reflective natural language plan can be abstracted with "`plan`" and "`plan-call`" expressions as in the following example:

```
[defplan 'a deliberative planner to have a positive goal
         for a cube to be on a pyramid'
  :matches ['a deliberative planner to have a positive
             goal for [? partial-state]']
   :frame [[partial-state 'a cube to be on a pyramid']]
    [property planner property layer deliberative
              property positive-goal
              [plan-call [plan [? partial-state]]]]]
```

In this way, analogous reflective plans can be created to allow the interpretation of any natural language physical partial state being a positive goal of the deliberative planning layer. While the above plan provides interpretations for partial states that involve a planner pursuing a positive goal, the same technique can be used to create reflective natural language plans that provide interpretations of partial states that include a planner that is avoiding negative goals.

3.5 ANALOGOUS PLAN INTERPRETATION

As previously described, the "`plan`" expression in a SALS natural language plan returns a plan object that either matches the name of a plan previously defined via the "`defplan`" expression, or if an analogy can be found to an existing plan, a new analogous plan object is created and returned as the result of the "`plan`" expression. Because of the possibility that multiple plans may match a given "`plan`" expression, it

is the task of the SALS compiler to imagine the effects of the different interpretations and decide upon one for execution. The SALS natural language plan compiler must handle multiple ambiguous return values for each "`plan`" expression. Let us consider again the following natural language plan that must be imagined and interpreted, which requires the compiler to sort through multiple possible interpretations:

```
[plan-call [plan 'if a cube is not on a pyramid, stack a
                 cube on a pyramid']]
```

A reasonable way to expect this natural language phrase to be interpreted is as a plan analogous to the natural language plan for the "`if`" expression similar to the one previously defined, as in the following:

```
[defplan 'if a cube is not on a pyramid, stack a cube on a
          pyramid'
  :matches ['if [? condition], [? true-branch]']
   :frame [[condition    'a cube is not on a pyramid']
           [true-branch  'stack a cube on a pyramid']]
    [if [plan-call [plan [? condition]]]
        [plan-call [plan [? true-branch]]]]]
```

Although this plan makes sense, there are many other possible problematic analogies to previously defined natural language plans that do not make any sense at all. The following problematic interpretation is one example:

```
[defplan 'if a cube is not on a pyramid, stack a cube on a
          pyramid'
  :matches ['[? subject] is not [? preposition]']
   :frame [[subject     'if a cube']
           [preposition 'on a pyramid, stack a cube on a
                         pyramid']]
    [not [plan-call [plan '[? subject] is
                          [? preposition]']]]]
```

Notice that the natural language value of the "`subject`" variable in the previous problematic interpretation is equal to "`if a cube`." The following is the result of one step in compiling this problematic interpretation:

```
[not [plan-call [plan 'if a cube is on a pyramid, stack a
                      cube on a pyramid']]]
```

Notice that the "`not`" expression has been moved to the front of this expression after it has been partially compiled. The following is the result of another couple steps of further interpreting the remaining natural language in this expression:

```
[not [if [exists [plan-call [plan 'a cube to be on a
                                    pyramid']]]
         [plan-call [plan 'stack a cube on a pyramid']]]]
```

When this problematic interpretation is imagined, the result from the "`exists`" expression is a Boolean value, so this satisfies the "`if`" conditional type requirement. However, the result of the "`if`" expression is the special type "`nil`," which is not a Boolean value in the SALS natural programming language. Because the "`not`" expression strictly requires a Boolean type value, the imaginative interpretation of this plan fails when the nil value from the "`if`" statement reaches the "`not`" expression. Using strict typing in the low-level details of the SALS natural programming language allows ambiguous high-level expressions with many possible interpretations to be narrowed down to a few that make programmatic sense. By introducing more constraints to the low-level details of the SALS natural programming language, many types of plan failures can be imagined and avoided, even while using this very loose type of natural language analogical matching technique. Not all errors can be imagined and avoided through imaginative compiling, but many types of failures are avoided in this way. However, some failures, such as expectation failures, can only be realized during the actual execution of the plan.

3.6 IMAGINING THE EFFECTS OF AMBIGUOUS PLANS

As natural language plans are interpreted, the hypothetical effects of any resource activations in the layer below are also simultaneously imagined in the counterfactual knowledge base of that planning layer. Imagining the effects of resource activations first requires that hypothetical models of the effects of resource activations exist. Hypothetical models of the effects of resource activations in SALS provide a rule-based mapping from the preconditions of the resource activation to the potential transframes for the resource activation. The preconditions and transframes are in terms of the abstract partial states that have previously existed in the knowledge base that the planning layer is trying to control. For example, "`a gripper that is me being above a cube shaped block`" could be one of many preconditions for an action. All partial states that can be returned by the "`relationship`" and "`property`" expressions in the SALS natural programming language are efficiently abstracted asynchronously from the knowledge base that the planning layer is trying to control. I will describe the details of the abstract asynchronous learning algorithm for each planning layer in chapter 4. For now, know

that abstract hypothetical models of resource activations are learned and can be used for imagining the effects of resource activations during the natural language plan interpretation process.

Because some expressions in the SALS natural planning language can return multiple possible ambiguous values, one task of the planning layer is to decide which of the multiple possible interpretations is complete, accomplishes positive goals and avoids negative goals. This means that each expression in the SALS natural programming language may have one or more possible different outcomes, depending on the interpretation path and which one of potentially multiple ambiguous values is chosen to be executed when a decision must be made. In order to keep track of each of these different interpretations, each plan expression is allocated an *execution node* object and each decision among multiple ambiguous argument values is allocated an *argument decision node* object in the plan knowledge base of that planning layer. The execution node objects correspond to the functional hierarchy of the imagined natural language plan execution, while the argument decision node objects represent any points in the imagined execution where two potentially different sub-trees of the functional execution could occur based on different choices between multiple possible ambiguous return values from sub-expressions of the current expression being imaginatively executed. Figure 16 shows a graph representation of the frame-based deliberative plan knowledge base as it is in the process of imaginatively evaluating the following simple plan:

```
[defplan 'a cube is not on a pyramid'
  :matches ['[? subject] is not [? preposition]']
   :frame [[subject     'a cube']
           [preposition 'on a pyramid']]
    [not [plan-call [plan '[? subject] is
                          [? preposition]']]]]
```

When this natural language plan is imaginatively interpreted, there are multiple possible values returned from the "`plan`" expression, which returns analogous plans that match the natural language phrase, "`a cube is on a pyramid`." In this case, there are the two following plans that are created as analogies to other known plans:

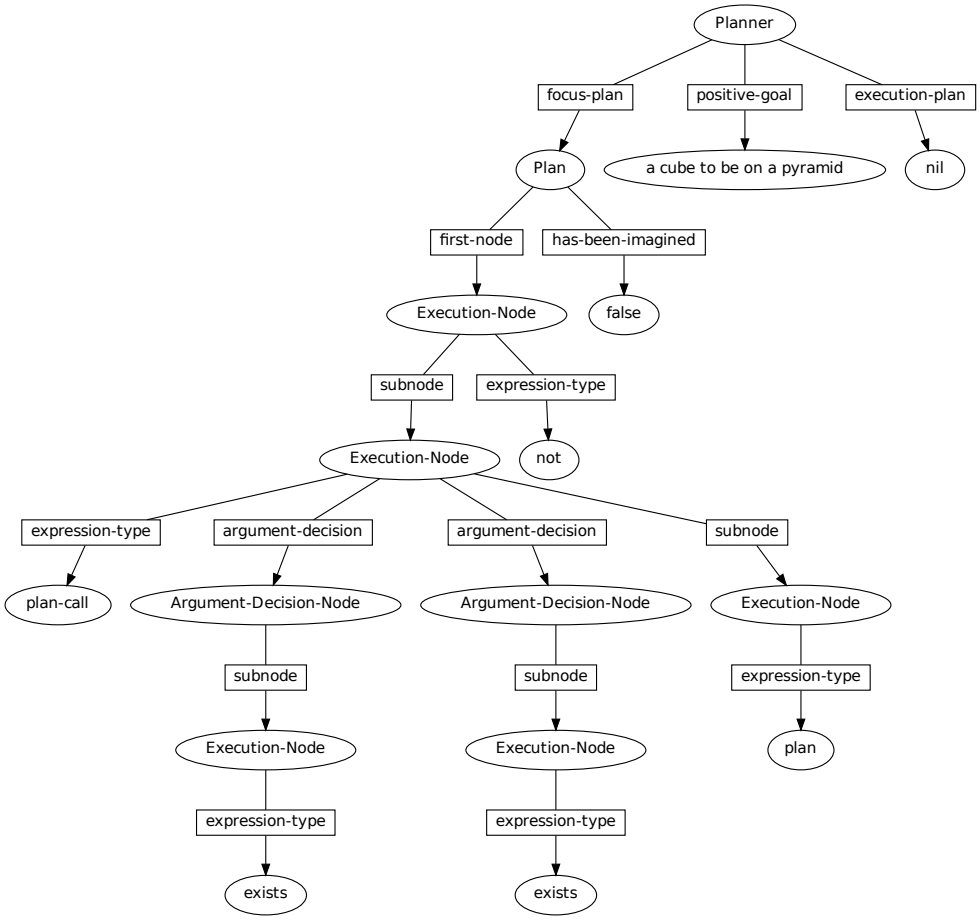


Figure 16: A graph representation of the deliberative plan knowledge base where nodes are represented by ellipses and edge labels are represented by rectangles. A simple plan with multiple ambiguous interpretations is being imaginatively interpreted and evaluated. The deliberative *planner* object is focusing on a *plan* object that has been partially evaluated. The first execution-node object of this plan object represents the partial interpretation of a "`not`" expression that has a "`plan-call`" sub-expression. The "`plan`" expression returns two possible values, which are interpreted separately under "`argument-decision-node`" expressions.

1.
```
[defplan 'a pyramid is on a cube'
  :matches ['a [? top-shape] is on a [? bottom-shape]']
   :frame [[top-shape    'pyramid']
           [bottom-shape 'cube']]
    [exists [relationship block property shape
                            [? top-shape]
                          preposition on
                          block property shape
                            [? bottom-shape]]]]
```

2.
```
[defplan 'a pyramid is on a cube'
  :matches ['a [? top-color] is on a [? bottom-color]']
   :frame [[top-color    'pyramid']
           [bottom-color 'cube']]
    [exists [relationship block property color
                            [? top-color]
                          preposition on
                          block property color
                            [? bottom-color]]]]
```

The first of these two analogous interpretations is what one would expect: the natural language phrases "`pyramid`" and "`cube`" are interpreted to be shapes of blocks that are on top of one another. The second of these interpretations is less obvious and incorrectly interprets "`pyramid`" and "`cube`" to be colors. This problematic interpretation is an analogy to the following plan that the SALS AI already knows:

```
[defplan 'a red is on a blue'
  :matches ['a [? top-color] is on a [? bottom-color]']
   :frame [[top-color    'red']
           [bottom-color 'blue']]
    [exists [relationship block property color
                            [? top-color]
                          preposition on
                          block property color
                            [? bottom-color]]]]
```

The SALS AI does not immediately know that "`red`" and "`blue`" are colors, while "`cube`" and "`pyramid`" are shapes. While types of partial states could be programmed into the SALS AI so that these specific words could be assigned specific symbolic types, this is not the approach taken in the SALS AI. Instead, both of these interpretations are plausible in the SALS AI. However, when the SALS AI does finish imagining all possible interpretations of a natural language phrase, each different resulting

analogous plan has a set of associated hypothetical states that this plan may or may not accomplish. If the possible effects of a plan include a "`pyramid`" color, which does not make sense, the SALS AI sees that this is not one of its goals, so it ignores this interpretation for that reason alone. The SALS AI is a goal-oriented natural language understanding system in this sense—finding those natural language plan interpretations that it thinks will accomplish its positive goals. On the other hand, the SALS AI considers its negative goals in the opposite sense: when a natural language plan interpretation is hypothesized to accomplish a negative goal, that plan interpretation is ignored. The SALS AI can be considered to be an "optimistic" or "pessimistic" natural language understanding system in these cases. The important point here is that the SALS AI interprets natural language plans, while simultaneously considering syntax, semantics, current environmental context, learned hypothetical knowledge about the effects of actions as well as the current positive and negative goals of the AI.

There are two ways that a natural language plan is hypothesized to cause a specific partial state: (1) learned hypothetical models are used to predict the effects of actions, and (2) existence checks for partial states during the imaginative interpretation of the plan are used as secondary evidence that a plan may or may not be expecting a specific partial state to exist during its execution. For example, if a natural language plan checks for the existence of the partial state, "`a cube shaped block to be on top of a pyramid shaped block`," this existence check provides secondary evidence that this plan could be expecting this state to exist during its execution. This secondary evidence of the possible intentions of a plan is an example of knowledge learned during the imaginative interpretation of natural language plans in the SALS AI.

4

LEARNING ASYNCHRONOUSLY FROM EXPERIENCE

In the previous chapter, chapter 3, I have described the details of one of two main types of learning in the SALS cognitive architecture. To briefly review, every planning layer in the SALS cognitive architecture, including the deliberative, reflective and super-reflective layers, is capable of learning in two different ways: (1) from "being told" and (2) from "experience." Learning from being told occurs in terms of natural language plans that are programmed into the different planning layers of the AI by the user or potentially another AI. When plans are told to a planning layer in the AI, the planning layer reasons about different interpretations and imagined effects of the natural language plans. In Figure 12, which is reproduced in Figure 17 for convenience, shows the information pathways in SALS that are involved in learning from being told as well as learning from experience. When a layer of the AI learns from being told, a natural language plan is communicated to that layer from a source external to the AI, such as a human user. When a layer of the AI learns from experience, two streams of trace events are received from the layer below that are asynchronously woven into hypothetical causal models of the effects of actions. In this chapter, I will focus on how the SALS AI learns from the experience of actually executing the compiled natural language plans that result from the successful interpretation and imagination described in the previous chapter. Learning from experience gives the SALS AI new hypothetical models of the effects of activating resources in the layers below that a given planning layer is trying to control.

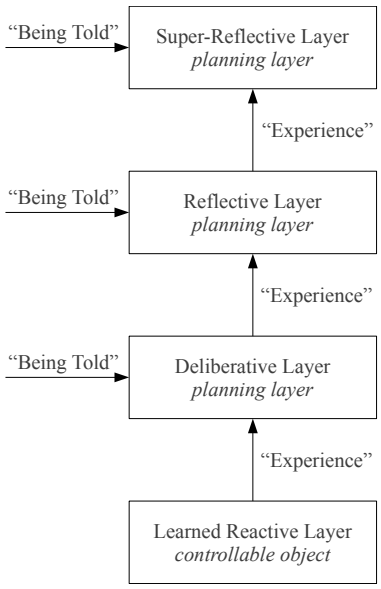


Figure 17: Learning from being told and learning from experience both occur in each SALS planning layer. This figure is reproduced from Figure 12 on page 60 of the previous chapter.

4.1 TWO EXPERIENTIAL EVENT STREAMS

Each planning layer in the SALS AI receives two event streams from the layer below that it is trying to control: (1) a stream of all changes in the knowledge base that the planning layer is trying to control and (2) a stream of all activations and completions of resource executions. I will refer to these two asynchronous information processing pathways as:

1. *partial state event reification*, and
2. *resource causal rule-learning*.

For the deliberative planning layer, these two streams of events come from (1) the changes in the physical knowledge base in the learned reactive layer and (2) the activation and completion events for the resources in the learned reactive physical agency. Analogously, for the reflective layer, these two streams of events come from (1) changes in the deliberative plan knowledge base in the deliberative layer and (2) the activation and completion events for the resources in the deliberative plan agency. The super-reflective layer follows the same pattern with the two streams of events coming from (1) the changes in the reflective plan knowledge base in the reflective layer and (2) the activation and completion events for the resources in the reflective plan agency. Figure 18 shows two experiential event streams that flow into the deliberative planning layer

from the learned reactive layer. Analogous event streams flow from the deliberative planning layer to the reflective planning layer and from the reflective planning layer to the super-reflective planning layer. Each of these two event streams are processed by separate information processing pathways each involving multiple planning layer agencies and knowledge bases before being woven into hypothetical models of the effects of resource executions of the learned reactive layer. Partial state event reification is performed in the first stage of asynchronous information processing. Changes in the physical knowledge base are streamed to the deliberative planning layer where they are reified into the physical partial state event knowledge base. Resource causal rule-learning is performed in the second stage of asynchronous information processing. Activation and completion events of resources in the learned reactive physical agency are streamed to the deliberative planning layer where they are correlated with the physical partial state event knowledge base where rule learning is used to develop hypothetical models of the effects of actions.

Figure 19 shows the two experiential event streams that flow into the reflective planning layer from the deliberative planning layer. Analogous event streams flow from the learned reactive layer to the deliberative planning layer and from the reflective planning layer to the super-reflective planning layer. Each of these two event streams are processed by separate information processing pathways each involving multiple planning layer agencies and knowledge bases before being woven into hypothetical models of the effects of resource executions of the deliberative planning layer. Partial state event reification is performed in the first stage of asynchronous information processing. Changes in the deliberative plan knowledge base are streamed to the reflective planning layer where they are reified into the deliberative plan partial state event knowledge base. Resource causal rule-learning is performed in the second stage of asynchronous information processing. Activation and completion events of resources in the deliberative planning agency are streamed to the reflective planning layer where they are correlated with the deliberative plan partial state event knowledge base where rule learning is used to develop hypothetical models of the effects of actions.

Figure 20 shows the two experiential event streams that flow into the super-reflective planning layer from the reflective planning layer. Analogous event streams flow from the learned reactive layer to the deliberative planning layer and from the deliberative planning layer to the reflective planning layer. Each of these two event streams are processed by separate information processing pathways each involving

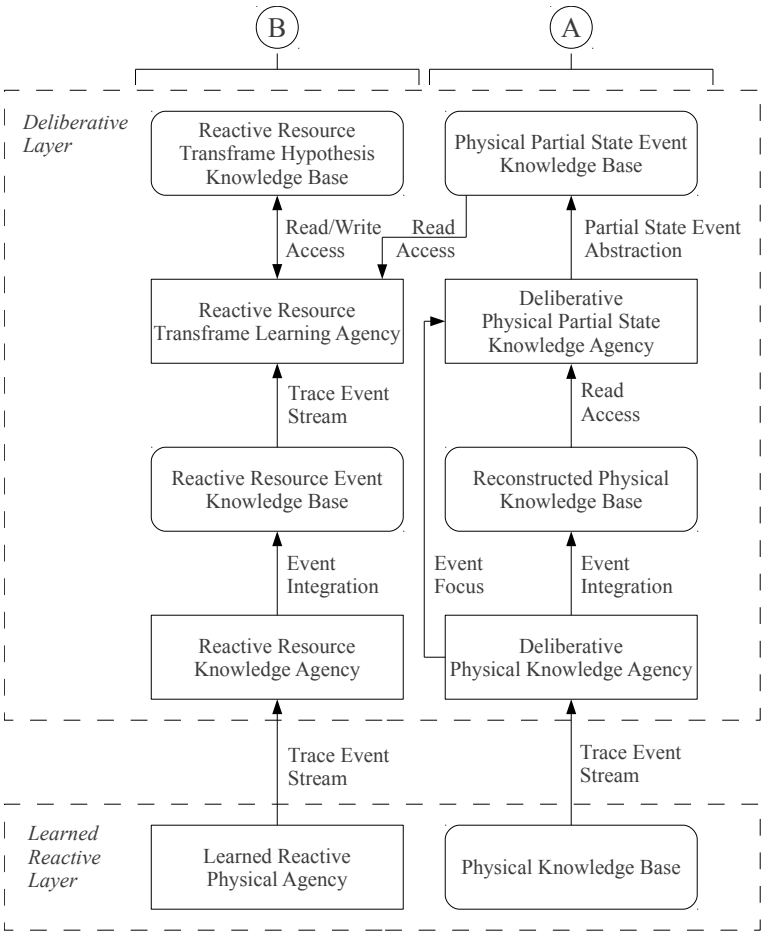


Figure 18: Two experiential event streams flow from the learned reactive layer into the deliberative planning layer in the SALS AI. Analogous event streams flow from the deliberative planning layer to the reflective planning layer and from the reflective planning layer to the super-reflective planning layer. Each of these two event streams are processed by separate information processing pathways each involving multiple planning layer agencies and knowledge bases before being woven into hypothetical models of the effects of resource executions of the learned reactive layer. (A) Partial state event reification is performed in the first stage of asynchronous information processing. Changes in the physical knowledge base are streamed to the deliberative planning layer where they are reified into the physical partial state event knowledge base. (B) Resource causal rule-learning is performed in the second stage of asynchronous information processing. Activation and completion events of resources in the learned reactive physical agency are streamed to the deliberative planning layer where they are correlated with the physical partial state event knowledge base where rule learning is used to develop hypothetical models of the effects of actions.

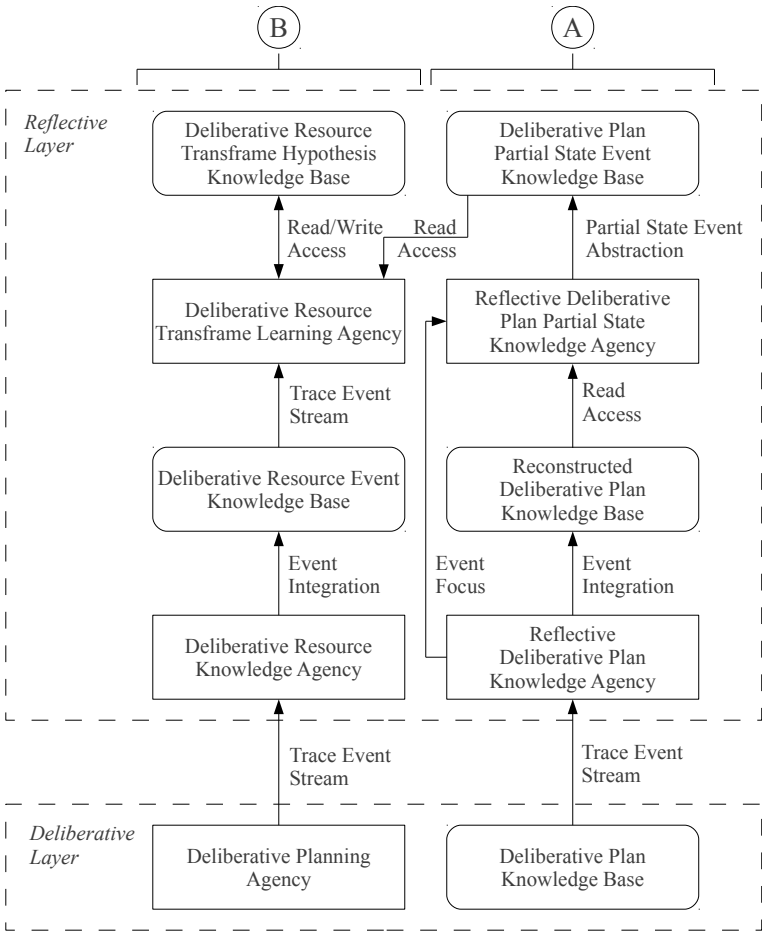


Figure 19: Two experiential event streams flow from the deliberative planning layer into the reflective planning layer in the SALS AI. Analogous event streams flow from the learned reactive layer to the deliberative planning layer and from the reflective planning layer to the super-reflective planning layer. Each of these two event streams are processed by separate information processing pathways each involving multiple planning layer agencies and knowledge bases before being woven into hypothetical models of the effects of resource executions of the deliberative planning layer. (A) Partial state event reification is performed in the first stage of asynchronous information processing. Changes in the deliberative plan knowledge base are streamed to the reflective planning layer where they are reified into the deliberative plan partial state event knowledge base. (B) Resource causal rule-learning is performed in the second stage of asynchronous information processing. Activation and completion events of resources in the deliberative planning agency are streamed to the reflective planning layer where they are correlated with the deliberative plan partial state event knowledge base where rule learning is used to develop hypothetical models of the effects of actions.

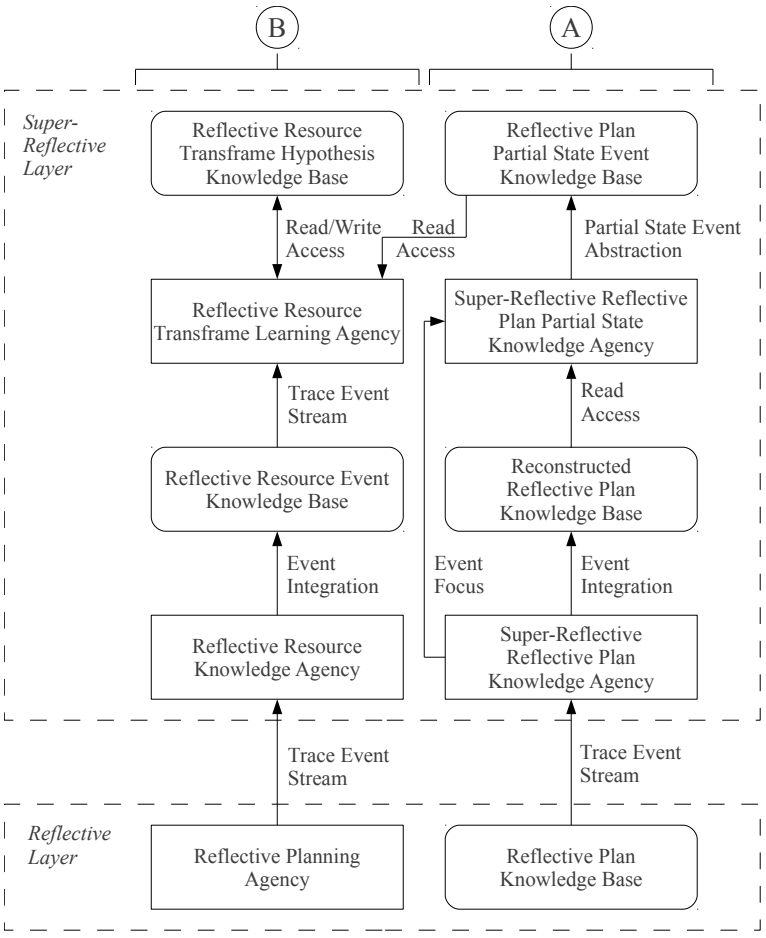


Figure 20: Two experiential event streams flow from the reflective planning layer into the super-reflective planning layer in the SALS AI. Analogous event streams flow from the learned reactive layer to the deliberative planning layer and from the deliberative planning layer to the reflective planning layer. Each of these two event streams are processed by separate information processing pathways each involving multiple planning layer agencies and knowledge bases before being woven into hypothetical models of the effects of resource executions of the reflective planning layer. (A) Partial state event reification is performed in the first stage of asynchronous information processing. Changes in the reflective plan knowledge base are streamed to the super-reflective planning layer where they are reified into the reflective plan partial state event knowledge base. (B) Resource causal rule-learning is performed in the second stage of asynchronous information processing. Activation and completion events of resources in the reflective planning agency are streamed to the super-reflective planning layer where they are correlated with the reflective plan partial state event knowledge base where rule learning is used to develop hypothetical models of the effects of actions.

multiple planning layer agencies and knowledge bases before being woven into hypothetical models of the effects of resource executions of the reflective planning layer. Partial state event reification is performed in the first stage of asynchronous information processing. Changes in the reflective plan knowledge base are streamed to the super-reflective planning layer where they are reified into the reflective plan partial state event knowledge base. Resource causal rule-learning is performed in the second stage of asynchronous information processing. Activation and completion events of resources in the reflective planning agency are streamed to the super-reflective planning layer where they are correlated with the reflective plan partial state event knowledge base where rule learning is used to develop hypothetical models of the effects of actions.

4.2 PARTIAL STATE EVENT REIFICATION

Partial state event reification is the first stage of asynchronous information processing of an experiential event stream that any given planning layer receives from the changes in the knowledge base that it is trying to control. *Reification* is the process that allows a subgraph in the layer below to be replaced by a symbol in the layer above. A change event object has the following six properties:

1. `time`
2. `change-type`
3. `source`
4. `key-type`
5. `key`
6. `target`

The "`time`" of the change event object is the clock time at which the change occurred. The "`change-type`" property of the change event object is one of two symbolic values: (1) "`add`" or (2) "`remove`." The "`source`" property of the change event object is the frame-based object in the knowledge base that has had a slot value added or removed. The "`key-type`" and "`key`" properties of the change event object are both symbolic values that together refer to the name of the slot of the frame-based object in the knowledge base that has been added or removed. The "`target`" property of the change event object is the slot value of the frame-based object in the knowledge base that has been added or removed. The "`target`" property can either be a symbolic property or another frame-based object.

Every single change that occurs within the knowledge base that a planning layer is trying to control results in a change event object being appended to a procedurally reflective event stream that flows into the planning layer above. The first agency in the planning layer that receives this event stream reconstructs a copy of the knowledge base that it is trying to control. Because an event stream can be buffered, this reconstructed knowledge base can move forward in time more slowly than the knowledge base that the planning layer is trying to control. Creating a reconstruction of the knowledge base under control is important because this allows the partial state event reification to occur asynchronously with the process that is making changes to the knowledge base under control. In practice, because sets of changes to knowledge bases often occur in bursts, the buffering of event streams usually does not lead to arbitrarily long buffer lengths. Once an event has been integrated into the knowledge base reconstruction, this change results in adding or removing a subset of partial states from the knowledge base reconstruction. This subset of partial state changes is computed and each change in this subset is integrated into the partial state event knowledge base. Computing all possible partial states would be an intractable problem, so the SALS AI does not reify all partial states that occur in the reconstructed knowledge base, but instead only focuses on two specific types of partial states: (1) "`relationship`" partial states and (2) "`property`" partial states.

The SALS "`relationship`" and "`property`" types of partial states are the same as those described in chapter 3 as the return values of the "`relationship`" and "`property`" expressions of the SALS natural programming language. To briefly review, the following are the ten arguments to the "`relationship`" expression, which define a SALS "`relationship`" partial state:

1. `source-type`
2. `source-key-type`
3. `source-key`
4. `source-value`
5. `key-type`
6. `key`
7. `target-type`
8. `target-key-type`
9. `target-key`
10. `target-value`

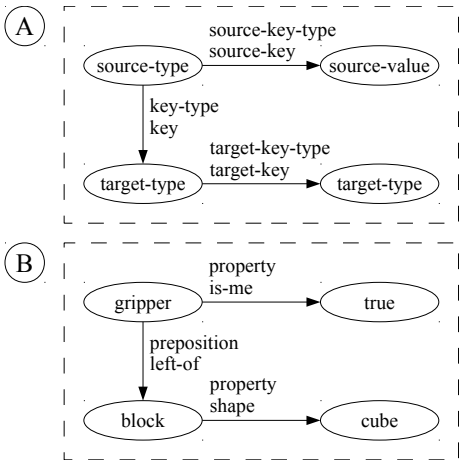


Figure 21: A SALS "relationship" partial state. This figure is reproduced from Figure 13 on page 64 for convenience. The (A) top graph shows the ten argument names for the "relationship" expression. The (B) bottom graph shows a potential partial state of the physical knowledge base that literally means, "a cube shaped block to be to the left of a gripper that is me."

Figure 21 shows how the arguments to the SALS "relationship" partial state map to the frame types, slot values, and properties of a frame-based knowledge base as it is represented as a graph. The SALS "property" partial state represents the conjunction of two different properties of a frame-based object and takes the following seven arguments:

1. source-type
2. source-key-type
3. source-key
4. source-value
5. key-type
6. key
7. value

Figure 21 shows how the arguments to the "property" expression map to the frame types, slot values, and properties of a frame-based knowledge base that is represented as a graph. While the reception of a single "add" or "remove" change event object only adds or removes a single edge from the knowledge base, this results in potentially adding or removing many "relationship" and "property" type partial states to or from that knowledge base. Figure 23 shows how a "remove" change event object is integrated into the reconstructed physical knowledge

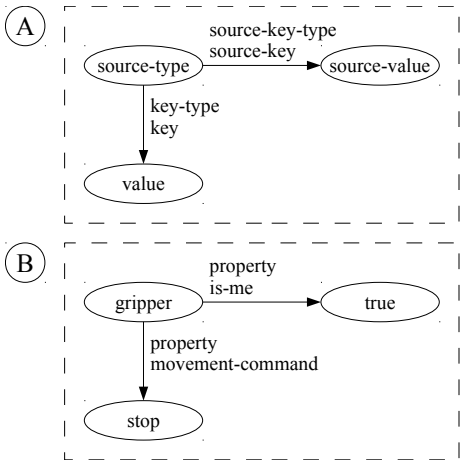


Figure 22: The SALS "`property`" partial state object. This figure is reproduced from Figure 14 on page 65 for convenience. The (A) top graph shows the seven argument names for the "`property`" expression. The (B) bottom graph shows a potential partial state of the physical knowledge base that literally means, "`a gripper to be me and have a stop movement command`."

base. Figure 24 shows how an "`add`" change event object is integrated into the reconstructed physical knowledge base.

The fact that the SALS cognitive architecture only has two types of partial state objects, the "`relationship`" and "`property`" partial state objects, is a current limitation of the SALS cognitive architecture. This means that the architecture cannot pursue goals or make judgments about other potentially more complex types of partial states in the knowledge bases that planning layers are attempting to control. For example, the "`property`" partial state object is only able to describe partial states of a knowledge base that involve two properties of a single frame-based object, while the "`relationship`" partial state object is only able to describe partial states of a knowledge base that involve one relationship between two frame-based objects with one symbolic property each. While these two types of partial state objects can sometimes be combined in clever ways to describe some more complex partial states, generally, if a goal condition requires a partial state object description of three or more frame-based objects that are related with three or more properties each, there are not currently SALS partial state objects that directly describe these more complex types of partial states. In chapter 8, I will describe an idea for how types of partial states in the SALS cognitive architecture could be reified in future research to include any specific type of subgraph in a frame-based knowledge base, but for now, the

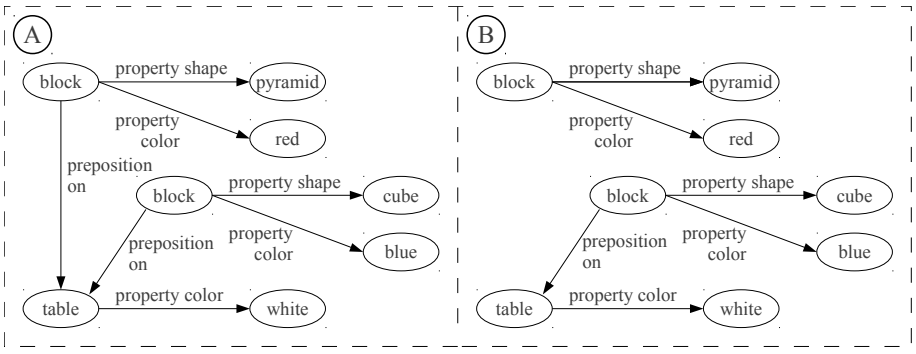


Figure 23: A "remove" change event object is integrated into the reconstructed physical knowledge base. In this case, a pyramid shaped block is no longer related by the "on" relationship to a white colored table object. While the reception of a single "remove" change event object only removes a single edge from the knowledge base, this results in potentially removing many partial states from that knowledge base. (A) The reconstructed physical knowledge base in the deliberative layer *before* a "remove" change event object is received by the deliberative physical knowledge agency. (B) The reconstructed physical knowledge base in the deliberative layer *after* a "remove" change event object is received by the deliberative physical knowledge agency.

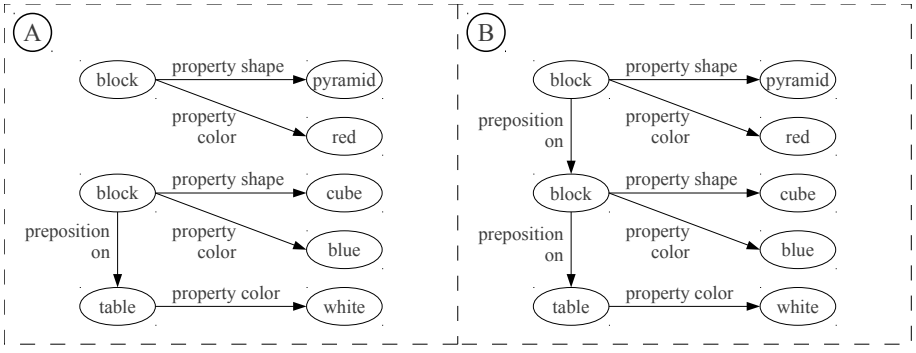


Figure 24: An "add" change event object is integrated into the reconstructed physical knowledge base. In this case, a pyramid shaped block is now related by the "on" relationship to a cube shaped block object. While the reception of a single "add" change event object only adds a single edge to the knowledge base, this results in potentially adding many partial states to that knowledge base. (A) The reconstructed physical knowledge base in the deliberative layer *before* an "add" change event object is received by the deliberative physical knowledge agency. (B) The reconstructed physical knowledge base in the deliberative layer *after* an "add" change event object is received by the deliberative physical knowledge agency.

reification of the "relationship" and "property" types in SALS are each specially implemented to efficiently reify each of these types of partial state objects from a frame-based knowledge base, avoiding the NP-complete subgraph isomorphism decision problem (Messmer 1995, Messmer & Bunke 2000) implied by a careless implementation of this more specific problem, which gains in efficiency by taking advantage of specially implemented locally directed searches for each different type of partial state object. After a change event from the physical knowledge base is received by the deliberative physical knowledge agency, the same change event is passed to the deliberative physical partial state agency to focus a local search around this change in the reconstructed physical knowledge base that searches for "property" and "relationship" partial state objects. If the change event object is a "remove" event object, the local search in the reconstructed physical knowledge base is performed *before* the reconstructed physical knowledge base is updated, while if the change event object is an "add" event object, the local search in the reconstructed physical knowledge base is performed *after* the reconstructed physical knowledge base is updated.

When a partial state object is found to be added by the deliberative physical partial state agency, a *partial state event* object is added to the physical partial state event knowledge base in the deliberative layer. A partial state event object consists of the following three properties:

1. start-time
2. end-time
3. partial-state

A partial state event object that has just been created has an absolute "start-time" value, which is a copy of the "time" slot value of the change event object from the physical knowledge base in the learned reactive layer. The "end-time" value for the partial state event object is initially unknown, so the symbol "after" is used to indicate that the partial state event object has not yet ended. The "partial-state" slot of the partial state event object is a reference to the partial state object that has been found in the reconstructed physical knowledge base in the deliberative layer. When a partial state object is found to be removed by the deliberative physical partial state agency, an absolute "end-time" slot value is added to the partial state event object that was previously added to the physical partial state event knowledge base.

Although I have described this process of partial state event reification in terms of the deliberative layer reifying partial state event objects from a stream of change event objects derived from changes in the

physical knowledge base in the learned reactive layer, an analogous process exists between each planning layer and the knowledge base in the layer below that it is trying to control. The reflective deliberative plan knowledge agency receives a stream of change event objects from the deliberative plan knowledge base in the deliberative layer, and the reflective deliberative plan partial state knowledge agency in the reflective layer reifies the partial states in the reconstructed deliberative plan knowledge base into partial state event objects that are added to the deliberative plan partial state event knowledge base in the reflective layer. Similarly, the super-reflective layer reifies partial states of the reflective plan knowledge base and adds new partial state event objects to the reflective plan partial state event knowledge base.

4.3 RESOURCE CAUSAL RULE-LEARNING

The second asynchronous stage of information processing involved in asynchronously learning from experience in the SALS AI is the *resource causal rule-learning* stage. I will first focus on how this stage works when the deliberative layer receives a stream of procedurally reflective events from the learned reactive layer with the understanding that an analogous asynchronous information process exists in each planning layer in the SALS AI. The reactive resource knowledge agency in the deliberative layer receives a stream of activation and completion event objects from the resources in the learned reactive physical agency of the learned reactive layer. When an activation event object is received by the reactive resource knowledge agency in the deliberative layer, a *resource execution event* object is added to the reactive resource event knowledge base. Like the partial state event object, a resource execution event object consists of the following three properties:

1. `start-time`
2. `end-time`
3. `resource`

When a new resource execution event object is created, its "`start-time`" slot value is copied from the "`time`" slot of the activation event object that the reactive resource knowledge agency in the deliberative layer has received from the learned reactive physical agency in the learned reactive layer. When a completion event object is received from the learned reactive layer, the "`end-time`" slot value of the appropriate resource execution event object is copied from the "`time`" slot of the completion event object. Transframes (Minsky 1975) represent change. When a

resource execution event has completed, a transframe is created that keeps track of the added or removed information in the physical partial state event knowledge base between the "`start-time`" and "`end-time`" slot values of the resource execution event. Creating this transframe requires retrieving all of the partial state events that intersect with the "`start-time`" slot value as well as all of the partial state events that intersect with the "`end-time`" slot value.

Every knowledge base that contains event objects in the SALS AI is organized for indexing by time by using an interval tree that stores all events that have been added to that knowledge base. These types of knowledge bases in the SALS AI are called *event knowledge bases*, and they have a $O(\log(n))$ time complexity for retrieving events by time, given that n is the number of events in the event knowledge base. To maintain consistency within the interval tree within the event knowledge base, some precautions must be taken when changing the values of the "`start-time`" and "`end-time`" slots of any event object, which could potentially invalidate the entire interval tree. To make this bookkeeping transparent to the plan execution, reflectively traced callbacks are called whenever a slot value is added to or removed from the event knowledge base. If the slot value refers to the "`start-time`" or the "`end-time`" of an event object, the event object is first removed from the interval tree with its old slot value, the slot value is changed, then the event object is inserted into the interval tree once the slot value change is complete. This means that the plan simply executes normally, which reflectively causes the addition and removal of event objects to the reconstructed event knowledge base, freely changing their "`start-time`" and "`end-time`" slot values as callbacks take care of the interval tree organization.

4.4 DELIBERATIVELY LEARNING ABOUT PHYSICAL ACTIONS

Table 2 and Table 3 present an example of a learning opportunity in the deliberative layer because of a failure to predict the effects of physical actions during deliberative planning. This learning opportunity results from an expectation failure that occurs when the SALS AI imagines and then actually executes the action of dropping a cube on a pyramid. Once the partial state transframe has been computed for a given action resource execution event, this transframe is used to train a rule-learning algorithm that attempts to categorize the partial state preconditions for different sets of the possible "`add`" ($+$) and "`remove`" ($-$) changes. A rule-learning algorithm is used in the SALS AI to predict the transframe

A.	"a pyramid shaped block to be sitting on a white colored table"
B.	"a gripper that is me to be above a pyramid shaped block"
C.	"a gripper that is me to be holding a cube shaped block"
D.	"a cube shaped block to be sitting on a pyramid shaped block"
E.	"a cube shaped block to be sitting on a white colored table"

Table 2: A selection of physical partial states that occur during the example learning story presented in chapter 1.

Partial State	*Precond.*	*Expected Trans.*	*Expected Post-cond.*	*Actual Trans.*	*Actual Post-cond.*
A	1		1		1
B	1		1		1
C	1	−	0	−	0
D	0	+	1		0
E	0		0	+	1

Table 3: An example expectation failure when the deliberative layer incorrectly hypothesizes physical partial states. Refer to Table 2 for definitions of partial states, A–E. Shown are the imagined and actual transitions for executing the action of dropping a cube on a pyramid. The fact that the expected transframe does not match the actual transframe presents the AI with a learning opportunity. The shaded area highlights the expectation failure. The symbol "1" represents that the partial state exists. The symbol "0" represents that the partial state does not exist. The symbol "+" represents that the partial state is added. The symbol "−" represents that the partial state is removed.

change	*positive example* ABCDE	*negative example* ABCDE
−A		11100
+A		11100
−B		11100
+B		11100
−C	11100	
+C		11100
−D		11100
+D		11100
−E		11100
+E	11100	

Table 4: Positive and negative training preconditions from the physical learning example presented previously in Table 3. This learning situation involves two changes that led to incorrect predictions: "+D" and "+E." The addition of partial state "D" is incorrectly expected, while the addition of partial state "E" is incorrectly *not* expected. The shaded areas represent the expectation failure, the failure to predict the correct state changes. The precondition state, "11100," can be used as a positive or negative training example for the appropriate version hypothesis spaces for learning new rules for predicting these changes correctly in the future.

that will result from executing a resource in given preconditions. This rule-learning algorithm is based on *version spaces* (Mitchell 1997).

Version spaces are an efficient representation for all possible conjunctive functions that map a set of binary inputs to a single binary output. These functions are also called *hypotheses*. When these hypotheses turn out to be wrong given new data, the hypothesis version space is refined. In the case of a false positive, the most general functions of the version space are specialized. In the case of a false negative, the most specific hypotheses of the version space are generalized. Table 4 shows an example of preconditions being used as positive and negative training examples for the version hypothesis spaces that lead to an expectation failure in the example previously presented in Table 3. In the SALS AI, the binary inputs to the rule-learning algorithm represent whether or not a partial state exists in the preconditions of the resource execution event. The single binary output from the rule-learning algorithm repre-

sents whether or not a set of partial state transframe changes will be the result of executing the resource in the given partial state preconditions. These hypothesis version spaces are used to predict the effects of natural language plans in the counterfactual knowledge base of each planning layer during the interpretation and imagination process described in chapter 3.

4.5 REFLECTIVELY LEARNING ABOUT DELIBERATIVE ACTIONS

Table 5 and Table 6 present an example of a learning opportunity in the reflective layer because of a failure to predict the effects of deliberative planning actions during reflective planning. This learning opportunity results from an expectation failure that occurs when the SALS AI imagines and then actually executes the action of executing a plan to "`stack a cube on a pyramid`." In the example story presented in chapter 1, the reflective layer predicts that executing the plan to "`stack a cube on a pyramid`" will not result in any deliberative plans having physical execution failures, a negative goal state that the reflective planner is trying to avoid in the deliberative plan knowledge base. When the reflective plan decides to execute this deliberative plan, based on the prediction that it will not lead to failure, there is an expectation failure at the reflective layer when the deliberative plan actually does fail during its execution. This expectation failure in the reflective layer is shown as highlighted areas of Table 5 and Table 6. This failure of the reflective layer to predict the effects of deliberative actions on the deliberative plan knowledge base presents the SALS AI with a learning opportunity at the reflective layer. Table 7 shows an example of preconditions being used as positive and negative training examples for the version hypothesis spaces that lead to an expectation failure in the example previously presented in Table 3.

A.	"a deliberative planner to be focusing on a plan that has been imagined"
B.	"a deliberative planner to be focused on a plan that is hypothesized to cause a cube to be on a pyramid"
C.	"a deliberative planner to be focused on a plan that is hypothesized to cause a pyramid to be on a cube"
D.	"a deliberative planner to be focusing on a plan that has been executed"
E.	"a deliberative planner to be focused on a plan that has failed in execution"

Table 5: A selection of deliberative plan partial states that occur during the example learning story presented in chapter 1.

Partial State	*Precond.*	*Expected Trans.*	*Expected Post-cond.*	*Actual Trans.*	*Actual Post-cond.*
A	1		1		1
B	1		1		1
C	0		0		0
D	0	+	1	+	1
E	0		0	+	1

Table 6: An example expectation failure when the reflective layer incorrectly hypothesizes deliberative plan partial states. Refer to Table 5 for definitions of partial states, A–E. Shown are the imagined and actual transitions for executing the action of executing a plan to "`stack a cube on a pyramid`." The fact that the expected transframe does not match the actual transframe presents the AI with a learning opportunity. The shaded area highlights the expectation failure. The symbol "1" represents that the partial state exists. The symbol "0" represents that the partial state does not exist. The symbol "+" represents that the partial state is added. The symbol "−" represents that the partial state is removed.

change	*positive example* ABCDE	*negative example* ABCDE
−A		11000
+A		11000
−B		11000
+B		11000
−C		11000
+C		11000
−D		11000
+D	11000	
−E		11000
+E	11000	

Table 7: Positive and negative training preconditions from the deliberative plan learning example presented previously in Table 6. This learning situation involves one change that led to an incorrect prediction: "+E." The addition of partial state "E" is incorrectly *not* expected. The shaded areas represent the expectation failure, the failure to predict the correct state changes. The precondition state, "11000," can be used as a positive training example for the appropriate version hypothesis spaces for learning new rules for predicting this change correctly in the future.

4.6 LAZY ALLOCATION OF HYPOTHESIS SPACES

It would be inefficient to allocate a new hypothesis version space object for each type of "`add`" or "`remove`" change in a partial state transframe for a given action resource because sets of partial state changes often occur together and multiple hypothesis version space rule-learning algorithms would contain redundant information for these sets of partial state changes. To reduce the number of hypothesis version spaces that are allocated to each action resource to predict the partial state changes that it may cause when executed, each co-occurring set of partial state changes for each action resource is calculated.

For example, consider the physical partial state changes presented in Table 4. For the moment, imagine that the AI has had no previous experience and that the information in this table is the only experience available to the deliberative layer's hypothesis version space rule-learning algorithm. In this case, notice that many of the partial state changes have exactly the same positive and negative training examples thus far. Table 8 shows how hypothesis version spaces are not allocated in order to predict every partial state change. Given only the information in Table 4, Table 8 shows that only two hypothesis version spaces are allocated in order to predict 10 different partial state changes. As more learning experience is added to each layer of the SALS AI, these sets of partial state changes are subdivided and new hypothesis version spaces are allocated in this conservative manner. I refer to this conservative method of hypothesis version space allocation as "lazy," which avoids allocating what would be redundant hypotheses in multiple version spaces.

hypothesis version space change set	*change*	*positive example* ABCDE	*negative example* ABCDE
#1	−A		11100
	+A		11100
	−B		11100
	+B		11100
	+C		11100
	−D		11100
	+D		11100
	−E		11100
#2	−C	11100	
	+E	11100	

Table 8: Lazy allocation of hypothesis version spaces by grouping sets of transframe changes that occur in identical contexts. Each of these sets is only allocated one hypothesis version space rule-learning algorithm.

4.7 SUMMARY

In this chapter, I have described how a planning layer reifies a select subset of partial states from the layer below that it uses as an efficient representation for learning, imagination, and goal recognition. Also, I have discussed two examples of learning: (1) one focused on the deliberative planning layer learning from expectation failures in physical knowledge, and (2) one focused on the reflective planning layer learning from expectation failures in deliberative plan knowledge. Finally, I have described a method of "lazy" hypothesis version space allocation that avoids redundant hypotheses from being stored in multiple version spaces, allowing learning to predict entire sets of partial state changes from the layer below by using fewer non-redundant hypothesis version spaces. In the next chapter, I will describe how the examples in this thesis are programmed in the low-level lisp-like programming language and virtual machine. I hope that the open-source nature of the SALS AI implementation will allow this Substrate for Accountable Layered Systems to enable other researchers to apply these techniques to their own research problem domains, given the practical programming knowledge presented in the next chapter.

5

VIRTUAL MACHINE AND PROGRAMMING LANGUAGE

The SALS AI is a cognitive architecture that is constructed on a virtual machine and low-level Lisp-like programming language that implicitly supports the tracing of results and behavior of the system to the data and through the procedures that produced those results and that behavior (Morgan 2009). Good traces make a system accountable and help to enable the analysis of success and failure, and thus enhancing the ability of the system to learn from mistakes. The SALS virtual machine and low-level Lisp-like programming language collectively form the substrate that executes the entire SALS cognitive architecture. In this chapter, I will focus on the details of the low-level Lisp-like programming language that enable learning from failure. In chapter 4, I described the details of how an asynchronous learning algorithm can learn from these failures so that these failures can be avoided in future natural language plan executions through learning refined causal models of the hypothetical imagined effects of natural language plans. In this chapter, I will describe the details of the low-level Lisp-like language that enable these types of asynchronous learning algorithms.

The SALS virtual machine provides for general parallelism and concurrency, while supporting the automatic collection of audit trails for all processes, including the processes that analyze audit trails. The native support of a low-level Lisp-like language in the SALS architecture allows, as in machine language, a program to be data that can be easily manipulated by a program, making it easier for a user or an automatic procedure to read, edit, and write programs as they are debugged. Large concurrent and parallel systems are often difficult to build and debug because of the complex causal interactions between all of the different processes. For this reason, every parallel task in SALS has an associated *cause* object. If any task creates a new parallel task, then a new cause object is created for the child task with a parent cause object reference. Cause objects represent meta-information about a task, such as in what way a compiled procedure should be executed, which can be used for causally organizing the effects of groups of parallel tasks, controlling and monitoring the occurrence of any type of event through any of the procedural abstraction barriers of the SALS virtual machine. Every parallel process in the SALS virtual machine has a trace that we can

compute from, that we can reason about. The built-in processes for reasoning about reflective traces in the SALS virtual machine are critical to the asynchronous learning examples presented in the previous two chapters.

5.1 VIRTUAL MACHINE

The SALS virtual machine is designed to take advantage of the next generation of multithreaded and multicore CPUs, so that new reflective algorithms will more easily be designed to exhibit "strong" scaling properties (Sodan et al. 2010). To allow the development of these new types of algorithms, the SALS virtual machine includes an explicit representation, called a *virtual processor* object, that represents each hyperthread in each core of each CPU in the target hardware platform. Each of the SALS virtual processors is used in order to organize the scheduling of SALS *fiber* objects, which are the bytecode threads that execute in the SALS virtual machine. In this way, each SALS fiber can be assigned to a specific hyperthread in the target hardware platform. To prevent local cache misses for on-chip caches, the SALS virtual machine has a separate memory pool allocated for each of the SALS virtual processors. CPUs that have 4 cores and 2 hyperthreads per core, 8 memory pools are allocated when the SALS virtual machine is created. A major bottleneck in systems with large numbers of processors and cores are mutual exclusion or *mutex* locks that protect shared resources. The SALS virtual machine avoids all mutexes in the memory layer by using these separate dedicated memory pools for each hyperthread in each CPU core for memory allocation and concurrent garbage collection. Mutexes are provided in the SALS architecture but low-level memory allocation and collection can operate at full-speed without any mutex locks for low-level memory operations. Figure 25 shows an example of how SALS allocates virtual processors for a hardware configuration with two processors, each having four cores and each core having two hyperthreads. Each hyperthread is organized into a binary tree structure that is used to calculate distances between hardware hyperthreads. This binary tree distance metric is used to dynamically distribute fiber loads across hyperthreaded CPU cores in order to attempt to utilize as much on-chip CPU cache as possible.

The SALS virtual machine has been tested with a maximum of 32 processors and 128 gigabytes of RAM on the Nautilus supercomputer at the National Institute for Computational Sciences. However, most of the examples presented in this dissertation have been executed on a

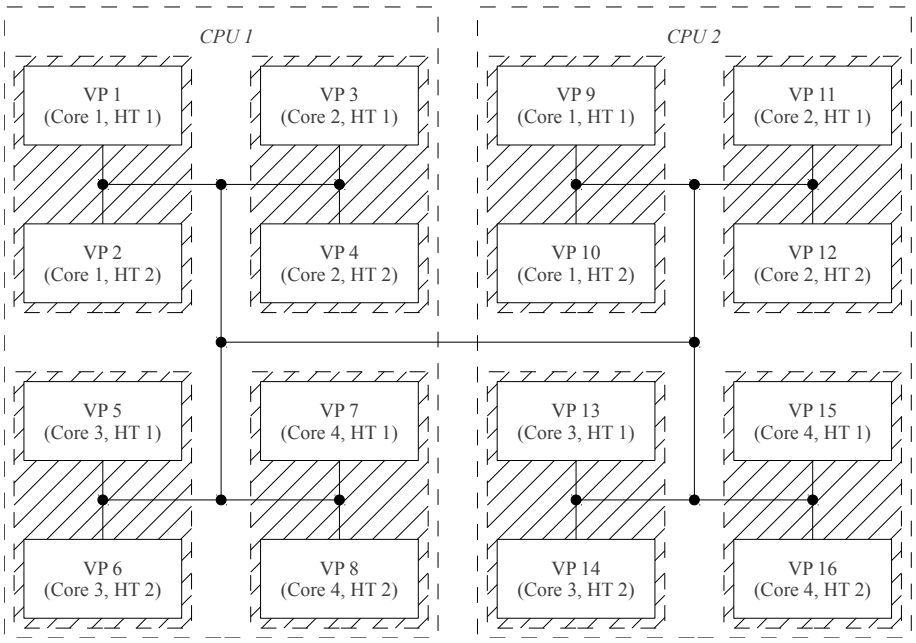


Figure 25: An example of how SALS allocates virtual processors (VP) for a hardware configuration with two multithreaded multicore processors, each having four cores and each core having two hyperthreads (HT). Each hyperthread is organized into a binary tree structure that is used to calculate distances between hardware hyperthreads. This binary tree distance metric is used to dynamically distribute fiber loads across hyperthreaded CPU cores in order to attempt to utilize as much on-chip CPU cache as possible.

bit range	*# of values*	*memory tier name*
1 – 17	131,072	`computer identity`
18 – 27	1,024	`pool index`
28 – 64	137,438,953,472	`block address`

Table 9: Bit allocation within the SALS memory pointer.

personal computer with only 4 CPU cores and 2 hyperthreads per core. The SALS virtual machine has a tricolor garbage collection algorithm that takes advantage of parallelizing work in concurrent processors. Memory pointers in the SALS memory system are 64-bit integers that have space reserved for three hierarchical memory tiers: (1) "`computer identity`," (2) "`pool index`," (3) and "`block address`." Table 9 shows the bit allocation within the 64-bit SALS memory pointer. The computer identity is zero if the memory pointer is referencing memory on the local computer. The computer identity is an integer greater than zero if the memory is on another computer that has connected to the SALS memory system through a low-level peer-to-peer socket connection. The ability of the SALS architecture to connect to other running SALS architectures allows it to potentially scale to take advantage of the hardware on multiple networked computers to solve one large shared-memory problem. The pool index is an integer reference into an array of memory pools on the given computer. On any given computer, one memory pool is allocated for each hyperthread in each CPU core on each CPU. The block address is an integer that is a block index into the given memory pool. A block size of one byte has been used for the examples presented in this dissertation but larger block sizes have been tested and can be used if more memory must be addressed.

5.2 PACKAGE MANAGER

The SALS programming language allows compiling and running computer programs that consist of many thousands of lines of code. For example, the examples presented in this dissertation depend on loading 154 packages, which are composed of over 30,000 lines of code that are written in the SALS low-level Lisp-like programming language. The SALS programming language includes a package manager to organize dependencies within this codebase. In addition to a low-level Lisp-like codebase of packages, the SALS virtual machine has the ability to load packages that contain optimized routines that are composed of compiled machine code. Machine code extensions to the SALS virtual machine are written in the C programming language. The C codebase that compiles to this machine code totals over 150,000 lines of C code. The virtual machine compiles and runs on any POSIX compliant operating system. The following is a declaration of a package definition in the SALS low-level Lisp-like language that helps to organize these different types of dependencies within the SALS codebase:

```
[defpackage semantic_knowledge_base
  :packages          [equals_hash
                      forgetful_event_stream
                      semantic_realm
                      semantic_causal_event
                      semantic_relationship_key
                      semantic_frame
                      semantic_object
                      semantic_event]
  :sources           ['semantic_knowledge_base-core.sals']
  :dynamic_libraries ['libf2e_semantic_knowledge_base.so']]
```

The "`defpackage`" expression in the SALS programming language means "define package." The first argument to the defpackage expression is the name of the package to be defined, "`semantic knowledge base`" in this case. The remaining arguments to the defpackage expression are optional and three such arguments are shown in this example: (1) "`packages`," (2) "`sources`," and (3) "`dynamic libraries`." The "`packages`" optional argument specifies a list of other packages that must be loaded before this package is loaded. The "`sources`" optional argument specifies a list of SALS files that are to be loaded when this package is loaded. The "`dynamic libraries`" optional argument specifies a list of files that contain machine code extensions that should be loaded into the SALS virtual machine when this package is loaded.

Every package and machine code extension to the SALS virtual machine and programming language performs "causal reflective tracing" on each concurrently executing process. The following sections describe examples of different uses of the causal reflective tracing features that have been implemented throughout the SALS virtual machine and its extensions. At the end of this chapter, I will describe one high-level package in the SALS programming language that performs the conjunctive hypothesis space version space rule-learning algorithm, which is key to the asynchronous experiential learning presented in chapter 4.

5.3 CAUSAL REFLECTIVE TRACING

Parallel tasks in the virtual machine are called *fibers* in order to distinguish them from the *threads* in the underlying operating system kernel. The parallel fibers create, mutate, and read from memory as they execute sequences of compiled bytecodes. At any point between bytecode executions, the memory system is static and can be saved to or loaded from non-volatile memory, such as a harddisk. The current execution state of every fiber is represented in the global environment. To be fully reflective on all of the procedural effects of any given fiber, I introduce a technique called *causal reflective tracing*. Causal reflective tracing is a way of defining variables that are specific to each fiber that can be used to control the low-level memory access, mutation, and creation functions. This allows one fiber to subscribe to the procedural trace events of another fiber without receiving procedural trace events of its own execution, which would lead to an infinite regress, halting the system. Further, because the virtual machine is inherently a parallel processing system, a given fiber will often start a number of child fibers that handle part of the processing for the parent fiber. When a new fiber is created, the child fiber inherits the causal variable bindings of its parent fiber, enabling the same procedural tracing options for the child as well. Causal reflective tracing is one of the basic tools for keeping track of which pieces of memory were created, mutated, or read by which other fibers. When evaluating the theoretical time complexity of concurrent procedurally reflective control algorithms, it should be noted that creating procedural trace events for the execution of a given fiber slows the fiber down only by a constant factor, not affecting the algorithm's "big-O" time complexity on an ideal concurrent shared-memory system, of which the virtual machine is an approximation.

While causal reflective tracing focuses the procedural event tracing to memory interactions of specific groups of fibers, this still results in

millions of events to consider every real-time second of execution. To further focus on specific objects within the memory system, specific pieces of memory can be created called *semantic memory*. Semantic memory objects are created, mutated, and accessed in roughly the same way as all of the frame-based objects in the memory system with a few extra reflective tracing features. For example, semantic objects provide event streams that can be subscribed to by a number of different parallel listeners in different fibers. Also semantic object pointers are bidirectional to ease traversal by reflective pattern matching algorithms.

Because it becomes awkward to subscribe to each and every frame-based object that may be interesting to the reflective focus, semantic frames that are created by specific fibers can be added to collections of semantic frames that are called *semantic knowledge bases*. Semantic knowledge bases are good for organizing entire layers or subsets of reflective layers that contain different types of semantic frame objects. Semantic knowledge bases allow the same forgetful event stream subscription services as semantic frames with the additional capability of tracing the addition and removal of entire semantic frames to and from the knowledge base.

While knowledge base reconstructions are extremely fast to reference, $O(1)$, they require a duplication of the memory requirements of the focus knowledge base for every different point in time that is required. To allow efficient access of the state of knowledge bases at arbitrary points in the past, *semantic event knowledge bases* are another type of representation that is reflectively and asynchronously maintained to not slow down the primary procedure under reflective focus. chapter 4 describes how the SALS asynchronous learning algorithm uses the event knowledge base object to calculate partial state transframes. I will briefly review the features of the event knowledge base as well as a few details about how this object is initialized and used in the SALS low-level Lisp-like programming language.

The semantic event knowledge base stores a type of semantic frame called a *semantic event*. A semantic event is a type of semantic frame object that represents an interval in time, which may include partial knowledge if the start or end times do not exist, making the interval potentially open-ended. Semantic event knowledge bases are reflectively traced and the knowledge is always stored in two different representations, the basic semantic event frames as well as a balanced interval tree that always represents the current state of the semantic event knowledge base. The balanced interval tree allows accessing the state of the focus knowledge base in $O(\log(n))$ time, where n is the number of

events stored in the semantic event knowledge base. Although the time complexity is not as efficient as the constant, $O(1)$, access time of the reflectively reconstructed semantic knowledge base, the semantic event interval tree knowledge base only requires $O(n)$ memory complexity to allow access to the structure of the knowledge base at any point in the past, where n is the number of semantic events. Figure 26 shows how to create a new "`semantic_event_knowledge_base`" type object in the SALS low-level Lisp-like interactive programming language.

```
 in-> [new semantic_event_knowledge_base nil [new semantic_realm]]
out-> [semantic_event_knowledge_base
        semantic_event_tree         [semantic_event_tree ...]
        semantic_frame_set          [set ...]
        trace_remove_semantic_frame []
        trace_callback_funks_frame  [frame ...]
        semantic_realm              [semantic_realm ...]
        trace_event_stream          [forgetful_event_stream ...]
        trace_add_semantic_frame    []
        ...]
```

Figure 26: How to create a new "`semantic_event_knowledge_base`" type object. When "`semantic_event`" type objects are added, removed, or modified while they are in this type of knowledge base, the knowledge base updates an always-accurate event interval tree structure for all of the events for efficient, $O(\log(n))$, access to the events at any point in time.

By default, when there are no listeners to the procedural event streams of a semantic frame-based object, no reflective events are created, allowing the use of the object to run at full speed. When a listener subscribes to the procedural use of a specific semantic memory object, events are added to ordered streams for the listening subscribers. To conserve memory resources, when multiple parallel listeners are subscribed to a given event stream, only those events that have not already been seen by all of the subscribers are remembered. Once all subscribers have processed an event, all events before this event are forgotten. This type of memory conserving event stream is referred to as a *forgetful event stream*. In this way semantic frames report the addition and removal of slot values to reflective forgetful event stream subscribers. Once a knowledge base is created and we have an *important event stream iterator*, we can define a concurrent reflective fiber that processes the events reflectively after the real-time execution of the processes that modify this knowledge base. Figure 27 shows an example of how a reflective fiber can be created to process a `forgetful_event_stream` generated by a `semantic_knowledge_base`, which includes the state of a planning

machine, represented by a `semantic_planner` object. Note that this example runs at a constant factor slower than full speed because of the creation of mutation events for the `semantic_planner` object, but this factor has been kept relatively small, so this example completes almost immediately. The reflective process runs slower because it must consider how to print these events to the terminal in a readable form. Figure 28 shows the last two events created by the last line of this example that mutates the `imagine_time` slot value for the planner.

```
 in-> [let* [[realm          [new semantic_realm]]
             [knowledge_base [new semantic_event_knowledge_base
                                  nil realm]]
             [iterator       [get knowledge_base
                                  new-event_stream_iterator]]
             [planner        [new semantic_planner realm]]]

        'Start parallel reflective fiber.'
        [fiber [funk []
                 [while t
                   [let [[event [have iterator wait_for_current]]]
                     [print event]
                     [have iterator increment]]]]
               []]

        [set planner trace_add    t]
        [set planner trace_remove t]

        [have knowledge_base add_semantic_frame planner]

        [set planner imagine_time [new semantic_time [time]]]]
out-> []
```

Figure 27: A reflective fiber can be created to process a `forgetful_event_stream` generated by a `semantic_knowledge_base`, which includes a traced `semantic_planner`.

While every parallel fiber is allocated a unique `cause` object that provides a representation of the reflective tracing features for that fiber, sometimes it is useful to collect events from overlapping sets or different hierarchies of many fiber objects. A convenient solution to causally organizing very dynamic sets of fibers is provided by a *cause group* object. Cause groups are added to cause objects and inherited from parent to child when new causes are created. Because cause objects and cause group objects exist at the locus of every memory creation and mutation event in the SALS virtual machine, they provide a means of tracing any event that modifies any part of the SALS virtual machine. Figure 29 shows an example of causal reflective tracing that takes advantage of

cause group objects to gather run-time statistics of a fiber execution. Figure 30 shows another example of using a hierarchy of multiple cause group objects to causally scope the tracing of 30 parallel fibers that are hierarchically organized into 3 cause groups that contain 10 fibers each.

Sometimes it is helpful to know which fiber or cause created a specific piece of memory. For this reason, every piece of memory in the entire virtual machine includes a reference to its creation cause as well as its creation fiber. Figure 31 shows how a fiber can retrieve the creation cause of a semantic event object, it is useful to know which fiber or cause is responsible for creating a given semantic event object so that a reflective fiber can learn from the separate effects of different causal scopes that make modifications to semantic objects within a given knowledge base. This form of causal tracing is used in the SALS cognitive architecture to detect when one resource interacts with another resource, for example, by activating that resource or waiting for that resource to complete execution. An entire reflective knowledge base of current parallel resource interactions is maintained by tracing the causes of resource interactions in this way.

```
[semantic_frame_event
  time           [time
                   years        2012
                   months       8
                   days         31
                   ...]
  event_type     remove
  semantic_frame [semantic_planner
                   property phenomenal_name [...]
                   property planner_type    [[]]
                   property imagine_time    [[semantic_time ...]]
                   relation execute_plan    [[]]
                   relation imagine_plan    [[]]
                   relation focus_plan      [[]]
                   ...]
  key_type       property
  key            imagine_time
  value          []]
```

```
[semantic_frame_event
  time           [time years 2012 months 8 days 31 hours 23 ...]
  event_type     add
  semantic_frame [semantic_planner
                   property phenomenal_name [...]
                   property planner_type    [...]
                   property imagine_time    [...]
                   relation execute_plan    [...]
                   ...]
  key_type       property
  key            imagine_time
  value          [semantic_time
                   value [time
                           years        2012
                           months       8
                           days         31
                           ...]]]
```

Figure 28: The last two events created by the last line of the example in Figure 27: "`[set planner imagine_time [new semantic_time [time]]]`". This command mutates the `imagine_time` slot value for the planner. Notice that the first of the two events is a `remove` type of event, while the second is an `add` type event. This event knowledge is used in the SALS AI to create reconstructions of entire knowledge bases of physical as well as deliberative object types, like planners. Note that the first event removes the `[]` slot value of the `imagine_time` `property` of the `semantic_planner` object, while the second event adds the new value, thus completing the mutation.

```
 in-> [globalize cause_group [new cause_group]]
out-> []

 in-> [with-new-cause
        [have [this-cause] add_cause_group cause_group]
        [partimes [i 10]
          [print i]]]
0
1
2
3
4
6
7

58
9
out-> []

 in-> cause_group
out-> [cause_group
        execution_nanoseconds 533248657
        bytes_allocated_count 2766181
        bytecode_count        19495
        bytes_freed_count     0]
```

Figure 29: Using a `cause_group` object to gather statistics from causally scoped executions. A new `cause_group` object is first created in the global environment. The gathering of run-time statistics involves creating a new cause with the "`with-new-cause`" operator, adding this cause to the `cause_group` object, and subsequently running an experiment. This example creates ten parallel fibers that each print their numerical index from 0 to 9. At the final output of the example, the global `cause_group` object is printed to the screen, showing processor execution time, total bytes allocated, total bytecodes executed, and also the number of bytes garbage collected during the experiment.

```
 in-> [globalize cause_group [new cause_group]]
out-> []

 in-> [let [[begin_time [time]]
            [frame      [new frame]]]
        [with-new-cause
          [have [this-cause] add_cause_group cause_group]
          [partimes [i 3]
            [let [[subcause_group [new cause_group]]]
              [have frame add i subcause_group]
              [with-new-cause
                [have [this-cause] add_cause_group subcause_group]
                [partimes [j 10]
                  [terminal_format standard-terminal j]]]]]]
        [have frame add 'real-time [- [time] begin_time]]
        frame]
024031548162062794915695337878
out-> [frame
        0         [cause_group
                    execution_nanoseconds 3535418608
                    bytes_allocated_count 7715763
                    ...]
        2         [cause_group
                    execution_nanoseconds 3012761500
                    bytes_allocated_count 7590735
                    ...]
        1         [cause_group
                    execution_nanoseconds 3976116760
                    bytes_allocated_count 9598095
                    ...]
        real-time [relative_time
                    seconds      1
                    milliseconds 694
                    ...]]

 in-> cause_group
out-> [cause_group
        execution_nanoseconds 10730429054
        bytes_allocated_count 25622218
        bytecode_count        215649
        bytes_freed_count     0]
```

Figure 30: Gathering run-time statistics using parallel hierarchies of causal scopes. Overall, 30 parallel fibers are created, each prints a number from 0 to 9, and run-time statistics are gathered in two layers of causal scope hierarchy. The overall `cause_group` object is shown last, while three sub-`cause_group` objects are printed as the return value of the second expression. Notice that the algorithm uses 10.7 seconds of processor time, while only using 1.7 seconds of real-time.

```
 in-> [let* [[realm          [new semantic_realm]]
             [knowledge_base [new semantic_event_knowledge_base
                                  nil realm]]
             [iterator       [get knowledge_base
                                  new-event_stream_iterator]]
             [planner        [new semantic_planner realm]]]

       'Start parallel reflective fiber.'
       [fiber [funk []
                [while t
                  [let* [[event       [have iterator
                                            wait_for_current]]
                                [event-cause [get event cause]]]
                    [if [eq 'my-test [have event-cause lookup
                                           'cause-name]]
                        [print 'event-in-causal-focus]
                      [print 'event-out-of-causal-focus]]
                    [have iterator increment]]]]
              []]

       [set planner trace_add    t]
       [set planner trace_remove t]

       [have knowledge_base add_semantic_frame planner]

       [with-new-cause
         [cause-define cause-name 'my-test]
         [set planner imagine_time [new semantic_time [time]]]]]
out-> []
event-out-of-causal-focus
event-out-of-causal-focus
event-out-of-causal-focus
event-out-of-causal-focus
event-out-of-causal-focus
event-out-of-causal-focus
event-out-of-causal-focus
event-out-of-causal-focus
event-in-causal-focus
event-in-causal-focus
```

Figure 31: Causally scoped reflective event tracing. Since every piece of memory in the virtual machine has a reference to its `cause` object, causally focusing the reflective tracing example shown in Figure 27 is simply a matter of accessing the creation cause of each event, using the expression, `[get event cause]`.

5.4 CONJUNCTIVE HYPOTHESIS VERSION SPACE RULE-LEARNING

The SALS architecture includes a *version space* (Mitchell 1997) hypothesis rule-learning algorithm. A use of this hypothesis version space rule-learning algorithm was described in chapter 4 as part of a larger asynchronous learning algorithm that learns to predict the partial state *transframe* (Minsky 1975) effects of resource executions given the partial state preconditions of the execution. The hypothesis version space rule-learning algorithm included in the SALS architecture predicts one binary output feature given a number of binary input features. The benefit of the version space rule-learning algorithm is that it provides a relatively compact representation of an entire conjunctive hypothesis space by only representing the most general and most specific boundaries in the overall lattice of potential hypotheses. The included rule-learning algorithm is incremental, meaning that it refines its space of hypotheses as new examples of training data become known. The SALS AI hypothesis version space learning algorithm also includes removal callbacks for most general and most specific hypotheses, so that if these hypotheses are no longer supported given new training data, these hooks allow tracing dependencies for correcting hypothetically supported counterfactual knowledge in the counterfactual knowledge base of each planning layer of the SALS AI. Figure 32 shows how to create a `concept_version_space` object that can be trained. Figure 33 shows an example of training the `concept_version_space` object, given training examples. Figure 34 shows how hypothesis removal callbacks can be used to monitor the loss of support for a specific example, attempting to find new support whenever the previous hypothetical support is lost.

```
 in-> [new concept_version_space]

out-> [concept_version_space
        specific_hypotheses [[concept_version_space_hypothesis]]
        general_hypotheses  [[concept_version_space_hypothesis]]
        variable_name_set   [set elements []]]
```

Figure 32: Creating a new `concept_version_space` conjunctive hypothesis learning algorithm. This object keeps a list of the most specific hypotheses that support positive output predictions as well as the most general hypotheses that support negative output predictions. There is a set of variable names, or frame slot names, that the algorithm has seen previously.

```
 in-> [let [[concept [new concept_version_space]]]

        'Add a positive training example to the concept.'
        [let [[example [new concept_version_space_example t]]]
          [have example add_variable_value 'color 'blue]
          [have example add_variable_value 'shape 'cube]
          [have concept train_on_example example]]

        [terminal_format standard-terminal
                         '\ntrain-1: ' concept]

        'Add a negative training example to the concept.'
        [let [[example [new concept_version_space_example nil]]]
          [have example add_variable_value 'color 'blue]
          [have example add_variable_value 'shape 'pyramid]
          [have concept train_on_example example]]

        concept]
train-1: [concept_version_space
           specific_hypotheses [[concept_version_space_hypothesis
                                  shape cube
                                  color blue]]
           general_hypotheses  [[concept_version_space_hypothesis
                                  shape ?
                                  color ?]]]
out-> [concept_version_space
        specific_hypotheses [[concept_version_space_hypothesis
                               shape cube
                               color blue]]
        general_hypotheses  [[concept_version_space_hypothesis
                               shape cube
                               color ?]]]
```

Figure 33: Training a `concept_version_space` conjunctive hypothesis learning algorithm. A new `concept_version_space` is created. Two `concept_version_space_examples` are created, one positive and one negative. The concept is printed after learning from the positive example. The final concept is returned after being trained on both examples.

```
 in-> [let [[concept [new concept_version_space]]]
        [print 'Training on positive example.']
        [let [[example [new concept_version_space_example t]]]
          [have example add_variable_value 'color 'blue]
          [have example add_variable_value 'shape 'cube]
          [have concept train_on_example example]]
        [print 'Adding removal callbacks to hypotheses.']
        [let [[example [new concept_version_space_example t]]]
          [have example add_variable_value 'color 'green]
          [have example add_variable_value 'shape 'cube]
          [label find_new_support []
            [mapc [funk [hypothesis]
                    [terminal_format standard-terminal
                                     '\nNew supporting '
                                     'hypothesis: ' hypothesis]
                    [have hypothesis add_removal_callback
                                     [funk []
                                       [print 'Lost support.']
                                       [find_new_support]]
                                     []]]
                  [get concept supporting_hypotheses example]]]
          [find_new_support]]
        [print 'Training on negative example.']
        [let [[example [new concept_version_space_example nil]]]
          [have example add_variable_value 'color 'blue]
          [have example add_variable_value 'shape 'pyramid]
          [have concept train_on_example example]]
        [print 'Done.']
        nil]
'Training on positive example.'
'Adding removal callbacks to hypotheses.'
New supporting hypothesis: [concept_version_space_hypothesis
                            shape ?
                            color ?]
'Training on negative example.'
'Lost support.'
New supporting hypothesis: [concept_version_space_hypothesis
                            shape cube
                            color ?]
'Done.'
out-> []
```

Figure 34: Hypothesis removal callbacks monitor the loss of support for a specific example, attempting to find new support whenever the previous hypothetical support is lost. In this case, a positive example of a `green cube` is attempting to maintain hypothetical support in the concept's hypothesis space. Each sharp cornered rectangular box introduces an additional causal scope for all of the processes this interpretation step causes to execute.

5.5 COMPILING, IMAGINING AND EXECUTING A NATURAL LANGUAGE PLAN

The causal reflective tracing features introduced in the SALS virtual machine allow simple but powerful forms of organizational structure in the implementation of complex reasoning processes. As an example, the process that compiles natural language plans is the part of the SALS AI that is the most complex and that benefits the most from the causal reflective tracing features of the virtual machine. Natural language plans are compiled in each planning layer, including the deliberative, reflective, and super-reflective planning layers. The main overview and structure of the natural language plan interpretation process is described in chapter 3. The low-level causal reflective tracing features of the SALS virtual machine are critical to the organization of this plan interpretation process. Figure 35 shows a procedural trace of a partially imagined plan execution during the compiling phase of natural language interpretation. Procedural traces such as this one are easy to create because of the causal context that is provided by default in the SALS virtual machine. The causal context for a natural language plan that is being interpreted is as follows:

- `execution_mode`: Compiled plans can either be interpreted in one of two modes: `imagine` or `execute`. When a plan is executed in the `imagine` mode, the plan uses learned hypothetical models of actions in order to predict their effects in a counterfactual event knowledge base within the planning layer. A plan is generally interpreted in the `imagine` mode before it is interpreted in the `execute` mode. The reflective trace that is produced from interpreting a plan in the `imagine` mode is used for a second compile stage that produces an executable plan that contains neither natural language nor other forms of ambiguity. Sensory functions that check for the existence of partial states access the counterfactual knowledge base when a plan is interpreted in the `imagine` mode. When a plan is interpreted in the `execute` mode, actions are actually executed by activating and suppressing resources in the layer below. Also, a plan interpreted in the `execute` mode directly accesses the layer below when checking whether or not partial states exist for sensory functions.

- `top_plan`: The initial top-level plan that has caused the rest of the procedural trace structure to exist. A plan interpretation process usually begins with a natural language phrase, such as "`stack a cube on a pyramid`." Analogous plans are then found that might

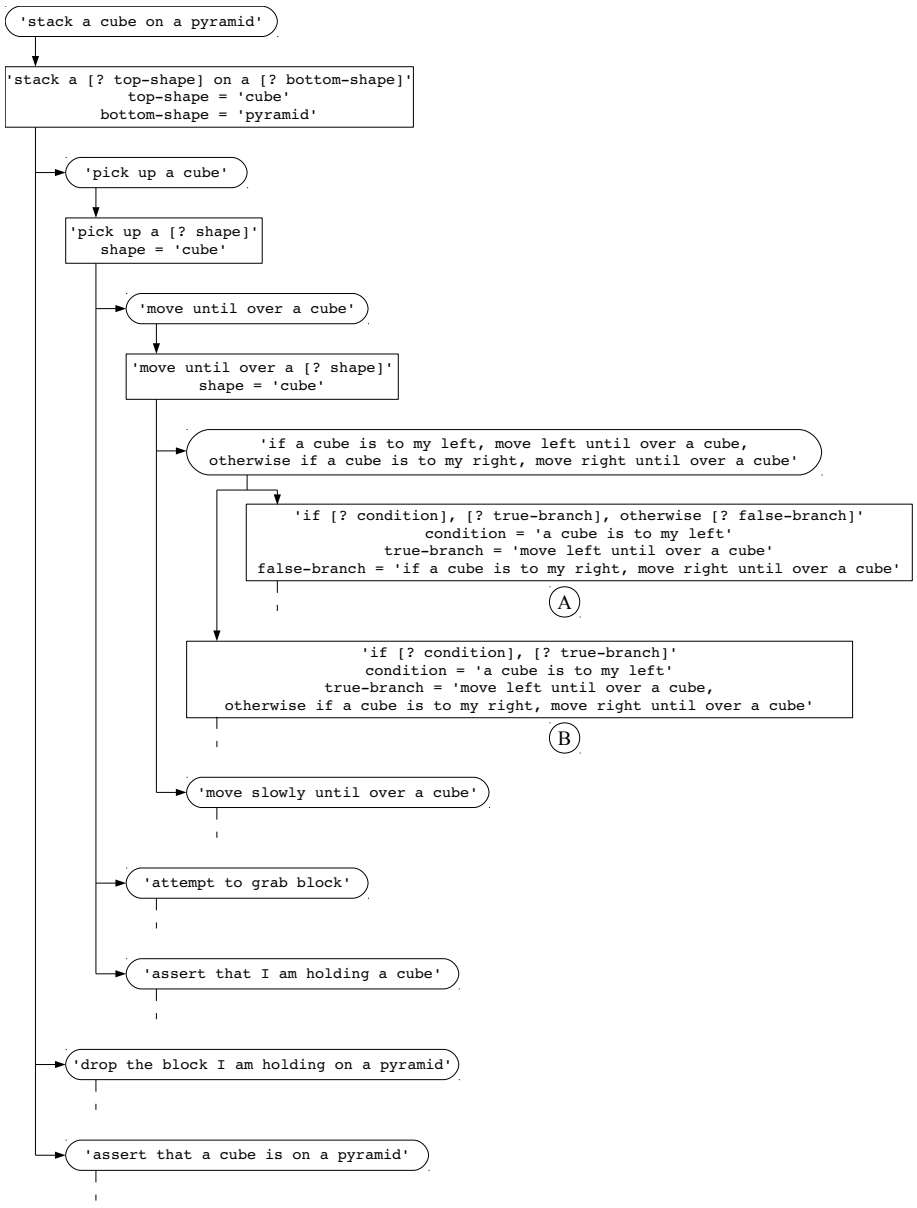


Figure 35: Procedural trace of partially imagined plan execution during compiling phase of language interpretation. Rectangles with rounded corners represent language expressions that must be interpreted. The top rounded rectangle is the initial language phrase that causes this entire procedural trace to be created. Rectangles with sharp corners represent abstract analogy pattern templates that match the rounded rectangle immediately above them. Analogy templates expand into multiple step plans that include more language phrases to be interpreted. Labels (A) and (B) mark two analogy patterns that simultaneously match the language phrase above them. (A) will eventually lead to a correct and complete interpretation, while (B) will lead to an incomplete interpretation.

provide interpretations of this initial phrase. These newly created analogous plans are each interpreted as top-level plans to see if the result of their imagined interpretations are complete and successful.

- `imagine_time`: Only used when the `execution_mode` is set to `imagine`, the `imagine_time` causal context slot stores the current hypothetical time that an imagined action would occur during the current imagined plan interpretation process. This causal context is used for the addition of partial state existence events to the counterfactual event knowledge base of the planning layer. Also, this imagined time is used when imagined sensory functions access the counterfactual event knowledge base, which stores all events in an interval tree for efficient access.

- `check_exists_partial_state_set`: A set of partial states that have been checked for by sensory functions within the currently interpreted plan. This set is used as an auxiliary set of hypotheses about the effects of plan execution. This set is useful when the SALS AI has very little experience executing a specific type of plan or set of actions. For example, if a plan assures a partial state exists for "`a gripper that is me to be holding a cube`," then although the SALS AI has no experience executing this plan, it can hypothesize that this might be an effect of the plan, given this set from the causal context of the plan interpretation.

- `current_plan_execution_set`: When the `execution_mode` is set to `imagine`, all possible conditional branches of a plan are interpreted and their effects are imagined with distinct causal contextual scopes. The `current_plan_execution_set` is used to keep track of which plans are currently being interpreted on the plan stack. This allows for stack-based recursive loops to be compiled and their effects completely imagined, while avoiding the potential infinite loops that may result from the actual execution of these plans.

- `imagine_event_tree`: The current set of events that have been imagined within this causal scope. This is necessary for scoping the results returned from the counterfactual event knowledge base required for sensory functions to check whether or not partial states may hypothetically exist.

- `resource_activation_set`: The current set of resource activations that have occurred within this causal scope.

The ability to quickly add various new forms of causal context that are available at any point in a compiled plan greatly simplifies that task of maintaining intricate procedural traces of the most complex aspects of the SALS AI, the multiple-staged process of compiling natural language plans, including processes that imagine the effects of various interpretations during the second compile stage.

5.6 SUMMARY

The SALS virtual machine and low-level programming language are the foundation upon which the SALS cognitive architecture has been constructed. I have described the details of the concurrent hardware abstractions, including multiple core and hyperthreaded CPUs as well as distributed memory pointers. The low-level basics of causal reflective tracing have been discussed, along with how a simple rule-learning algorithm may be applied to a hypothetical prediction, and how new hypothetical support for knowledge can be regained when previous supports have been lost. Basic programming examples have been given for implementing basic causal reflective tracing as well as this simple rule-learning example. Finally, I have discussed how the most complex processes in the SALS AI are organized by the scoping of causal contexts in the multiple-staged compile process that is used for compiling natural language plans through imagining their possible conditional effects. In the next chapter, I will describe related research to the four contributions that I have described in the previous four chapters. In chapter 7, I will present multiple evaluations that defend the point that the time-complexity of the SALS AI only increases linearly as more reflective layers are added to the architecture.

Part II

RELATED WORK

6

RELATED MODELS

The term reflection is a commonly used word in computer science and AI. The idea is extremely simple and is a modeling contribution of this dissertation, but because of its simplicity, it is a widely applicable idea. In fact, Maes (1987, 1988) distinguishes over 30 different types of "computational reflection," grounded in the computer science literature. The type of computational reflection that is introduced in this dissertation is not included in Maes' overview, although it is based on many of the forms of computational reflection that Maes describes, i.e. procedural reflection, type reflection, and frame reflection.

The SALS AI has been inspired by a previous implementation of an Emotion Machine cognitive architecture, called *EM-ONE* (Singh 2005). EM-ONE implements reflective thinking using a commonsense narrative representation based on an Allegro Prolog extension of Allegro Lisp. The EM-ONE architecture is a *critic-selector* model of problem solving (Sussman 1973, Singh 2002, Singh et al. 2004, Singh & Minsky 2005, Singh 2005, Minsky 2006, Morgan 2009). Knowledge in the EM-ONE architecture is divided into three domains: (1) physical, (2) social, and (3) mental. The EM-ONE AI controls a physical simulation that contains two one-armed robots that can work together to build a table. The EM-ONE AI contains three layers of reflective control: (1) reactive, (2) deliberative, and (3) reflective. While the EM-ONE cognitive architecture represents the first major implementation of the Emotion Machine theory of mind, it was limited in a number of ways:

1. Critics are specified in a declarative logical form, which does not allow for learning to optimize the procedural aspects of Prolog's implicit declarative search.
2. The implemented *Critic-L* language allows for inserted procedural Lisp code, but any procedural code inserted in this way is not reflectively traced.
3. Only learns from being told commonsense narratives and does not learn from its experience to better predict the effects of executing its physical or mental actions.
4. Commonsense narratives cannot be specified in natural language.

5. All activities execute in serial, so no critics or selectors can execute in parallel. Reasoning activities in each layer also occur in serial, so that the layers of control cannot execute concurrently.
6. Does not take advantage of multiple CPUs or CPUs with multiple cores.
7. Allegro Lisp and Allegro Prolog are expensive tools, barring collaborative research with many independent researchers that cannot afford such tools.

The SALS cognitive architecture aims to provide one cohesive solution to these limitations in the foundation of the EM-ONE Emotion Machine implementation. In focusing on solving these limitations, the SALS architecture has failed to model some good aspects of the EM-ONE architecture. The following are a number of parts of EM-ONE that are still future research for the SALS AI:

1. Self-reflective social knowledge.
2. The ability to refer to arbitrary partial states of a problem domain.
3. Critics.

The EM-ONE architecture includes a separate knowledge base for storing self-reflective social knowledge, the knowledge in the minds of other AIs, such as their beliefs and goals. The ability of an AI to learn abstract models of its own mind and use these models to hypothesize the state of mind in other AIs is referred to as *self-reflective* thinking in the Emotion Machine theory. This type of self-reflective thinking is also referred to as "theory of mind" in the cognitive science literature and has been found to exist in different neural circuits in the brain when compared to deliberative or "executive" functions (Saxe et al. 2006). Because the EM-ONE architecture is based on a Prolog substrate, it has the ability to refer to arbitrary partial states of a problem domain, while the SALS AI is currently limited to the two simple "`relationship`" and "`property`" types of partial states. This limitation is not fundamental to the SALS approach. The addition of more specific types of partial states requires modification of the existing partial state reification process. Chapter 8 describes a plan for defining a process that would allow general types of subgraphs as partial states that could be reified automatically by the SALS AI. The EM-ONE architecture can easily define critics that recognize arbitrarily complex declarative patterns in the problem domain. The disadvantage of relying on a declarative substrate, such as Prolog, is that the procedural aspects of this substrate are not reflectively traced and cannot be optimized by the EM-ONE architecture. While the SALS

architecture must have explicit plans for recognizing more complex partial states, the fact that these procedures are plans in the SALS AI means that different procedures for recognizing these partial states in the problem domain can be reflected upon, modified, compared and optimized by the SALS AI. The addition of critics as well as self-reflective and self-conscious layers of thinking is described as a future extension for the SALS architecture in chapter 8.

6.1 COMPUTATIONAL METACOGNITION

The type of reflection implemented in this thesis is a form of *computational metacognition* (Cox & Raja 2008, 2010), which begins with a *ground level*, the problem domain for the AI to control. A reasoning layer reasons about and solves problems in this ground level problem domain. This problem solver is refered to as the *object level* reasoner. In addition to the object level reasoner solving problems in the ground domain, a *meta-level* reasoner solves problems in the object level reasoner. This cascading of two levels of reasoners, where one reasoner reasons about a problem domain and another reasoner that reasons about the first object level reasoner is similar to the lower four layers of the Emotion Machine architecture that have been implemented in this thesis. The bottom two reactive layers of the SALS AI can be roughly thought of as analogous to the ground level of computational metacognition. The object level of computational metacognition is a control loop that receives inputs from the ground level, processes these, and sends commands back to the ground level. The object level of computational metacognition is analogous to the deliberative planning layer in the SALS Emotion Machine cognitive architecture. The meta-level of computational metacognition completes two cascaded control loops: the object level controlling the ground level and the meta-level controlling the object level. The meta-level of computational metacognition is analogous to the reflective planning layer of the SALS Emotion Machine cognitive architecture. Table 10 shows how

Metacognition Meta Level	$\approx$	Emotion Machine Reflective Layer
Metacognition Object Level	$\approx$	Emotion Machine Deliberative Layer
Metacognition Ground Level	$\approx$	Emotion Machine Reactive Layers

Table 10: The levels of computational metacognition mapped to the Emotion Machine cognitive architecture presented in this dissertation.

the levels of computational metacognition map to the Emotion Machine cognitive architecture presented in this dissertation.

6.2 META-PLANNING AND META-PLAN RECOGNITION

Wilensky (1981) describes *meta-planning* as representing and using knowledge about planning in problem solving and natural language understanding domains. He describes *PAM*, a story understanding system, and *PANDORA*, a story understanding and problem solving system, that both use higher-level goals and plans that he calls *meta-goals* and *meta-plans*. The basic goals and plans in PANDORA are analogous to the goals and plans in the SALS deliberative planning layer, while the meta-goals and meta-plans are analogous to the goals and plans in the SALS reflective planning layer. Wilensky describes story understanding as a form of inverse planning or plan recognition (Kautz 1987, Charniak & Goldman 1993, Kerkez & Cox 2003). In this sense, a planning system is given a set of goals that are used to generate a sequence of actions that accomplish the goals, while a story understanding system is given a sequence of actions that are used to generate a set of goals that explain those actions. Wilensky emphasizes that both story understanding and planning require representations of meta-goals and meta-plans to reason about common types of human thinking. Wilensky gives the following two example story understanding problems:

1. John was in a hurry to get to Las Vegas, but he noticed that there were a lot of cops around so he stuck to the speed limit.
2. John was eating dinner when he noticed that a thief was trying to break into his house. After he finished his dessert, John called the police.

In Wilensky's first example, John is understood to have two goals: (1) to get to Las Vegas as quickly as possible, and (2) to avoid getting a ticket. Wilensky's reasoning system recognizes that these two goals conflict and that John has pursued the meta-goal of resolving goal conflicts. Because getting a ticket has more negative value than the positive value of getting to Las Vegas quickly, it understands that the goal of speeding is abandoned due to pursuing this meta-goal. In the second story, Wilensky's reasoning system recognizes that John has made an unintelligent decision to continue eating his dessert while someone is robbing his house. In terms of meta-goals, Wilensky's system recognizes that John did not have the meta-goal to delay pursuing a less valuable goal in light of the presence of a new more valuable goal. So, Wilensky's system is able to perform meta-planning and meta-plan recognition, which are both future research goals for the SALS architecture. There are a few key differences between the SALS architecture and PANDORA:

1. Goals and meta-goals in PANDORA are both considered to be the same type of knowledge and are stored in the same knowledge base that is reasoned about by one monolithic planner, while the SALS AI keeps these categorically different types of knowledge separated into hierarchical layers of different knowledge bases and types of planning processes.
2. PANDORA is not connected to an external problem domain, while the SALS AI is capable of responding to the various types of plan failures that result from executing plans, such as physical expectation failures.
3. PANDORA only learns from being told knowledge, while the SALS AI learns from both being told natural language plans as well as from the experience of executing these plans.

Winston (2011) describes *Genesis*, an AI that performs reflective story understanding on English language stories. The Genesis AI has a ground level problem domain that consists of the knowledge that is directly stated in the story. The ground level story knowledge in the Genesis AI is analogous to the physical knowledge in the SALS AI, except that the Genesis AI does not explicitly separate physical knowledge from knowledge about the intentions and emotional states of social agents, which I see as self-reflective knowledge, requiring the AI to have self-models, a future extension of the SALS AI. The Genesis AI has a deliberative reasoning layer that uses analogies to English stories that the AI has been told in the past to infer causal connections between the ground level story elements.

The knowledge that causally connects the ground level of the Genesis AI is referred to as an *elaboration graph*, which I will refer to as the deliberative elaboration graph. The deliberative elaboration graph in the Genesis AI is analogous to the deliberative plans and transframes in the SALS AI, which provide causal explanations for changes between physical partial states. The construction of the deliberative elaboration graph in the Genesis AI is analogous to the deliberative interpretation and compiling of natural language plans in the SALS AI. Above the deliberative reasoning layer in the Genesis AI is a reflective reasoning layer that has a separate collection of reflective English stories that it has been told in the past. Reflective English stories in the Genesis AI are used to find analogical causal explanations that are combined to create a reflective elaboration graph that explains the deliberative elaboration graph. Winston describes two of the primary motivating hypotheses that have guided the development of the Genesis AI:

1. *The Strong Story Hypothesis*: The mechanisms that enable humans to tell, understand, and recombine stories separate human intelligence from that of other primates.
2. *The Directed Perception Hypothesis*: The mechanisms that enable humans to direct the resources of their perceptual systems to answer questions about real and imagined events account for much of commonsense knowledge.

Winston sees the current Genesis AI to be mostly a demonstration of progress at demonstrating a solution to the strong story hypothesis. In pursuit of the directed perception hypothesis, Rao (1998) has implemented the Architecture for Visual Routines (AVR). Preliminary success at combining the Genesis AI with Rao's AVR AI have been demonstrated (Winston 2011). The AVR AI uses a visual processing plan language that is based on an idea originally proposed by Ullman (1984), *visual routines*. Ullman proposes that there are two stages to visual processing: (1) a uniform low-level calculation over the entire visual scene, such as calculating the $2\frac{1}{2}$D sketch, and (2) visual routines that extract abstract spatial relations. Visual routines define objects and parts by having a visual plan interpreter that is focused on a specific part of the low-level sketch at any given point in time. Ullman suggests the following primitive visual operations:

1. *Shift of Processing Focus*: A process that controls where a visual operation is applied.
2. *Indexing*: Locations that are "interesting" in the base sketch, such as a blue patch in an otherwise completely red scene.
3. *Bounded Activation or Coloring*: The spreading of activation from the point of focus to "fill" a local region, which stops at boundaries in the base representation.
4. *Boundary Tracing*: Moves the focus along the edge of a region in the base representation.
5. *Marking*: Remember the location under the current focus so that it can either be ignored or returned to in the future.

These primitive visual operations are used to construct visual routines that become plans to perceive (Pryor et al. 1992, Pryor & Collins 1995, Velez et al. 2011) in the low-level visual system.

Although the first planning layer in the SALS AI is the deliberative layer, one can imagine that a solution to combining Winston's Genesis AI with Rao's AVR AI would be to extend the SALS planning layers into the learned reactive layer of the SALS AI, so that the translation

of visual knowledge from the built-in reactive visual knowledge base to the physical knowledge base in the learned reactive layer would be performed by a planning layer below the deliberative layer. Thus, the technique of extending the SALS AI's planning layers to higher layers of super-reflective control could be just as easily reversed by extending the SALS AI's planning layers to lower layers of planned control, so that reflective control could be applied to the types of planning-to-perceive problems as a coherent integration of Ullman's visual routines into deliberative reasoning layers of reflective control.

6.3 OPTIMALITY IN METACOGNITION

AI researchers often approach problem solving from the perspective of theories of "rationality" from the fields of decision theory and economics. From this perspective, rationality requires the AI to decide upon optimal actions with respect to the values or costs of its goals and activities. In the basic formulation, different actions have different costs and the optimal decision is the decision that minimizes this cost over some time period, possibly an infinite horizon. In simple domains, the solution to a problem of optimal control can be specified in closed form (Bertsekas 1995). In complex domains, optimal decision making requires intractable computations to be performed, and approximate or "satisficing" solutions to problems become necessary (Simon 1957, 1982). Good (1971) describes a decision making problem that includes costs for acting as well as costs for decision making that he calls *type II rationality*, a type of metacognition. Zilberstein (2011) describes *optimal metareasoning* as an approach to developing a formalism for evaluating the performance of a problem solver that is performing type II rationality. Optimal metacognition does not imply that the object level problem solver is optimal but instead that the meta-level problem solver is optimal. In *bounded optimality* the object level problem solver has certain trade-offs, such as solution quality versus time, that can be optimally manipulated by the meta-level problem solver (Russell & Wefald 1991). Zilberstein (2011) describes bounded optimality:

> This approach marks a shift from optimization over actions to optimization over programs. The program is bounded optimal for a given computational device for a given environment, if the expected utility of the program running on the device in the environment is at least as high as that of all other programs for the device. When the space of programs

> is finite, one can certainly argue that a bounded optimal solution exists. Finding it, however, could be very hard.

While the SALS AI is neither an optimal problem solver nor an optimal meta-level problem solver, the field of optimal problem solving does have the attractive feature that there is an objective metric of performance for all problem solving algorithms when viewed through the lens of optimality. There are a few key differences between most optimal meta-planning algorithms and the SALS AI:

1. Optimal metacognitive algorithms only learn from experience, while the SALS AI learns both from being told natural language plans as well as from the experience of executing these plans.
2. Optimal metacognitive algorithms tend to only consider simple control parameters of an object-level algorithm, such as the allocated execution time for a contract or anytime algorithm, while the SALS AI considers all possible programs that implement object-level reasoning.

While the SALS AI does not currently have an objective performance metric, it is interesting to consider how the field of bounded rationality could benefit from being told deliberative and reflective natural language plans as a method of more quickly searching the space of all possible planning programs.

6.4 MASSIVELY MULTITHREADED PROGRAMMING

While the SALS architecture will run on any hardware platform that supports POSIX threads, SALS has been optimized to take advantage of hardware platforms that utilize multithreaded and multicore CPUs as well as multiple CPUs. Also, the SALS memory layer is designed to be extended to peer-to-peer grid processor configurations. To avoid excess heat dissipation, current chip designs are trending toward higher transistor counts with slower clocks and lower voltages. Toward this end, chip designs have increased numbers of processor cores, memory caches, and SIMD coprocessors on each chip. The traditional approach to High-Performance Computing (HPC) has been the Symmetric Multiprocessor (SMP) model, which assumes that two or more processors access a shared global memory in an equivalent way. The current trend toward multicore and multithreaded CPUs is beginning to make old assumptions about HPC obsolete. The pin-out and interconnect problem increasingly means that memory latency is the bottleneck for high-performance applications. The cache memory configurations in new

CPUs, such as the Intel Core i7, which includes four cores, each with two hyperthreads, as well as L1 and L2 cache, means that data locality to threads will only become more of a software design problem for HPC as the number of cores in each CPU is expected to continue to double every 18-24 months (Sodan et al. 2010, Dongarra et al. 2007). Indeed, in an extreme case, the ATI RV770 GPU has more than 1,000 cores with 10 hyperthreads in each core and 80 floating point units on a single chip with very limited access to any global memory. Given the right software architecture, these massively multithreaded chips use their parallelism and concurrency to hide resource latency problems. The current economical trend in HPC is to build computers with thousands of commercially available CPUs and GPUs, which means that these trends in the consumer PC market are beginning to require software changes for high-performance scientific modeling applications. Sodan et al. (2010) describes two types of performance scaling in parallel algorithms:

1. *strong scaling*, and
2. *weak scaling*.

In strong scaling, more processors linearly decrease the overall execution time for a given algorithm, regardless of problem size. In weak scaling, the problem size must increase with the number of processors to see a linear decrease in the overall execution time for the larger problem size. Algorithms that demonstrate weak scaling will improve with additional processors only up to a given constant number of processors. Thus, algorithms that demonstrate weak scaling will generally increase in speed for a given problem size only when individual processor cores increase in speed. Like many traditional high-performance software libraries, traditional high-performance linear algebra libraries, such as LAPACK and ScaLAPACK, have relied on weak scaling to take advantage of new hardware platforms with more processor cores. As processor core speeds increase, linear algebra problems of a given size will increase in speed, while as processor core numbers increase, problems involving larger matrices benefit from the additional processor cores. The problem is that the current trend is for processor core speeds to be plateauing or even decreasing as processor core numbers increase, causing these traditional weak scaling algorithms to actually slow down as multicore and multithreaded CPUs are used in the next generation of HPC. Exacerbating the problem, cores in the new multithreaded and multicore CPUs increasingly share on-chip resources, such as L2 and even L3 caches, as well as sets of heterogeneous specialty resources, such as GPUs, SIMD coprocessors, and NICs for more efficient communication between many CPUs. These shared on-chip resources make execution

speeds of a thread on a single core to be heavily dependent on how other threads on the chip use those resources. For example, on multi-threaded and multicore CPUs, threads should be grouped that access the similar memory locations to maximize the efficiency of the shared on-chip cache lines. In effect, new HPC architectures are not SMPs and require new algorithm, compiler, and scheduler designs that emphasize strong scaling so that larger numbers of simpler processing cores with heterogeneous shared resources result in performance gains, regardless of problem size.

6.5 NAUTILUS

The SALS virtual machine has been compiled and tested on *Nautilus*, an SGI Altix UV 1000 system, the centerpiece of National Institute for Computational Sciences (NICS) Remote Data Analysis and Visualization Center (RDAV) (`www.nics.tennessee.edu`). Nautilus is a SMP architecture with 1024 cores (Intel Nehalem EX CPUs), 4 terabytes of global shared memory and 8 GPUs in a single system image. Nautilus has a 427 terabyte Lustre file system, a CPU speed of 2.0 gigahertz and a peak performance of 8.2 teraflops. The SALS AI has been tested on Nautilus with an allocation of 32 processors and 128 gigabytes of RAM. The evaluations of the SALS virtual machine in chapter 7 focus on a desktop personal computer with an Intel Core i7 CPU with 4 cores each with 2 hardware hyperthreads.

Part III

EVALUATION, FUTURE, AND CONCLUDING REMARKS

7 EVALUATION

The primary contribution of this thesis is a recursive implementation of a reflective planning layer that controls a deliberative planning process. Because of the recursive nature of this implementation, a super-reflective layer is also used to control and learn about the reflective planning process. The benefit of adding each additional reflective layer is that more can be learned from each deliberative failure by adding each additional reflective layer, but there is a computational trade-off in that each additional reflective layer also increases the computational complexity of the overall architecture. This chapter evaluates the Emotion Machine cognitive architecture contribution of this thesis by measuring the computational complexity introduced by adding additional reflective planning layers to the SALS deliberative planning process. First, the perceptual inputs to each planning layer are described and evaluated. Next, the time-complexity of natural language plan interpretation at each reflective layer is evaluated. Finally, the performance of the SALS virtual machine at efficiently executing parallel processes concurrently on multicore and hyperthreaded CPUs is evaluated.

7.1 COMPLEXITY OF PERCEPTUAL INPUTS

The key to beginning to think about the computational complexity in a given planning layer of the SALS cognitive architecture is in considering the perceptual inputs to that layer. The perceptual inputs to a planning layer determine the complexity of the goals that the planner can attempt to accomplish or avoid as well as the complexity of the causal models that the planning layer will construct for a given rule-learning algorithm. There are two different types of inputs to a SALS planning layer from the layer below:

1. Procedurally Reflective Event Streams
2. Direct Read Access

The first, procedurally reflective event streams, is the basis of the asynchronous learning from experience that was discussed in detail in chapter 4. To briefly review, learning from experience asynchronously abstracts a small subset of all possible partial states from a stream of

events that represent changes that have occurred in the knowledge base that the planning layer is trying to control. These abstracted partial state events are woven together with resource activation and completion events through a rule-learning algorithm to learn abstract hypothetical models of resource executions in the layer below. These hypothetical models are used by the planning layer to imagine the effects of plans before they are executed. The second type of input to a planning layer, direct read access, allows an executing plan to access the real-time state of the knowledge base that it is trying to control. The important thing to realize about both of these different methods of accessing and referring to the knowledge base in the layer below is that both of these methods work exclusively through a small subset of possible partial state abstractions. chapter 3 describes two specific types of these partial state abstractions that can be perceived by the planning layer: (1) the "`relationship`" expression, and (2) the "`property`" expression. The fact that all perceptual input to a planning layer is limited to these specific types of partial state abstractions limits the complexity of the control problem that the planning layer confronts. For example, because of the simplicity of the included partial states, the SALS AI cannot directly pursue a single goal to create a stack of three blocks. Instead, the deliberative layer of the SALS AI must be instructed, by either a user or the reflective layer, to pursue two goals composed of these simpler partial states to make a plan to stack three blocks: (1) "`Block-1 to be on Block-2`" and (2) "`Block-2 to be on Block-3`." The important point is not that the abstracted partial states are currently simple but instead that the planning layer is *limited* to perceiving its problem domain through partial state abstractions that reduce the complexity of the problem domain to a set of symbolic reifications that either exist or do not exist in the control domain.

Knowledge in a SALS planning layer includes plans, goals, a planner, and other knowledge used in the planning process. In addition to the planning objects, knowledge in a planning layer includes symbolically reified references to the knowledge in the layer below. Keeping clear distinctions between knowledge in different layers is critical to reducing the potential complexity of the SALS architecture. For example, while the deliberative plan knowledge base references physical knowledge in the learned reactive physical knowledge base, the deliberative plan knowledge does not actually contain physical knowledge. Instead, the deliberative plan knowledge base contains symbolized reifications of potential partial states of the physical knowledge base. Figure 36 shows a simple example of a deliberative goal that refers to a potential partial

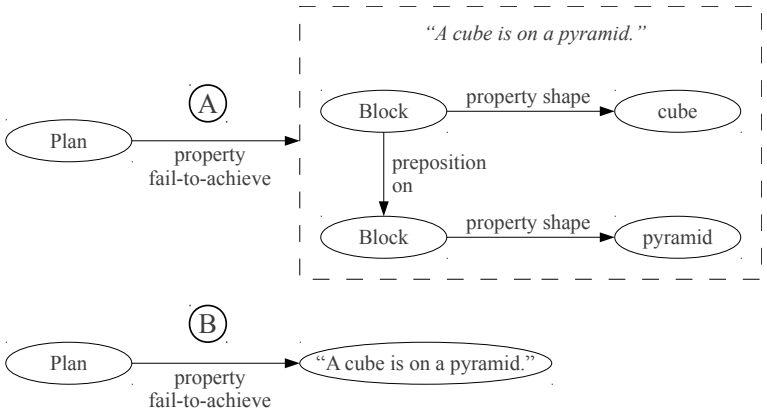


Figure 36: Deliberative physical partial state reification. (A) A visualization of the idea that a deliberative goal object can refer to a potential partial state of the physical knowledge base. (B) The actual representation of a deliberative goal object that has a symbolic "`partial-state`" property, the symbolic phrase, "`a cube is on a pyramid`," being a reference to the potential physical partial state visualized in (A). Each planning layer is limited to reflecting upon reified symbols that refer to partial states in the layer below and not the complexity of the partial states themselves.

state of the physical knowledge base and how this partial state is symbolically reified in the deliberative layer. Because deliberative knowledge only contains symbols that refer to physical partial states, this allows the deliberative layer to ignore the internal details of the physical partial state and simply treat the goal symbolically.

The number of partial states that are abstracted from any given knowledge base depends primarily on the structure of the knowledge within that knowledge base and whether or not that structure contains the types of partial states that the SALS AI is prepared to abstract. The partial states that are currently included in the SALS AI are meant to be both specific enough to handle the simple types of meaningful relationships that have been engineered to exist in each knowledge base while being general enough to leave room for those serendipitous abstractions that have not been engineered but may still end up being useful to the rule-learning algorithm in building causal models of resource executions. Table 11 shows how the sizes of the knowledge bases in the layers of the SALS AI relate to the numbers of partial states that are abstracted from these knowledge bases by the layers above. Notice that the total number of partial states that are abstracted from any given knowledge base is not directly a function of the number of frames in

Knowledge Base	*Frames*	*Partial States Abstracted*
Learned Reactive Physical	6	174
Deliberative Plan	120	122
Reflective Plan	224	103
Super-Reflective Plan	208	*N/A*

Table 11: A comparison of the sizes of knowledge bases in the layers of the SALS AI to the numbers of partial states that are abstracted from these knowledge bases. Notice that the number of partial states abstracted from any given knowledge base is not directly a function of the number of frames in the knowledge base. For example, the learned reactive physical knowledge base contains very few frames with a highly interconnected structure that contains many of the types of partial states that SALS is designed to abstract, while the plan knowledge bases contain a relatively large number of frames with fewer and more sparse relationships that result in fewer abstracted partial states. The super-reflective layer is currently the highest layer in the SALS AI, so no partial states are currently abstracted from the super-reflective plan knowledge base.

that knowledge base. For example, the learned reactive physical knowledge base contains relatively few frames with a highly interconnected structure that contains many of the types of partial states that SALS is designed to abstract, while the plan knowledge bases contain a relatively large number of frames with fewer and more sparse relationships that result in fewer abstracted partial states. The number of partial states that are the perceptual input to each planning layer *decreases* as subsequent layers of reflective planning are added to the SALS AI. In general, this rule may not hold for two reasons:

1. Planning layers may become more interconnected as the AI gains experience.
2. More types of partial states may be added to the SALS AI in the future.

Both of these situations may cause the total number of partial states abstracted from a plan knowledge base to grow intractably. Because of this potential problem, an eye must be kept on these numbers when designing new types of knowledge representations for each planning layer in the SALS AI. A general solution would consider the number of perceptual inputs to a planning layer as a control problem in itself. Currently, the number of perceptual inputs to a planning layer in the

SALS architecture grows sub-linearly with each subsequently higher layer of reflective control.

7.2 COMPLEXITY OF PLAN INTERPRETATION

While the perceptual inputs to higher-level planning layers in the SALS AI are currently kept at a tractable sub-linear complexity for each additional layer, there is still the question of how the planning process scales in time-complexity for each subsequently higher layer in the SALS AI. The planning process can be broken down into two main stages:

1. Plan Interpretation and Imagination
2. Plan Execution

Plan interpretation and imagination involves a search through a number of possible partial interpretations of natural language phrases expressed in the simple SALS template matching programming language, which has been described in chapter 3. To briefly review, the search through possible natural language interpretations involves finding analogies to other already known natural language plans that match the phrase that is currently being interpreted. The interpretation process (1) considers the current state of the domain that the planning layer is trying to control, (2) ignores interpretations that compile to programs that are imagined to have bugs, such as passing the wrong argument type to a low-level function, and also (3) ignores interpretations that do not accomplish the current goals of the planning layer or fail to avoid negative goals. Depending on the type of reflectively chosen planning process, plan execution generally occurs after a natural language plan has been interpreted, all natural language ambiguity has been compiled away, and the plan has been deemed relevant to either accomplishing positive goals or avoiding negative goals.

Because the plan interpretation process requires a search through a number of ambiguously specified analogies to previously known natural language plans, this process can potentially have a considerable time-complexity as natural language phrases become longer and more complex at higher level reflective layers. There is a distinction between the time-complexity of learning from being told and learning from experience. When a planning layer in the SALS AI learns from experience, a stream of procedurally reflective trace events are received from changes in the layer below and these are abstracted into partial states that are symbolically reified before being stored in the causal hypotheses of the planning layer. When these hypotheses are used to hypothesize the

effects of a plan, only the reified symbol that refers to the partial state in the layer below is stored as related to the plan in the planning layer. When a reflective layer above the planning layer learns from experience, it receives a stream of events that includes the new relationship between the plan and the symbolized partial state in the planning layer. The reflective layer itself symbolically reifies this partial state of the planning layer. As this process continues up through each additional reflective planning layer, each reflective layer performs one symbolic reification that is again symbolically reified by the next layer as part of a higher-level partial state that ultimately may refer, through symbolic reference, to the original physical knowledge, the ground knowledge layer.[1] Figure 37 shows an example of three layers of recursive symbolic reification. The following are three pieces of knowledge from different layers of the SALS AI, where the first piece of knowledge is symbolically referred to by the second, and the second is symbolically referred to by the third:

1. Physical Knowledge: "`a pyramid is on a cube`."
2. Deliberative Knowledge: "`a deliberative planner is focusing on a plan that has failed to achieve the goal for a cube to be on a pyramid`."
3. Reflective Knowledge: "`a reflective planner is focusing on a plan that has failed to avoid the negative goal for a deliberative planner to be focusing on a plan that has failed to achieve the goal for a cube to be on a pyramid`."

To continue to reference physical knowledge in higher layers, the natural language phrases become longer, more complex, and take more time to interpret and compile to their reified symbolic forms. Table 12 shows a comparison of the time-complexities for interpreting each of these three natural language phrases. As can be seen in this table, interpreting, imagining and compiling natural language phrases that refer to physical knowledge at higher layers of reflective planning grows quickly in time-complexity in just the first few layers. For this reason, while the SALS AI is capable of interpreting these types of natural language phrases that reference knowledge in multiple layers below a given planning layer, these types of natural language phrases are strictly avoided in the implementation of the natural language planning algorithms in

1 It should be clear that reflective knowledge has no requirement to be ultimately grounded in physical knowledge. Knowledge can ultimately refer to the partial states of any layer. See Minsky (2011) for a more detailed discussion of interior grounding in reflective and self-reflective thinking.

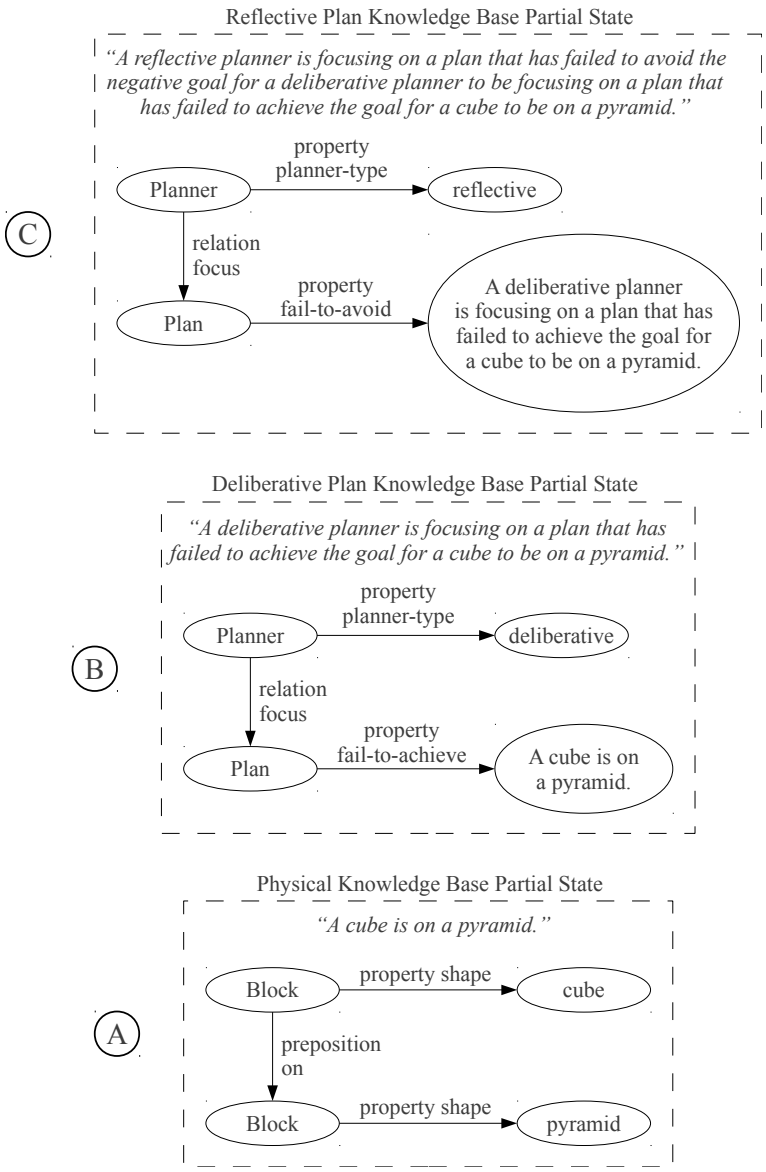


Figure 37: Three layers of knowledge that form a hierarchy of reified symbolic reference. (A) A physical partial state that is symbolically reified in the deliberative layer. (B) A deliberative partial state that is symbolically reified in the reflective layer. The deliberative partial state contains the symbolic reification of *A*. (C) A reflective partial state that is symbolically reified in the super-reflective layer. The reflective partial state contains the symbolic reification of *B*. Also, the reflective partial state indirectly references *A* through its direct reference to *B*.

	Referent	*Reference*	*Execution Nodes*
(A)	Physical	Deliberative	18
(B)	Deliberative	Reflective	92
(C)	Reflective	Super-Reflective	125

Table 12: A comparison of the execution node time-complexity of three different natural language phrases that reference one another in recursive hierarchy. (A) A natural language phrase that compiles to a physical knowledge partial state in the deliberative layer. (B) A natural language phrase that compiles to a deliberative knowledge partial state in the reflective layer that includes a symbolically reified reference to the physical partial state in *A*. (C) A natural language phrase that compiles to a reflective knowledge partial state in the super-reflective layer that includes a symbolically reified direct reference to the deliberative partial state in *B* and an indirect reference to the physical partial state in *A*.

the reflective and super-reflective layers. These types of hierarchical references through all of the layers below any given planning layer are commonly abstracted by the process of learning by experience, where the time complexity is sub-linear per additional layer, but while these knowledge references are possible to express in natural language to the SALS AI, they are avoided in the definitions of the plans that define the planning processes because of this problematic scaling factor in learning these types of complex knowledge from being told.

The only knowledge references that are told to the planning layers in the SALS AI are the goals in the immediate layer below the planning layer that the planning layer should try to accomplish or avoid. Examples of these goals are as follows:

1. Reflective Plan: "`want a cube to be on a pyramid`."
2. Reflective Plan: "`want a pyramid to be on a cube`."
3. Super-Reflective Plan: "`Avoid a deliberative planner being focused on a plan that has failed`."

Table 13 shows the time-complexities of interpreting these natural language phrases in the reflective and super-reflective layers that specify goals for the deliberative and reflective layers, respectively. The time-complexity of interpreting these natural language phrases that specify goals in the deliberative and reflective layers have equal time-complexity. This result implies a linear time-complexity for specifying more goals in each additional reflective layer in the SALS AI.

	Plan Layer	*Execution Nodes*
(A)	Reflective Plan	18
(B)	Reflective Plan	18
(C)	Super-Reflective Plan	18

Table 13: Time-complexities of interpreting natural language plans that specify goals in only the immediate layer below the plan. Note that the time-complexity of interpreting these natural language phrases that specify goals in the deliberative and reflective layers have equal time-complexity. This result implies a linear increase in time-complexity for each additional layer of plans that specify goals in only the one layer immediately below the plans. As previously shown, plans that specify goals that indirectly reference knowledge in layers multiple levels below the plan have a greater than linear increase in time-complexity.

Because the natural language plans in the reflective and super-reflective layers that define the planning algorithms are independent of any specific goals that they may be trying to accomplish, these plans do not contain any knowledge references at all. Instead, the natural language plans that define the planning algorithms are simply composed of resource activations in the immediate layer below. The following are plans of action in each of the three planning layers of the SALS AI:

1. Deliberative: "`stack a cube on a pyramid`."
2. Reflective: "`find recent plan to accomplish my positive goals`."
3. Super-Reflective: "`find recent plan to avoid my negative goals`."

Table 14 shows the time-complexity of interpreting these plans for action in the each of the three planning layers of the SALS AI. The reflective planning plans are roughly of the same time-complexity as the super-reflective planning plans. The fact that planning plans are of the same time-complexity independent of their interpretation layer implies a linear increase in overall complexity as more planning layers are added to the SALS AI.

	Plan Layer	*Execution Nodes*
(A)	Deliberative Physical Plan	3003
(B)	Reflective Planning Plan	170
(C)	Super-Reflective Planning Plan	152

Table 14: The time-complexity of interpreting plans for action in the three planning layers of the SALS AI. (A) The deliberative physical plans take much more time-complexity to interpret because they are idiosyncratic to the Blocks World planning domain, including specific references to shapes, colors, and prepositional relationships. Each different problem domain will have a different set of deliberative plans in order to manipulate that domain. (B) The reflective planning plans are interpreted and become the deliberative planning processes. (C) The super-reflective planning plans are interpreted and become the reflective planning processes. Notice that the reflective planning plan is roughly of the same time-complexity as the super-reflective planning plan. The fact that planning plans are of the same time-complexity independent of their interpretation layer implies a linear increase in overall complexity as more planning layers are added to the SALS AI.

7.3 EFFICIENCY OF CONCURRENT EXECUTION

The SALS virtual machine is built to take advantage of multiple processors and multicore processors with hardware hyperthreads. chapter 5 discusses the details of the parallel and concurrent processing features of the SALS virtual machine and low-level Lisp-like programming language. To briefly review, to reduce cache misses, a separate memory pool is allocated for each separate hardware hyperthread in the target hardware platform. Each of these memory pools is allocated a virtual processor object in the SALS virtual machine. Each concurrent virtual processor has a separate scheduler that simulates parallelism for a number of parallel processes, called *fibers* to distinguish them from the low-level operating system thread objects. The SALS garbage collector is a concurrent implementation of a tricolor garbage collection algorithm. Figure 38 compares the speedup gained by executing multiple processes in parallel on the SALS virtual machine to the optimal speedup that could be expected. These tests were executed on a GNU/Linux system. Each parallel process in this experiment is performing the same numerical processing task on memory that is independent of other tasks. The number of computations performed by a single parallel process is used as the unity standard for the speedup comparisons. These tests were

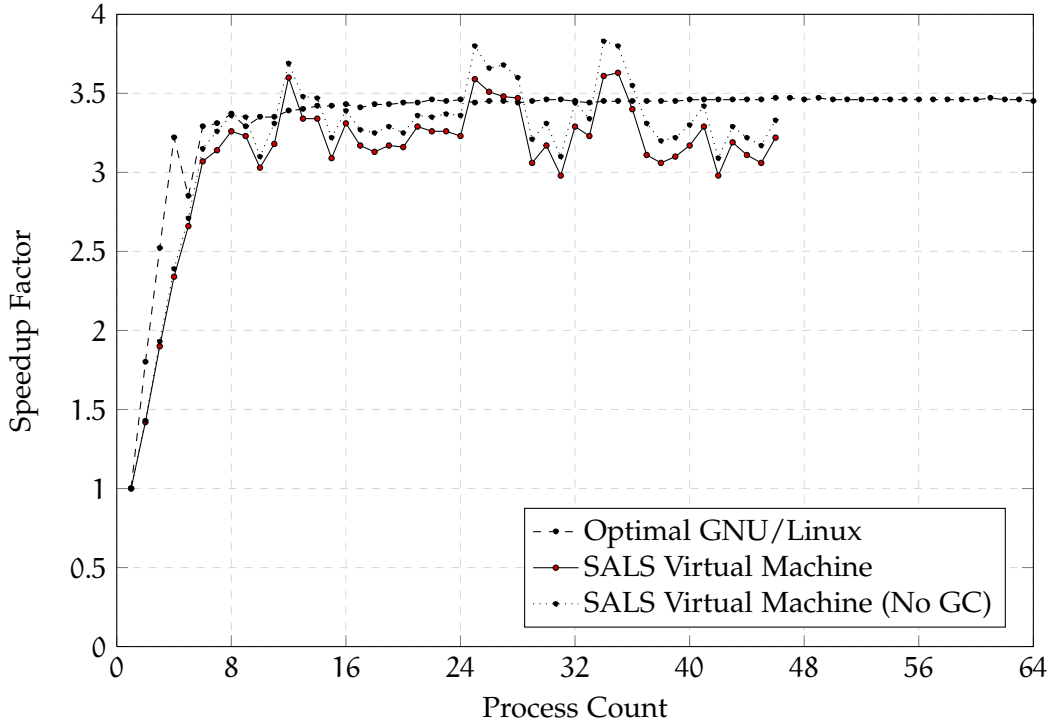


Figure 38: The speedup gained by executing multiple processes concurrently in the SALS virtual machine. The solid line plot represents the speedup experienced by the SALS AI as more fiber processes are executed in parallel. The dashed line plot represents the optimal speedup that can be expected on this specific hardware. Each parallel process in this experiment is performing the same numerical processing task on memory that is independent of other tasks. The number of computations performed by a single parallel process is used as the unity standard for the speedup comparisons. These tests were run on an Intel Core i7 processor with 4 cores, each with 2 hardware hyperthreads. In these tests, 8 virtual processors and 8 memory pools were allocated, one for each hyperthread in the target hardware platform. Each test was performed 100 times to average out variations caused by intermittent garbage collections. The optimal speedup is calculated by a short C program that uses POSIX thread primitives, optimized C arithmetic, and no memory allocation or garbage collection. The SALS AI has a small overhead for each additional parallel process, which is seen by the downward slope of the solid plot. The SALS virtual machine performs most efficiently on this hardware with 7 parallel fibers executing. These tests were executed on a GNU/Linux system.

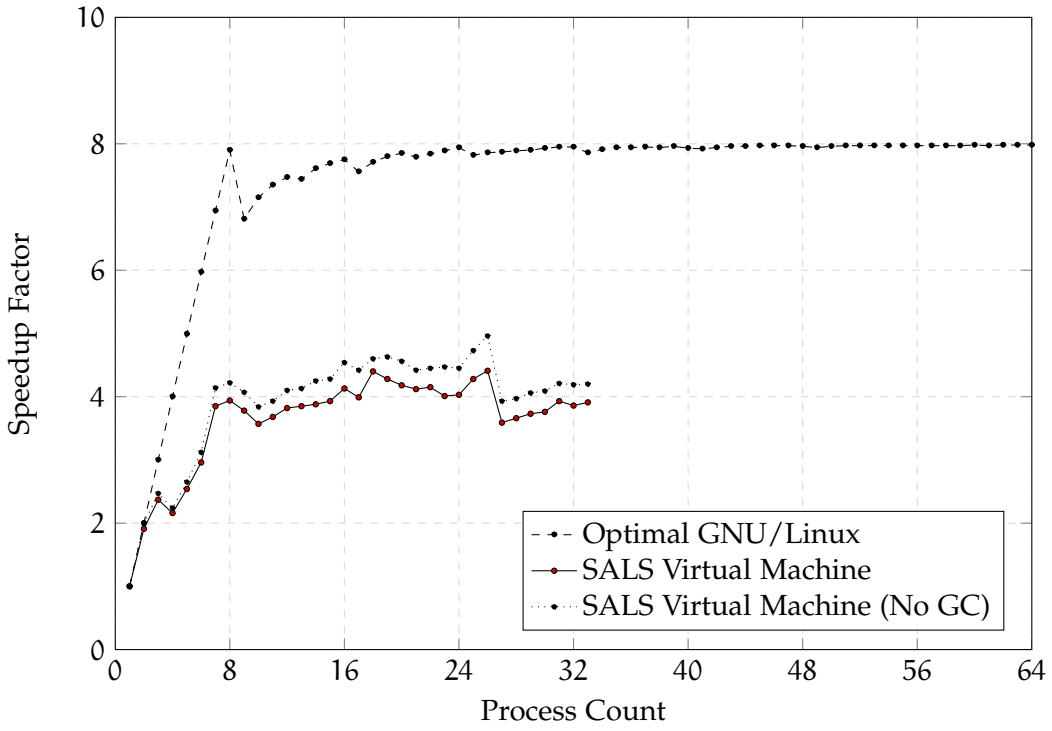


Figure 39: Parallel processing speedup on Dual AMD64 CPUs with 4 cores each.

run on an Intel Core i7 processor with 4 cores, each with 2 hardware hyperthreads. In these tests, 8 virtual processors and 8 memory pools were allocated, one for each hyperthread in the target hardware platform. Each test was performed 100 times to average out variations caused by intermittent garbage collections. The optimal speedup is calculated by a short C program that uses POSIX thread primitives, optimized C arithmetic, and no memory allocation or garbage collection. The SALS AI has a small overhead for each additional parallel process, which is seen by the downward slope of the solid plot.

All of the time-complexities that have been presented in this chapter do not incorporate any speedup that is gained by using concurrent or parallel hardware. There are many aspects of the SALS AI that take advantage of the concurrent processing capabilities of the SALS virtual machine. For example, because the SALS natural language planning language is a simple functional grammar, any ambiguous interpretations are performed in parallel processes in the SALS virtual machine. Also, each planning layer contains two asynchronous learning stages that each contain two parallel agencies that concurrently process streams of procedurally reflective trace events.

8

FUTURE

The primary focus of this thesis is a tractable implementation of a cognitive architecture for the recursive application of reflective layers of planning and learning. Because the focus of this dissertation has been on the larger recursive structure of the SALS cognitive architecture, one layer of which is a closed-loop non-reflective AI, each necessary sub-component of a single planning layer in the SALS AI has been kept relatively simple. Therefore, each of these components of the current skeleton of the SALS AI can be fleshed out in the future, while keeping the core tractable cognitive architecture for a many-layered reflective AI. The following are four areas of research that would improve every layer in the SALS AI:

1. General partial state abstraction. The ability to abstract more general partial states from a control domain would allow the addition of *critics* that recognize relatively specific and intricate problematic partial states in the control domain, as is done in the EM-ONE architecture (Singh 2005), the precedent for this work.
2. A propagation model of knowledge maintenance that traces knowledge provenance from refined hypotheses learned from experience to plan goal accomplishment and avoidance hypotheses (Radul & Sussman 2009). Currently, the SALS AI must completely reimagine the effects of a plan when new hypotheses are learned from experience.
3. General grammar definitions in the natural language programming language, such as those presented by Winograd (1970). Currently, the SALS natural programming language is limited to a simple functional grammar.
4. More efficient usage of multicore, hyperthreaded, and multiple-processor hardware platforms as well as completion of the distributed processing model that is partially implemented in the memory layer of the SALS virtual machine, which would allow peer-to-peer distributed processor support.

In addition to making improvements to the existent structure of the SALS AI, there are a number of aspects missing from the SALS AI that could be added to the architecture to better model human commonsense

thinking. A few selected ideas for new additions to the SALS AI are as follows:

1. Self-reflective and self-consciously reflective thinking, the top two layers of the Emotion Machine theory of human commonsense thinking (Minsky 2006), including self-models and social knowledge.
2. Extending reflective planning layers downward into the domain of plans-to-perceive (Pryor et al. 1992, Pryor & Collins 1995, Velez et al. 2011), so that the SALS AI can learn plans for abstracting spatial relations between symbolic perceptions, such as the visual routines first implemented by Ullman (1984).
3. Plan and meta-plan recognition as a way to include current research on reflective story understanding architectures that attempt to explain the actions of actors in terms of goals and meta-goals that the actor may be trying to accomplish (Wilensky 1981, Cox & Ram 1999, Cox & Raja 2010, Winston 2011).

8.1 GENERAL PARTIAL STATE ABSTRACTION

The SALS AI is currently restricted in the types of partial states that it can abstract from the knowledge base that it is trying to control. The two types of partial states that are included in the current implementation are the "`relationship`" and "`property`" types that have been described in chapter 3. Implementing more general types of partial state abstraction in the SALS AI must be done carefully to keep the architecture tractable at higher layers. One goal for implementing perceptual partial state abstractions of greater complexity would be to allow for implementing critics from EM-ONE (Singh 2005). Putting efficiency aside for the moment, implementing new types of partial state abstractions for the SALS AI is a straightforward task. There are two places in each planning layer where partial state abstraction processes are executed:

1. *Asynchronous Procedural Event Stream Abstraction*: The asynchronous learning algorithm that exists in each reflective layer abstracts partial states from a historical reconstructed copy of the knowledge base that the planning layer is trying to control.
2. *Direct Real-Time Read Abstraction*: An executing plan may execute a procedure that checks to see if a specific type of partial state currently exists in the knowledge base that it is trying to control.

Asynchronous learning from procedural event streams requires that when a planning layer receives a change event from the knowledge base

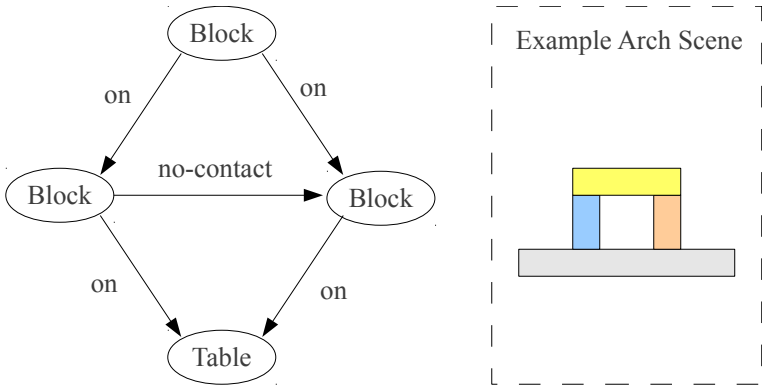


Figure 40: An example of a more complex type of partial state than the SALS AI can currently abstract, which represents an arch.

in the layer below, the planning layer integrates this change into a reconstructed copy of that knowledge base. The reconstructed knowledge base represents the state of the knowledge base that the planning layer is trying to control at a point in the past. The asynchronous abstraction process works on this reconstruction to determine whether or not the change has either created a new instance of a type of partial state or has removed a previously existing instance. The asynchronous procedural event stream abstraction of partial state types works to learn causal hypotheses for the effects of resource executions in the layer below. When a plan is currently executing in the planning layer, it is sometimes useful to determine whether or not a specific partial state currently exists in the knowledge base that the planning layer is trying to control. The asynchronous partial state abstraction cannot be used for this type of real-time query because the asynchronous abstraction algorithm works on an historical copy of the knowledge in question. Therefore, the direct real-time read abstraction of a specific partial state from the current state of the knowledge base is sometimes necessary for this purpose.

To demonstrate the difference in time-complexity between these two methods of abstracting types of partial states, let us consider that a partial state is a specific subgraph that is to be symbolically reified both asynchronously as well as by direct read access from a graph representation of the frame-based knowledge base that the planning layer is trying to control. Figure 40 shows an example of a more complex type of partial state than the SALS AI can currently abstract, which represents an "arch," similar to the arch category learned by near-miss learning by Winston (1970). As these types of partial states become more complex, the direct real-time read approach to abstraction becomes very

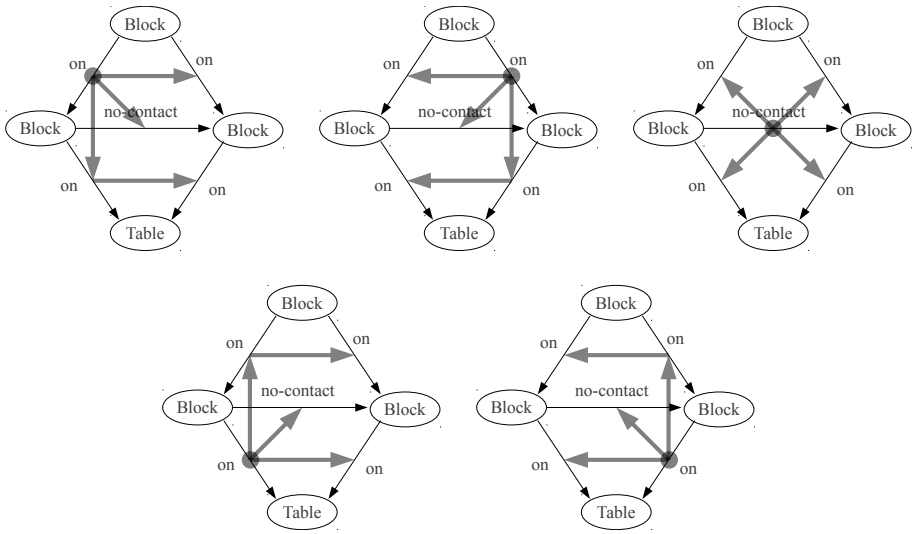


Figure 41: Five minimum spanning trees between the edges of the arch example partial state. One minimum spanning tree is shown for each edge in the partial state of interest. Precomputing one minimum spanning tree rooted at each edge in the partial state of interest reduces the general NP-complete graph isomorphism decision problem to a rooted tree matching problem.

similar to the NP-complete subgraph isomorphism decision problem, which quickly becomes intractable for large subgraphs.[1]

Consider how the asynchronous abstraction of subgraphs from change event streams can improve upon the intractable subgraph isomorphism problem. The key to realize is that the change event focuses the task of the asynchronous abstraction process, which changes the general graph isomorphism problem into a focused tree matching problem, given that spanning trees are precomputed for each edge in the partial state graph of interest. Figure 41 shows an example of five minimum spanning trees, one for each edge of the arch example partial state. Computing a minimum spanning tree for a connected graph has polynomial (FP) time-complexity. Precomputing one minimum spanning tree rooted at each edge in the partial state of interest reduces the general NP-complete graph isomorphism decision problem to a rooted tree matching problem.

1 Given a set of subgraphs that are of interest, preprocessing these subgraphs can lead to a practically more efficient hierarchical "graph stitching" algorithm for finding which of this set of subgraphs exist within a target graph (Messmer 1995, Messmer & Bunke 2000). For large enough graphs, however, these algorithms are still intractable for real-time queries. These algorithms have been implemented and are included in the SALS AI, but are not used for the examples in this dissertation.

While there may be a more efficient solution to the asynchronous abstraction of partial states in the SALS AI, the problem of real-time abstraction is still an NP-complete problem. I do not see any easy solutions to this problem, but there are two ways that I see to avoid it. One way to avoid the problem is to limit the real-time partial state abstraction to a predefined set of small graphs that only involve 5 to 10 nodes. Messmer (1995) describes an efficient algorithm for this type of query. Note that this type of query would still allow the asynchronous learning of causal models from partial states with precomputed minimum spanning trees rooted at change events. Another way to avoid the NP-complete problem of real-time abstraction of partial states is to ignore it. If the asynchronous abstraction of partial states can run quickly enough, perhaps by allocating a parallel processor and a separate reconstructed knowledge base to each type of partial state, the asynchronous algorithm could potentially show little enough latency to make it near enough to real-time to be practically useful.

8.2 PROPAGATION BY PROVENANCE OF HYPOTHETICAL KNOWLEDGE

Currently in the SALS AI, when a resource is executed by a planning layer, generalized hypotheses are learned that are used to predict the effects of those resource executions in the future. For example, these hypotheses are used to predict the effects of a plan when that plan is imagined being executed as part of the planning process in a planning layer. When a plan is imagined in this way, these effects are associated with the plan object in the plan knowledge base of that layer. Subsequently, when the hypotheses that support these conclusions are refined due to new experience gained through the execution of similar plans or actions, the hypothetical support for these original conclusions may be lost. The problem is that these original conclusions remain associated with the plan objects, even in the case when the hypothetical support for those conclusions is lost through the refinement of the hypothesis spaces from which they were originally derived. Because of the potential loss of hypothetical support for the effects of plans, the plan must be completely re-imagined whenever the asynchronous learning algorithms have learned new information through the experiential event streams.

Radul & Sussman (2009) explain a type of Truth Maintenance System (TMS) (Doyle 1978) called a *propagator*. A propagator network is composed of *cells* that represent variables in a computational process that can have multiple ambiguous values. The propagator network also has the ability to track the provenance of the derivations for the values

stored in these cells. The advantage of TMSs and propagator networks is the ability to perform dependency directed backtracking to find new justifications for beliefs when contradictions in knowledge are encountered. If the knowledge derivations in the SALS AI were built based upon a propagator network, the contradictions between hypothetical assumptions and new experiential input could be propagated efficiently to the effects that are associated with plan objects. One advantage of propagator networks over traditional TMS systems is their ability to maintain multiple possibly contradictory sets of beliefs that are derived from different sources of information. The ability to maintain multiple, possibly conflicting belief structures in a propagator network would allow the SALS AI to consider the source of the knowledge and develop ways of reasoning that determine when one source is more reliable than another. For example, when the SALS AI learns new plans from being told, the source of this knowledge could be associated with different users or other specific AIs that are the source of these new assertions. When one source is found to be valid for one domain but invalid for another, the SALS AI could use the features of a propagator network to easily switch between reasoning using different sets of knowledge sources in different circumstances.

8.3 GENERAL NATURAL LANGUAGE GRAMMAR

The SALS AI currently interprets natural language plans, while simultaneously considering syntax, semantics, current environmental context, learned hypothetical knowledge about the effects of actions as well as the current positive and negative goals of the AI. The SALS natural programming language is currently limited to a simple functional grammar. One problem with many natural language grammars are that they usually assume a specific language, such as English, as the basis for their grammar. Although the SALS AI grammar is simple, the benefit of this simplicity is that because it is based on a Unicode string matching template, it is able to represent and reason with natural language phrases from any natural human language, including languages like Chinese and Japanese, which do not have words separated by spaces. While there are many ways to improve the SALS natural language interpreter, the following are four that are immediately applicable to the SALS AI:

1. *Mutation, Side-Effects and Causal Variables*: One immediate way to maintain the generality of the SALS natural language interpreter, while adding the complexities of a general natural language grammar, is to allow the language to have mutation side-effects, so that

the interpreter can be stateful. Adding mutation side-effects to the SALS natural language interpreter introduces a complexity to the current interpretation search process that would become intractable without an explicit additional method of curbing this complexity. Dependency tracing would be a promising way to reign in the complexity of this search process.

2. *One Consolidated Natural Language Interface*: The SALS AI currently receives natural language plans through a separate interface to each planning layer. One future goal for the architecture is to have one natural language input to the entire architecture that enters the architecture through the lowest-level perceptual knowledge bases of the built-in reactive layer and propagates upward through the planning layers until the natural language phrase is completely understood.
3. *Incremental Natural Language Understanding*: To increase the realism of the SALS natural language interpreter, the ability of the interpreter to receive and process a stream of phonetic or character-level symbols in temporal order, rather than in template form, could allow the SALS AI to begin imagining multiple possible interpretations of a statement or phrase before that statement or phrase has been completely communicated to the SALS AI.
4. *Natural Language Analogies Between Layers*: The inherent ambiguity of natural language plans allows analogies to be made between plans within any given planning layer. An exciting avenue for future research is the creation of analogies between planning layers in the SALS AI. For example, consider that the AI has learned a plan for stacking blocks in the Blocks World domain that involves first dividing the blocks into two groups on the table: (1) cubes and (2) pyramids. The AI can execute a simple block stacking plan that simply selects blocks from the group with only cubes in order to create a stack without again considering the shapes of the blocks in question. This plan for physical action, could provide an analogy for the reflective thinking layer if the AI confronts a problem that requires plans to be grouped into two lists: (1) those plans that activate resources concurrently, and (2) those plans that execute resources sequentially. This could be a way to plan quickly without imagining potential resource conflicts in plans with concurrent resource executions. Finding analogies between natural language plans of different layers in the SALS AI could lead to faster learning in new domains of reflective thinking, for

example, if the AI knew many plans for physical action and few for reflective action.

Natural language plans are an important part of a cognitive architecture because of the potential for the AI to be able to explain its own reasoning processes in terms of this natural language understanding. There are many directions for extending the use of natural language in the SALS AI.

8.4 SELF-REFLECTIVE THINKING

The *self-reflective* and *self-conscious* layers are the top two layers of the Emotion Machine theory of human commonsense thinking (Minsky 2006). These two layers are not included in the current version of the SALS AI. The self-reflective layer includes plans that control the planning resources in the lower reflective layers. The self-reflective layer contains *self-models* which represent large-scale reconfigurations of the thinking resources in the lower reflective layers of the AI. These global changes to the way an AI is thinking are sometimes referred to as "emotions" or "personalities" depending on the time-scale. My usage of these terms is analogous to the usage of the terms "weather" and "climate." A personality is a longer-term plan involving shorter-term emotion actions. The following is a list of some simple emotion and personality plan definitions that could be added to a self-reflective layer of the SALS AI:

- Be happy: Do nothing.
- Be tired: Make a plan to go to sleep.
- Sleep: Imagine executing recently used plans in a variety of contexts.
- Be frustrated: Try to find goals to give up that are not worth accomplishing.
- Grieve: Replace old plans and goals that depend upon lost resources.
- Be athletic: Get better at making plans that require me to exercise.
- Be friendly: Get better at making plans to help friends accomplish their goals.

The self-reflective layer's plan knowledge base could include the following types of self-model knowledge, which would be reflections of the above plans being executed:

- I am happy.

- I am tired.
- I am frustrated.
- I am grieving.
- I am athletic.
- I am friendly.

One distinguishing factor of the self-reflective layer is that it not only includes self-models but also analogous *other-models*. The following is a list of other-model knowledge:

- Joe is happy.
- Carol is tired.
- Carol is frustrated.
- Suzy is grieving.
- Ralph is athletic.
- Lauren is friendly.

The self-reflective layer has the capability to abstractly simulate self-models and other-models in terms of all of the partial states in the layers below. This simulation can probably be performed tractably because only a linear increase in complexity has been seen to be added to the SALS overall architecture with each additional layer in the current architecture.

One way to simulate other-models and self-models analogously is by adding an ontology to the SALS AI's knowledge bases. Currently the SALS AI can only abstract and simulate symbolically reified partial states that represent the problem domain. These collections of symbols that are the input to each planning layer of the SALS AI are not *object-oriented*. Meaning that they are not organized into frame-based object representations. Learning an ontology of object types is future research for the SALS AI. The ability to abstract and simulate object types may be related to learning object binding affordances (Stoytchev 2005). Also, learning these object types may lead to an object model that could naturally be extended for learning self-models and other-models.

8.5 SELF-CONSCIOUS THINKING

The *self-conscious* layer is the top layer of the Emotion Machine theory of human commonsense thinking. The self-conscious layer is a planning layer that controls the self-reflective layer below. Self-conscious thinking involves thinking about what one person is thinking about

another person. The representations in the self-conscious thinking layer include relationships between self-models and other-models from the self-reflective layer. The following are examples of self-conscious knowledge:

- Joe is imagining that Lauren is happy.
- Ralph is unaware that Suzy is grieving.
- Carol imagines that Ralph knows a plan to stack two blocks.
- Lauren knows that I failed to stack two blocks.
- Ralph and Lauren are Suzy's children.

The self-conscious thinking layer is the first reflective layer in the Emotion Machine theory to include reasoning about the roles of individuals in the context of social groups. Minsky (2006) explains one of these types of relationships that allows the direct inheritance of high-level goals from one agent to another agent that he calls the *imprimer* relationship. An imprimer functions in a developmental situation where a learner inherits high-level goals from their imprimer. A parent or caregiver in the context of raising a child might be seen as imprimers to the developing child that teach the child what kinds of social roles, personality traits, and other ways of thinking should be practiced or ignored.

8.6 PLANNING TO PERCEIVE

Section 6.2 describes the *Architecture for Visual Routines (AVR)* (Rao 1998) that is based on an idea originally proposed by Ullman (1984). To briefly review, *visual routines* define objects and parts by having a visual plan interpreter that is focused on a specific part of the low-level visual input at any given point in time. Although the first planning layer in the SALS AI is the deliberative layer, one can imagine that a solution to including a perceptual planning layer would be to extend the SALS planning layers into the learned reactive layer of the SALS AI, so that the translation of visual knowledge from the built-in reactive visual knowledge base to the physical knowledge base in the learned reactive layer would be performed by a planning layer below the deliberative layer. Thus, the technique of extending the SALS AI's planning layers to higher layers of super-reflective control could be just as easily reversed by extending the SALS AI's planning layers to lower layers of planned control, so that reflective control could be applied to the types of planning-to-perceive problems as a coherent integration of Ullman's visual routines into deliberative reasoning layers of reflective control.

8.7 PLAN RECOGNITION AND STORY UNDERSTANDING

section 6.2 has described the relationship between the planning implemented in the SALS AI and the related problems of plan recognition, meta-plan recognition, and story understanding. To briefly review, Wilensky (1981) describes story understanding as a form of inverse planning or plan recognition. In this sense, a planning system is given a set of goals that are used to generate a sequence of actions that accomplish the goals, while a story understanding system is given a sequence of actions that are used to generate a set of goals that explain those actions. Wilensky emphasizes that both story understanding and planning require representations of meta-goals and meta-plans to reason about common types of human thinking. Plan recognition could occur analogously in each layer of the SALS AI. Resource executions are the actions of each planning layer. The plan recognition problem for the deliberative layer could be considered to be a problem that takes a collection of resource execution events and gives as output the goals that these actions could be hypothetically trying to accomplish. One method for accomplishing this would be to compare the input traces to the imaginative traces from executing each plan for physical action, then, hypothesize the goals that the plans with matching traces are hypothesized to cause. Currently, the SALS AI is directed more toward planning, so that plans for plan recognition can be reflected upon by the SALS AI. The SALS AI is currently more of a planning system than a plan recognition system for one simple reason: methods for plan recognition can be written in the planning system so that they can be automatically learned about and debugged by the SALS AI.

9

CONCLUSION

Building a model of general human intelligence is a massive engineering effort that includes the shared expertise of all the disciplines of cognitive science, including: philosophy, artificial intelligence, psychology, linguistics, anthropology, and neuroscience. While this dissertation is focused on building an AI that scales to novel forms of metacognition that occur at higher layers of reflective intelligence than have before been modeled, I hope that the future of the SALS architecture enjoys contributions from all of the diverse sub-disciplines of cognitive science. The SALS AI is meant to be a user-friendly platform for encoding many examples of intelligence from each of these disciplines that are usually researched separately. While the current strengths and weaknesses of the SALS AI have been discussed throughout this dissertation, I hope that the following features of the SALS architecture help toward this collaborative end:

1. Simple Syntax.
2. Natural Language Programming Interface.
3. Object-Oriented Frame-Based Representations.
4. Cognitive Architectural Primitives, such as Resources, Agencies, Planning Layers, and Reflectively Traced Knowledge Bases.
5. Causal Organization of Parallel Processes.
6. Concurrent Virtual Machine.
7. Included Machine Learning and Graph Algorithms.
8. Extension Package Manager.
9. On-Line Documentation (Morgan 2012).
10. Open-Source Community.

Given that the current five-layered version of the SALS AI is designed to take advantage of multicore processors with local cache, as computer hardware trends toward CPUs with more cores on each chip with each core being simpler and with more on-chip cache storing memory local to computations, the SALS virtual machine is well placed to benefit from this coming wave of distributed multithreaded hardware platforms. This dissertation has described the four contributions of this thesis:

1. Emotion Machine Cognitive Architecture (chapter 2)
2. Learning from Being Told Natural Language Plans (chapter 3)
3. Learning Asynchronously from Experience (chapter 4)
4. Virtual Machine and Programming Language (chapter 5)

chapter 6 has compared the SALS AI to a number of related contemporary research disciplines, including: computational metacognition, meta-planning and meta-plan recognition, optimality in metacognition, and massively multithreaded programming. chapter 7 evaluates the SALS AI by showing the reflective architecture to scale at a tractable linear increase in time-complexity for each additional layer, which implies an N layered metacognitive architecture that allows not only planning and learning about a problem domain but also any number of reflective layers of planning and learning about the thinking processes themselves. A number of promising directions for future research with the SALS AI have been described in chapter 8:

1. Self-reflective and self-consciously reflective thinking, the top two layers of the Emotion Machine theory of human commonsense thinking (Minsky 2006), including self-models and social knowledge.
2. Extending reflective planning layers downward into the domain of plans-to-perceive (Pryor et al. 1992, Pryor & Collins 1995, Velez et al. 2011), so that the SALS AI can learn plans for abstracting spatial relations between symbolic perceptions, such as the visual routines first implemented by Ullman (1984).
3. Plan and meta-plan recognition as a way to include current research on reflective story understanding architectures.
4. General partial state abstraction processes.
5. A propagation model of knowledge maintenance that traces knowledge provenance from refined hypotheses learned from experience to plan goal accomplishment and avoidance hypotheses.
6. General grammar definitions for the natural language programming language.

7. More efficient usage of multicore, hyperthreaded, and multiple-processor hardware platforms as well as completion of the distributed processing model that is partially implemented in the memory layer of the SALS virtual machine, which would allow peer-to-peer distributed processor support.

For more information about downloading and getting started experimenting with the freely distributed open source implementation of the SALS AI, please see Appendix A.

Part IV

APPENDIX

A

THE CODE

A.1 OPEN SOURCE

All of the code developed for this PhD is open source, free software, and is distributed under the terms of the GNU General Public License. A copy of this program should be available on-line or by contacting the author, Bo Morgan (bo@mit.edu).

A.2 README

```
funk2: causally reflective parallel programming language

  funk2
  funk2 <source.fu2>
  funk2 -x <command>
  funk2 -i <bootstrap-image>

    <source.fu2>

        A user supplied filename of file from which to read and
        execute source code after booting and before exiting.

    -x <command>

        A user supplied command to execute after booting and before
        exiting.

    -i <bootstrap-image>

        The bootstrap image to load before parsing any other
        commands.  This option is useful for scripting Funk2 programs
        without needing to recompile.

    -p <portnum>

        The localhost peer-command-server port number.


TO PERFORM LOCAL BUILD:

  ./configure
  make


TO RUN LOCAL BUILD:

  source funk2-exports
```

```
  bin/funk2      (from original compile directory)


TO PERFORM SYSTEM-WIDE INSTALLATION:

  ./configure
  make
  make install  (as root)


TO RUN SYSTEM-WIDE INSTALLATION:

  funk2          (from anywhere on system)


HOMEPAGE:

  http://funk2.org/


GIT ACCESS:

  https://github.com/bunuelo/funk2


LICENSE:

  GNU General Public License v3


Last Modified: 2013.02.26

Code Mass

-------------------------------------------------------------------------------


    Lines of Funk2 Code.....:    60673 total
    Words of Funk2 Code.....:   294830 total
    Characters of Funk2 Code: 3026879 total

    Lines of C Code.........:   157729 total
    Words of C Code.........:   611577 total
    Characters of C Code....: 8331511 total

    Total Lines of Code.....:   220241 total
    Total Words of Code.....:   912309 total
    Total Characters of Code: 11506475 total

-------------------------------------------------------------------------------


README Last Generated: Tue Feb 26 21:49:25 EST 2013
```

A.3 FILE LISTING

```
built-in/alien/alien.fpkg
built-in/alien/alien.fu2
built-in/ansi/ansi.fpkg
built-in/ansi/primfunks-ansi.fu2
built-in/basic_bug_responses/basic_bug_responses.fpkg
built-in/basic_bug_responses/basic_bug_responses.fu2
built-in/graph_cluster/bootstrap-graph_cluster.fu2
built-in/graph_cluster/graph_cluster.fpkg
built-in/graph_cluster/primfunks-graph_cluster.fu2
built-in/graph_match_error_correcting/graph_match_error_correcting.fpkg
built-in/graph_match_error_correcting/graph_match_error_correcting.fu2
built-in/graph_match_error_correcting/graph_match_error_correcting-
    primfunks.fu2
built-in/graphviz/graphviz.fpkg
built-in/graphviz/graphviz.fu2
built-in/graphviz/graphviz-primfunks.fu2
built-in/math/math.fpkg
built-in/math/math.fu2
built-in/mutex/mutex.fpkg
built-in/mutex/mutex.fu2
built-in/natural_language/dictionary_frame.fu2
built-in/natural_language/natural_language_command.fu2
built-in/natural_language/natural_language.fpkg
built-in/natural_language/natural_language-primfunks.fu2
built-in/natural_language/parse_tree.fu2
built-in/natural_language/skb-test.fu2
built-in/number/bootstrap-number.fu2
built-in/number/number.fpkg
built-in/number/primfunks-number.fu2
built-in/utilities/errno.fu2
built-in/utilities/fcntl.fu2
built-in/utilities/ioctl.fu2
built-in/utilities/socket.fu2
built-in/utilities/utilities.fpkg
built-in/xmlrpc/bootstrap-xmlrpc.fu2
built-in/xmlrpc/primfunks-xmlrpc.fu2
built-in/xmlrpc/xmlrpc.fpkg
c/configurator.c
c/debugbreak.c
c/f2_ansi.c
c/f2_ansi.h
c/f2_apropos.c
c/f2_apropos.h
c/f2_archconfig.h
c/f2_array.c
c/f2_array.h
c/f2_atomic.h
c/f2_buffered_file.c
c/f2_buffered_file.h
c/f2_buffered_socket.c
c/f2_buffered_socket.h
c/f2_bug.c
c/f2_bug.h
c/f2_bytecodes.c
c/f2_bytecodes.h
c/f2_cause.c
```

```
c/f2_cause.h
c/f2_chunk.c
c/f2_chunk.h
c/f2_circular_buffer.c
c/f2_circular_buffer.h
c/f2_command_line.c
c/f2_command_line.h
c/f2_compile.c
c/f2_compile.h
c/f2_compile_x86.c
c/f2_compile_x86.h
c/f2_core_extension.c
c/f2_core_extension_funk.c
c/f2_core_extension_funk.h
c/f2_core_extension.h
c/f2_cpu.c
c/f2_cpu.h
c/f2_debug_macros.h
c/f2_defragmenter.c
c/f2_defragmenter.h
c/f2_dlfcn.c
c/f2_dlfcn.h
c/f2_dptr.c
c/f2_dptr.h
c/f2_dynamic_memory.c
c/f2_dynamic_memory.h
c/f2_f2ptr_set.c
c/f2_f2ptr_set.h
c/f2_fiber.c
c/f2_fiber.h
c/f2_fileio.c
c/f2_fileio.h
c/f2_frame_objects.c
c/f2_frame_objects.h
c/f2_funk2_node.c
c/f2_funk2_node.h
c/f2_funktional.c
c/f2_funktional.h
c/f2_garbage_collector_block_header.c
c/f2_garbage_collector_block_header.h
c/f2_garbage_collector.c
c/f2_garbage_collector.h
c/f2_garbage_collector_pool.c
c/f2_garbage_collector_pool.h
c/f2_globalenv.c
c/f2_globalenv.h
c/f2_global.h
c/f2_glwindow.c
c/f2_glwindow.h
c/f2_gmodule.c
c/f2_gmodule.h
c/f2_graph.c
c/f2_graph_cluster.c
c/f2_graph_cluster.h
c/f2_graph.h
c/f2_graph_match_error_correcting.c
c/f2_graph_match_error_correcting.h
c/f2_graphviz.c
c/f2_graphviz.h
```

```
c/f2_hash.c
c/f2_hash.h
c/f2_heap.c
c/f2_heap.h
c/f2_html.c
c/f2_html.h
c/f2_larva.c
c/f2_larva.h
c/f2_load.c
c/f2_load.h
c/f2_malloc.c
c/f2_malloc.h
c/f2_management_thread.c
c/f2_management_thread.h
c/f2_math.c
c/f2_math.h
c/f2_matlab.c
c/f2_matlab.h
c/f2_memblock.c
c/f2_memblock.h
c/f2_memory.c
c/f2_memory.h
c/f2_memorypool.c
c/f2_memorypool.h
c/f2_module_registration.c
c/f2_module_registration.h
c/f2_natural_language.c
c/f2_natural_language.h
c/f2_never_delete_list.c
c/f2_never_delete_list.h
c/f2_nil.c
c/f2_nil.h
c/f2_number.c
c/f2_number.h
c/f2_object.c
c/f2_object.h
c/f2_opengl.c
c/f2_opengl.h
c/f2_optimize.c
c/f2_optimize.h
c/f2_os.h
c/f2_package.c
c/f2_package.h
c/f2_package_handler.c
c/f2_package_handler.h
c/f2_packet.c
c/f2_packet.h
c/f2_partial_order.c
c/f2_partial_order.h
c/f2_peer_command_server.c
c/f2_peer_command_server.h
c/f2_primes.c
c/f2_primes.h
c/f2_primfunks.c
c/f2_primfunks__errno.c
c/f2_primfunks__errno.h
c/f2_primfunks__fcntl.c
c/f2_primfunks__fcntl.h
c/f2_primfunks.h
```

```
c/f2_primfunks__ioctl.c
c/f2_primfunks__ioctl.h
c/f2_primfunks__locale.c
c/f2_primfunks__locale.h
c/f2_primfunks__stdlib.c
c/f2_primfunks__stdlib.h
c/f2_primmetros.c
c/f2_primmetros.h
c/f2_primobject__boolean.c
c/f2_primobject__boolean.h
c/f2_primobject__char_pointer.c
c/f2_primobject__char_pointer.h
c/f2_primobject__circular_buffer.c
c/f2_primobject__circular_buffer.h
c/f2_primobject__counter.c
c/f2_primobject__counter.h
c/f2_primobject__doublelinklist.c
c/f2_primobject__doublelinklist.h
c/f2_primobject__dynamic_library.c
c/f2_primobject__dynamic_library.h
c/f2_primobject__environment.c
c/f2_primobject__environment.h
c/f2_primobject__fiber_trigger.c
c/f2_primobject__fiber_trigger.h
c/f2_primobject__file_handle.c
c/f2_primobject__file_handle.h
c/f2_primobject__frame.c
c/f2_primobject__frame.h
c/f2_primobject__hash.c
c/f2_primobject__hash.h
c/f2_primobject__largeinteger.c
c/f2_primobject__largeinteger.h
c/f2_primobject__list.c
c/f2_primobject__list.h
c/f2_primobject__matrix.c
c/f2_primobject__matrix.h
c/f2_primobject__object.c
c/f2_primobject__object.h
c/f2_primobject__object_type.c
c/f2_primobject__object_type.h
c/f2_primobject__ptypehash.c
c/f2_primobject__ptypehash.h
c/f2_primobject__redblacktree.c
c/f2_primobject__redblacktree.h
c/f2_primobjects.c
c/f2_primobject__scheduler_ptypehash.c
c/f2_primobject__scheduler_ptypehash.h
c/f2_primobject__set.c
c/f2_primobject__set.h
c/f2_primobjects.h
c/f2_primobject__stream.c
c/f2_primobject__stream.h
c/f2_primobject__tensor.c
c/f2_primobject__tensor.h
c/f2_primobject__traced_cmutex.c
c/f2_primobject__traced_cmutex.h
c/f2_primobject_type.c
c/f2_primobject_type.h
c/f2_primobject_type_handler.c
```

```
c/f2_primobject_type_handler.h
c/f2_print.c
c/f2_print.h
c/f2_processor.c
c/f2_processor.h
c/f2_processor_mutex.c
c/f2_processor_mutex.h
c/f2_processor_readwritelock.c
c/f2_processor_readwritelock.h
c/f2_processor_spinlock.c
c/f2_processor_spinlock.h
c/f2_processor_thread.c
c/f2_processor_thread.h
c/f2_processor_thread_handler.c
c/f2_processor_thread_handler.h
c/f2_protected_alloc_array.c
c/f2_protected_alloc_array.h
c/f2_ptype.c
c/f2_ptype.h
c/f2_ptypes.c
c/f2_ptypes.h
c/f2_ptypes_memory.h
c/f2_ptypes_object_slots.c
c/f2_ptypes_object_slots.h
c/f2_reader.c
c/f2_reader.h
c/f2_redblacktree.c
c/f2_redblacktree.h
c/f2_reflective_memory.c
c/f2_scheduler.c
c/f2_scheduler.h
c/f2_scheduler_thread_controller.c
c/f2_scheduler_thread_controller.h
c/f2_set.c
c/f2_set.h
c/f2_signal.c
c/f2_signal.h
c/f2_simple_repl.c
c/f2_simple_repl.h
c/f2_socket.c
c/f2_socket_client.c
c/f2_socket_client.h
c/f2_socket.h
c/f2_socket_server.c
c/f2_socket_server.h
c/f2_sort.c
c/f2_sort.h
c/f2_staticmemory.c
c/f2_staticmemory.h
c/f2_status.c
c/f2_status.h
c/f2_string.c
c/f2_string.h
c/f2_surrogate_parent.c
c/f2_surrogate_parent.h
c/f2_system_file_handler.c
c/f2_system_file_handler.h
c/f2_system_headers.h
c/f2_system_processor.c
```

```
c/f2_system_processor.h
c/f2_terminal_print.c
c/f2_terminal_print.h
c/f2_termios.c
c/f2_termios.h
c/f2_time.c
c/f2_time.h
c/f2_trace.c
c/f2_trace.h
c/f2_tricolor_set.c
c/f2_tricolor_set.h
c/f2_user_thread_controller.c
c/f2_user_thread_controller.h
c/f2_virtual_processor.c
c/f2_virtual_processor.h
c/f2_virtual_processor_handler.c
c/f2_virtual_processor_handler.h
c/f2_virtual_processor_thread.c
c/f2_virtual_processor_thread.h
c/f2_xmlrpc.c
c/f2_xmlrpc.h
c/f2_zlib.c
c/f2_zlib.h
c/funk2.c
c/funk2.h
c/funk2_main.c
c/test.c
documentation/phdthesis/gfx/deliberative_plan_knowledge_base_graph.fu2
documentation/phdthesis/gfx/hierarchical_partial_state_graph.fu2
documentation/phdthesis/gfx/physical_knowledge_base_graph.fu2
documentation/phdthesis/gfx/plan_execution_node_graph.fu2
documentation/phdthesis/gfx/property_partial_state_graph.fu2
documentation/phdthesis/gfx/reflective_plan_knowledge_base_graph.fu2
documentation/phdthesis/gfx/relationship_partial_state_graph.fu2
example/cannons_algorithm/cannons_algorithm.fpkg
example/cannons_algorithm/cannons_algorithm.fu2
example/divisi2/divisi2.fpkg
example/divisi2/divisi2.fu2
example/em_two_webpage/em_two_webpage.fpkg
example/em_two_webpage/em_two_webpage.fu2
example/english_language/english_dictionary.fu2
example/english_language/english_dictionary_parse.fu2
example/english_language/english_language.fpkg
example/facebook/facebook.fpkg
example/facebook/facebook.fu2
example/funk2-htmldoc/funk2-htmldoc.fpkg
example/funk2-htmldoc/funk2-htmldoc.fu2
example/funk2-webpage/funk2-webpage.fpkg
example/funk2-webpage/funk2-webpage.fu2
example/graph_match/graph_match.fpkg
example/graph_match/graph_match.fu2
example/graph_match/graph_match-test.fpkg
example/graph_match/graph_match-test.fu2
example/gtk_timeline/gtk_timeline.fpkg
example/gtk_timeline/gtk_timeline.fu2
example/isis_world_client/isis_world_client.fpkg
example/isis_world_client/isis_world_client.fu2
example/isis_world/isis_agent_body.fu2
example/isis_world/isis_visual_agent.fu2
```

```
example/isis_world/isis_visual_object.fu2
example/isis_world/isis_world.fpkg
example/isis_world/isis_world.fu2
example/macbeth/macbeth.fpkg
example/macbeth/macbeth.fu2
example/mind/agency.fu2
example/mind/agent_body.fu2
example/mind/character.fu2
example/mind/mental_layer.fu2
example/mind/mind.fpkg
example/mind/mind.fu2
example/mind/mind_runtime_metric.fu2
example/mindmon-1.0/mindmon-1.0.fpkg
example/mindmon-1.0/mindmon-1.0.fu2
example/mindmon-blocks_world/mindmon-blocks_world-deliberative_action_
    activator.fu2
example/mindmon-blocks_world/mindmon-blocks_world-deliberative_plan_
    activator.fu2
example/mindmon-blocks_world/mindmon-blocks_world.fpkg
example/mindmon-blocks_world/mindmon-blocks_world.fu2
example/mindmon-blocks_world/mindmon-blocks_world-reactive_action_
    activator.fu2
example/mindmon-blocks_world/mindmon-blocks_world-reactive_plan_
    activator.fu2
example/mindmon-blocks_world/mindmon-blocks_world-reflective_action_
    activator.fu2
example/mindmon-blocks_world/mindmon-blocks_world-super_reflective_
    action_activator.fu2
example/mindmon-isis_world/mindmon-isis_world-builtin_reactive_physical
    _activator.fu2
example/mindmon-isis_world/mindmon-isis_world-deliberative_goal_
    activator.fu2
example/mindmon-isis_world/mindmon-isis_world.fpkg
example/mindmon-isis_world/mindmon-isis_world.fu2
example/mindmon-isis_world/mindmon-isis_world-learned_reactive_physical
    _activator.fu2
example/mindmon/mindmon_agent.fu2
example/mindmon/mindmon_agent_tool.fu2
example/mindmon/mindmon_agent_tool_widget.fu2
example/mindmon/mindmon_agent_widget.fu2
example/mindmon/mindmon_file.fu2
example/mindmon/mindmon.fpkg
example/mindmon/mindmon.fu2
example/mindmon/mindmon_knowledge.fu2
example/mindmon/mindmon_preference_editor.fu2
example/mindmon/mindmon_world.fu2
example/mind/physical_world.fu2
example/mind/resource.fu2
example/mind/self_model.fu2
example/mind/story.fu2
example/mind/story-graph.fu2
example/mind/vital_resource.fu2
example/moral_compass-isis_world/moral_compass-isis_world-builtin_
    reactive_physical_agency_resources.fu2
example/moral_compass-isis_world/moral_compass-isis_world.fpkg
example/moral_compass-isis_world/moral_compass-isis_world.fu2
example/moral_compass-isis_world/moral_compass-isis_world-learned_
    reactive_physical_agency_resources.fu2
```

```
example/moral_compass-isis_world/moral_compass-isis_world-learned_
    reactive_physical_agency_resources-functions.fu2
example/moral_compass/moral_agent_body.fu2
example/moral_compass/moral_compass.fpkg
example/moral_compass/moral_compass.fu2
example/moral_compass/self_conscious_imprimer_agency.fu2
example/moral_compass/self_conscious_layer.fu2
example/moral_compass/self_conscious_resource.fu2
example/moral_compass/self_reflective_layer.fu2
example/moral_compass/self_reflective_other_agents_knowledge_agency.fu2
example/moral_compass/self_reflective_resource.fu2
example/rct_webpage/rct_webpage.fpkg
example/rct_webpage/rct_webpage.fu2
example/reflective_mind-blocks_world/reflective_mind-blocks_world-
    builtin_reactive_physical_agency_resources.fu2
example/reflective_mind-blocks_world/reflective_mind-blocks_world.fpkg
example/reflective_mind-blocks_world/reflective_mind-blocks_world.fu2
example/reflective_mind-blocks_world/reflective_mind-blocks_world-
    learned_reactive_physical_agency_resources.fu2
example/reflective_mind/builtin_reactive_layer.fu2
example/reflective_mind/builtin_reactive_neural_plug_agency.fu2
example/reflective_mind/builtin_reactive_physical_agency.fu2
example/reflective_mind/builtin_reactive_resource.fu2
example/reflective_mind/builtin_reactive_sensory_agency.fu2
example/reflective_mind/controllable_object.fu2
example/reflective_mind/deliberative1_type_property_relation_goal.fu2
example/reflective_mind/deliberative_action.fu2
example/reflective_mind/deliberative_layer.fu2
example/reflective_mind/deliberative_resource.fu2
example/reflective_mind/learned_reactive_language_agency.fu2
example/reflective_mind/learned_reactive_layer.fu2
example/reflective_mind/learned_reactive_physical_agency.fu2
example/reflective_mind/learned_reactive_physical_knowledge_agency.fu2
example/reflective_mind/learned_reactive_resource.fu2
example/reflective_mind/learned_reactive_sensory_agency.fu2
example/reflective_mind/nonsemantic_plan.fu2
example/reflective_mind/object_type_property_relation_goal.fu2
example/reflective_mind/physical_type_property_relation_goal.fu2
example/reflective_mind/plan_mental_layer_execution_agency.fu2
example/reflective_mind/plan_mental_layer.fu2
example/reflective_mind/plan_mental_layer_imagination_agency.fu2
example/reflective_mind/plan_mental_layer_object_type_agency.fu2
example/reflective_mind/plan_mental_layer_plan_agency.fu2
example/reflective_mind/plan_mental_layer_resource.fu2
example/reflective_mind/plan_mental_layer_resource_knowledge_agency.fu2
example/reflective_mind/reflective_layer.fu2
example/reflective_mind/reflective_mind.fpkg
example/reflective_mind/reflective_mind.fu2
example/reflective_mind/reflective_mind_perception.fu2
example/reflective_mind/reflective_mind_proprioceptual_object.fu2
example/reflective_mind/reflective_mind_visual_agent.fu2
example/reflective_mind/reflective_mind_visual_object.fu2
example/reflective_mind/reflective_resource.fu2
example/reflective_mind/reflective_timer.fu2
example/reflective_mind/super_reflective_layer.fu2
example/reflective_mind/super_reflective_resource.fu2
example/reflective_timer/reflective_timer.fpkg
example/reflective_timer/reflective_timer.fu2
example/roboverse/roboverse.fpkg
```

```
example/roboverse/roboverse.fu2
example/socket-client/socket-client.fpkg
example/socket-client/socket-client.fu2
example/socket-server/socket-server.fpkg
example/socket-server/socket-server.fu2
example/traced_mind/traced_mind.fpkg
example/traced_mind/traced_mind.fu2
example/traced_mind/traced_resource.fu2
example/visualize/isismon_agent_visualization.fpkg
example/visualize/isismon_agent_visualization.fu2
example/visualize/visualize_test.fpkg
example/visualize/visualize_test.fu2
extension/blocks_world/blocks_world_block.fu2
extension/blocks_world/blocks_world.c
extension/blocks_world/blocks_world-core.fu2
extension/blocks_world/blocks_world.fpkg
extension/blocks_world/blocks_world.fu2
extension/blocks_world/blocks_world_gripper_controller.fu2
extension/blocks_world/blocks_world_gripper.fu2
extension/blocks_world/blocks_world.h
extension/blocks_world/blocks_world_physics.fu2
extension/blocks_world/blocks_world_sprite.fu2
extension/blocks_world/blocks_world_table.fu2
extension/blocks_world/blocks_world_window.fu2
extension/cairo/cairo.c
extension/cairo/cairo-core.fu2
extension/cairo/cairo.fpkg
extension/cairo/cairo.h
extension/conceptnet/conceptnet.c
extension/conceptnet/conceptnet-core.fu2
extension/conceptnet/conceptnet.fpkg
extension/conceptnet/conceptnet.h
extension/concept_version_space/concept_version_space.c
extension/concept_version_space/concept_version_space-core.fu2
extension/concept_version_space/concept_version_space.fpkg
extension/concept_version_space/concept_version_space.h
extension/equals_hash/equals_hash.c
extension/equals_hash/equals_hash-core.fu2
extension/equals_hash/equals_hash.fpkg
extension/equals_hash/equals_hash.h
extension/event_stream/event_stream.c
extension/event_stream/event_stream-core.fu2
extension/event_stream/event_stream.fpkg
extension/event_stream/event_stream.h
extension/fibermon/fibermon1.fu2
extension/fibermon/fibermon.c
extension/fibermon/fibermon-core.fu2
extension/fibermon/fibermon.fpkg
extension/fibermon/fibermon.fu2
extension/forgetful_event_stream/forgetful_event_stream.c
extension/forgetful_event_stream/forgetful_event_stream-core.fu2
extension/forgetful_event_stream/forgetful_event_stream.fpkg
extension/forgetful_event_stream/forgetful_event_stream.h
extension/forgetful_semantic_event_knowledge_base/forgetful_semantic_
    event_knowledge_base.c
extension/forgetful_semantic_event_knowledge_base/forgetful_semantic_
    event_knowledge_base-core.fu2
extension/forgetful_semantic_event_knowledge_base/forgetful_semantic_
    event_knowledge_base.fpkg
```

```
extension/forgetful_semantic_event_knowledge_base/forgetful_semantic_
    event_knowledge_base.h
extension/forgetful_semantic_resource_event_knowledge_base/forgetful_
    semantic_resource_event_knowledge_base.c
extension/forgetful_semantic_resource_event_knowledge_base/forgetful_
    semantic_resource_event_knowledge_base-core.fu2
extension/forgetful_semantic_resource_event_knowledge_base/forgetful_
    semantic_resource_event_knowledge_base.fpkg
extension/forgetful_semantic_resource_event_knowledge_base/forgetful_
    semantic_resource_event_knowledge_base.h
extension/forward_planner/forward_planner.c
extension/forward_planner/forward_planner-core.fu2
extension/forward_planner/forward_planner.fpkg
extension/frame_ball/frame_ball.c
extension/frame_ball/frame_ball-core.fu2
extension/frame_ball/frame_ball.fpkg
extension/graph_isomorphism/graph_isomorphism.c
extension/graph_isomorphism/graph_isomorphism-core.fu2
extension/graph_isomorphism/graph_isomorphism.fpkg
extension/graph_isomorphism/graph_isomorphism.h
extension/gtk_extension/gtk_extension.c
extension/gtk_extension/gtk_extension-core.fu2
extension/gtk_extension/gtk_extension.fpkg
extension/gtk_extension/gtk_extension.h
extension/gtk_extension/old_primfunks.fu2
extension/image/image.c
extension/image/image-core.fu2
extension/image/image.fpkg
extension/image/image.h
extension/image_sequence/image_sequence.c
extension/image_sequence/image_sequence-core.fu2
extension/image_sequence/image_sequence.fpkg
extension/image_sequence/image_sequence.h
extension/interval_tree/interval_tree.c
extension/interval_tree/interval_tree-core.fu2
extension/interval_tree/interval_tree.fpkg
extension/interval_tree/interval_tree.h
extension/keyboard/keyboard.c
extension/keyboard/keyboard-core.fu2
extension/keyboard/keyboard.fpkg
extension/keyboard/keyboard.h
extension/keyboard/keyboard-repl.fu2
extension/lick/lick.c
extension/lick/lick-core.fu2
extension/lick/lick.fpkg
extension/lick/lick.h
extension/meta_semantic_knowledge_base/meta_semantic_knowledge_base.c
extension/meta_semantic_knowledge_base/meta_semantic_knowledge_base-
    core.fu2
extension/meta_semantic_knowledge_base/meta_semantic_knowledge_base.
    fpkg
extension/meta_semantic_knowledge_base/meta_semantic_knowledge_base.h
extension/movie/movie.c
extension/movie/movie-core.fu2
extension/movie/movie.fpkg
extension/movie/movie.h
extension/pattern_match/pattern_match.c
extension/pattern_match/pattern_match-core.fu2
extension/pattern_match/pattern_match.fpkg
```

```
extension/pattern_match/pattern_match.h
extension/propogator/propogator.c
extension/propogator/propogator-core.fu2
extension/propogator/propogator.fpkg
extension/propogator/propogator.h
extension/semantic_action_event/semantic_action_event.c
extension/semantic_action_event/semantic_action_event-core.fu2
extension/semantic_action_event/semantic_action_event.fpkg
extension/semantic_action_event/semantic_action_event.h
extension/semantic_action_knowledge_base/semantic_action_knowledge_base
    .c
extension/semantic_action_knowledge_base/semantic_action_knowledge_base
    -core.fu2
extension/semantic_action_knowledge_base/semantic_action_knowledge_base
    .fpkg
extension/semantic_action_knowledge_base/semantic_action_knowledge_base
    .h
extension/semantic_action/semantic_action.c
extension/semantic_action/semantic_action-core.fu2
extension/semantic_action/semantic_action.fpkg
extension/semantic_action/semantic_action.h
extension/semantic_agent/semantic_agent.c
extension/semantic_agent/semantic_agent-core.fu2
extension/semantic_agent/semantic_agent.fpkg
extension/semantic_agent/semantic_agent.h
extension/semantic_category/semantic_category.c
extension/semantic_category/semantic_category-core.fu2
extension/semantic_category/semantic_category.fpkg
extension/semantic_category/semantic_category.h
extension/semantic_causal_concept/semantic_causal_concept.c
extension/semantic_causal_concept/semantic_causal_concept-core.fu2
extension/semantic_causal_concept/semantic_causal_concept.fpkg
extension/semantic_causal_concept/semantic_causal_concept.h
extension/semantic_causal_event/semantic_causal_event.c
extension/semantic_causal_event/semantic_causal_event-core.fu2
extension/semantic_causal_event/semantic_causal_event.fpkg
extension/semantic_causal_event/semantic_causal_event.h
extension/semantic_causal_object/semantic_causal_object.c
extension/semantic_causal_object/semantic_causal_object-core.fu2
extension/semantic_causal_object/semantic_causal_object.fpkg
extension/semantic_causal_object/semantic_causal_object.h
extension/semantic_cons/semantic_cons.c
extension/semantic_cons/semantic_cons-core.fu2
extension/semantic_cons/semantic_cons.fpkg
extension/semantic_cons/semantic_cons.h
extension/semantic_containment_object/semantic_containment_object.c
extension/semantic_containment_object/semantic_containment_object-core.
    fu2
extension/semantic_containment_object/semantic_containment_object.fpkg
extension/semantic_containment_object/semantic_containment_object.h
extension/semantic_counterfactual_transframe/semantic_counterfactual_
    transframe.c
extension/semantic_counterfactual_transframe/semantic_counterfactual_
    transframe-core.fu2
extension/semantic_counterfactual_transframe/semantic_counterfactual_
    transframe.fpkg
extension/semantic_counterfactual_transframe/semantic_counterfactual_
    transframe.h
extension/semantic_dependency/semantic_dependency.c
```

```
extension/semantic_dependency/semantic_dependency-core.fu2
extension/semantic_dependency/semantic_dependency.fpkg
extension/semantic_dependency/semantic_dependency.h
extension/semantic_directed_action_event/semantic_directed_action_event
    .c
extension/semantic_directed_action_event/semantic_directed_action_event
    -core.fu2
extension/semantic_directed_action_event/semantic_directed_action_event
    .fpkg
extension/semantic_directed_action_event/semantic_directed_action_event
    .h
extension/semantic_environment/semantic_environment.c
extension/semantic_environment/semantic_environment-core.fu2
extension/semantic_environment/semantic_environment.fpkg
extension/semantic_environment/semantic_environment.h
extension/semantic_event_knowledge_base/semantic_event_knowledge_base.c
extension/semantic_event_knowledge_base/semantic_event_knowledge_base-
    core.fu2
extension/semantic_event_knowledge_base/semantic_event_knowledge_base.
    fpkg
extension/semantic_event_knowledge_base/semantic_event_knowledge_base.h
extension/semantic_event/semantic_event.c
extension/semantic_event/semantic_event-core.fu2
extension/semantic_event/semantic_event.fpkg
extension/semantic_event/semantic_event.h
extension/semantic_event_sequence/semantic_event_sequence.c
extension/semantic_event_sequence/semantic_event_sequence-core.fu2
extension/semantic_event_sequence/semantic_event_sequence.fpkg
extension/semantic_event_sequence/semantic_event_sequence.h
extension/semantic_event_transframe/semantic_event_transframe.c
extension/semantic_event_transframe/semantic_event_transframe-core.fu2
extension/semantic_event_transframe/semantic_event_transframe.fpkg
extension/semantic_event_transframe/semantic_event_transframe.h
extension/semantic_event_tree/semantic_event_tree.c
extension/semantic_event_tree/semantic_event_tree-core.fu2
extension/semantic_event_tree/semantic_event_tree.fpkg
extension/semantic_event_tree/semantic_event_tree.h
extension/semantic_expectation_failure/semantic_expectation_failure.c
extension/semantic_expectation_failure/semantic_expectation_failure-
    core.fu2
extension/semantic_expectation_failure/semantic_expectation_failure.
    fpkg
extension/semantic_expectation_failure/semantic_expectation_failure.h
extension/semantic_frame/semantic_frame.c
extension/semantic_frame/semantic_frame-core.fu2
extension/semantic_frame/semantic_frame.fpkg
extension/semantic_frame/semantic_frame.h
extension/semantic_goal_action_causal_hypothesis/semantic_goal_action_
    causal_hypothesis.c
extension/semantic_goal_action_causal_hypothesis/semantic_goal_action_
    causal_hypothesis-core.fu2
extension/semantic_goal_action_causal_hypothesis/semantic_goal_action_
    causal_hypothesis.fpkg
extension/semantic_goal_action_causal_hypothesis/semantic_goal_action_
    causal_hypothesis.h
extension/semantic_goal_event/semantic_goal_event.c
extension/semantic_goal_event/semantic_goal_event-core.fu2
extension/semantic_goal_event/semantic_goal_event.fpkg
extension/semantic_goal_event/semantic_goal_event.h
```

```
extension/semantic_goal/semantic_goal.c
extension/semantic_goal/semantic_goal-core.fu2
extension/semantic_goal/semantic_goal.fpkg
extension/semantic_goal/semantic_goal.h
extension/semantic_knowledge_base/semantic_knowledge_base.c
extension/semantic_knowledge_base/semantic_knowledge_base-core.fu2
extension/semantic_knowledge_base/semantic_knowledge_base.fpkg
extension/semantic_knowledge_base/semantic_knowledge_base.h
extension/semantic_know_of_existence_event/semantic_know_of_existence_
    event.c
extension/semantic_know_of_existence_event/semantic_know_of_existence_
    event-core.fu2
extension/semantic_know_of_existence_event/semantic_know_of_existence_
    event.fpkg
extension/semantic_know_of_existence_event/semantic_know_of_existence_
    event.h
extension/semantic_know_of_relationship_event/semantic_know_of_
    relationship_event.c
extension/semantic_know_of_relationship_event/semantic_know_of_
    relationship_event-core.fu2
extension/semantic_know_of_relationship_event/semantic_know_of_
    relationship_event.fpkg
extension/semantic_know_of_relationship_event/semantic_know_of_
    relationship_event.h
extension/semantic_object/semantic_object.c
extension/semantic_object/semantic_object-core.fu2
extension/semantic_object/semantic_object.fpkg
extension/semantic_object/semantic_object.h
extension/semantic_object_type_event/semantic_object_type_event.c
extension/semantic_object_type_event/semantic_object_type_event-core.fu
    2
extension/semantic_object_type_event/semantic_object_type_event.fpkg
extension/semantic_object_type_event/semantic_object_type_event.h
extension/semantic_ordered_object/semantic_ordered_object.c
extension/semantic_ordered_object/semantic_ordered_object-core.fu2
extension/semantic_ordered_object/semantic_ordered_object.fpkg
extension/semantic_ordered_object/semantic_ordered_object.h
extension/semantic_packable_object/semantic_packable_object.c
extension/semantic_packable_object/semantic_packable_object-core.fu2
extension/semantic_packable_object/semantic_packable_object.fpkg
extension/semantic_packable_object/semantic_packable_object.h
extension/semantic_partial_state_property_relation/semantic_partial_
    state_property_relation.c
extension/semantic_partial_state_property_relation/semantic_partial_
    state_property_relation-core.fu2
extension/semantic_partial_state_property_relation/semantic_partial_
    state_property_relation.fpkg
extension/semantic_partial_state_property_relation/semantic_partial_
    state_property_relation.h
extension/semantic_partial_state_property/semantic_partial_state_
    property.c
extension/semantic_partial_state_property/semantic_partial_state_
    property-core.fu2
extension/semantic_partial_state_property/semantic_partial_state_
    property.fpkg
extension/semantic_partial_state_property/semantic_partial_state_
    property.h
extension/semantic_partial_state/semantic_partial_state.c
extension/semantic_partial_state/semantic_partial_state-core.fu2
```

```
extension/semantic_partial_state/semantic_partial_state.fpkg
extension/semantic_partial_state/semantic_partial_state.h
extension/semantic_plan_execution_node/semantic_plan_execution_node.c
extension/semantic_plan_execution_node/semantic_plan_execution_node—
    core.fu2
extension/semantic_plan_execution_node/semantic_plan_execution_node.
    fpkg
extension/semantic_plan_execution_node/semantic_plan_execution_node.h
extension/semantic_planner/semantic_planner.c
extension/semantic_planner/semantic_planner—core.fu2
extension/semantic_planner/semantic_planner.fpkg
extension/semantic_planner/semantic_planner.h
extension/semantic_plan_object/semantic_plan_object.c
extension/semantic_plan_object/semantic_plan_object—core.fu2
extension/semantic_plan_object/semantic_plan_object.fpkg
extension/semantic_plan_object/semantic_plan_object.h
extension/semantic_plan_object_type_event/semantic_plan_object_type_
    event.c
extension/semantic_plan_object_type_event/semantic_plan_object_type_
    event—core.fu2
extension/semantic_plan_object_type_event/semantic_plan_object_type_
    event.fpkg
extension/semantic_plan_object_type_event/semantic_plan_object_type_
    event.h
extension/semantic_plan_object_type_relation_event/semantic_plan_object
    _type_relation_event.c
extension/semantic_plan_object_type_relation_event/semantic_plan_object
    _type_relation_event—core.fu2
extension/semantic_plan_object_type_relation_event/semantic_plan_object
    _type_relation_event.fpkg
extension/semantic_plan_object_type_relation_event/semantic_plan_object
    _type_relation_event.h
extension/semantic_plan_operator_activation/semantic_plan_operator_
    activation.c
extension/semantic_plan_operator_activation/semantic_plan_operator_
    activation—core.fu2
extension/semantic_plan_operator_activation/semantic_plan_operator_
    activation.fpkg
extension/semantic_plan_operator_activation/semantic_plan_operator_
    activation.h
extension/semantic_plan_operator_parallel/semantic_plan_operator_
    parallel.c
extension/semantic_plan_operator_parallel/semantic_plan_operator_
    parallel—core.fu2
extension/semantic_plan_operator_parallel/semantic_plan_operator_
    parallel.fpkg
extension/semantic_plan_operator_parallel/semantic_plan_operator_
    parallel.h
extension/semantic_plan_operator_sequence/semantic_plan_operator_
    sequence.c
extension/semantic_plan_operator_sequence/semantic_plan_operator_
    sequence—core.fu2
extension/semantic_plan_operator_sequence/semantic_plan_operator_
    sequence.fpkg
extension/semantic_plan_operator_sequence/semantic_plan_operator_
    sequence.h
extension/semantic_plan_operator_suppression/semantic_plan_operator_
    suppession.fpkg
```

```
extension/semantic_plan_operator_suppression/semantic_plan_operator_
    suppression.c
extension/semantic_plan_operator_suppression/semantic_plan_operator_
    suppression-core.fu2
extension/semantic_plan_operator_suppression/semantic_plan_operator_
    suppression.fpkg
extension/semantic_plan_operator_suppression/semantic_plan_operator_
    suppression.h
extension/semantic_proprioception/semantic_proprioception.c
extension/semantic_proprioception/semantic_proprioception-core.fu2
extension/semantic_proprioception/semantic_proprioception.fpkg
extension/semantic_proprioception/semantic_proprioception.h
extension/semantic_proprioceptual_object/semantic_proprioceptual_object
    .c
extension/semantic_proprioceptual_object/semantic_proprioceptual_object
    -core.fu2
extension/semantic_proprioceptual_object/semantic_proprioceptual_object
    .fpkg
extension/semantic_proprioceptual_object/semantic_proprioceptual_object
    .h
extension/semantic_proprioceptual_orientation/semantic_proprioceptual_
    orientation.c
extension/semantic_proprioceptual_orientation/semantic_proprioceptual_
    orientation-core.fu2
extension/semantic_proprioceptual_orientation/semantic_proprioceptual_
    orientation.fpkg
extension/semantic_proprioceptual_orientation/semantic_proprioceptual_
    orientation.h
extension/semantic_proprioceptual_position/semantic_proprioceptual_
    position.c
extension/semantic_proprioceptual_position/semantic_proprioceptual_
    position-core.fu2
extension/semantic_proprioceptual_position/semantic_proprioceptual_
    position.fpkg
extension/semantic_proprioceptual_position/semantic_proprioceptual_
    position.h
extension/semantic_realm/semantic_realm.c
extension/semantic_realm/semantic_realm-core.fu2
extension/semantic_realm/semantic_realm.fpkg
extension/semantic_realm/semantic_realm.h
extension/semantic_reflective_object/semantic_reflective_object.c
extension/semantic_reflective_object/semantic_reflective_object-core.fu
    2
extension/semantic_reflective_object/semantic_reflective_object.fpkg
extension/semantic_reflective_object/semantic_reflective_object.h
extension/semantic_reflective_object_type_event/semantic_reflective_
    object_type_event.c
extension/semantic_reflective_object_type_event/semantic_reflective_
    object_type_event-core.fu2
extension/semantic_reflective_object_type_event/semantic_reflective_
    object_type_event.fpkg
extension/semantic_reflective_object_type_event/semantic_reflective_
    object_type_event.h
extension/semantic_reflective_object_type_property_event/semantic_
    reflective_object_type_property_event.c
extension/semantic_reflective_object_type_property_event/semantic_
    reflective_object_type_property_event-core.fu2
extension/semantic_reflective_object_type_property_event/semantic_
    reflective_object_type_property_event.fpkg
```

```
extension/semantic_reflective_object_type_property_event/semantic_
    reflective_object_type_property_event.h
extension/semantic_reflective_object_type_relation_event/semantic_
    reflective_object_type_relation_event.c
extension/semantic_reflective_object_type_relation_event/semantic_
    reflective_object_type_relation_event-core.fu2
extension/semantic_reflective_object_type_relation_event/semantic_
    reflective_object_type_relation_event.fpkg
extension/semantic_reflective_object_type_relation_event/semantic_
    reflective_object_type_relation_event.h
extension/semantic_relationship_key/semantic_relationship_key.c
extension/semantic_relationship_key/semantic_relationship_key-core.fu2
extension/semantic_relationship_key/semantic_relationship_key.fpkg
extension/semantic_relationship_key/semantic_relationship_key.h
extension/semantic_resource_action_event/semantic_resource_action_event
    .c
extension/semantic_resource_action_event/semantic_resource_action_event
    -core.fu2
extension/semantic_resource_action_event/semantic_resource_action_event
    .fpkg
extension/semantic_resource_action_event/semantic_resource_action_event
    .h
extension/semantic_resource_action_sequence/semantic_resource_action_
    sequence.c
extension/semantic_resource_action_sequence/semantic_resource_action_
    sequence-core.fu2
extension/semantic_resource_action_sequence/semantic_resource_action_
    sequence.fpkg
extension/semantic_resource_action_sequence/semantic_resource_action_
    sequence.h
extension/semantic_resource/semantic_resource.c
extension/semantic_resource/semantic_resource-core.fu2
extension/semantic_resource/semantic_resource.fpkg
extension/semantic_resource/semantic_resource.h
extension/semantic_self/semantic_self.c
extension/semantic_self/semantic_self-core.fu2
extension/semantic_self/semantic_self.fpkg
extension/semantic_self/semantic_self.h
extension/semantic_situation_category/semantic_situation_category.c
extension/semantic_situation_category/semantic_situation_category-core.
    fu2
extension/semantic_situation_category/semantic_situation_category.fpkg
extension/semantic_situation_category/semantic_situation_category.h
extension/semantic_situation/semantic_situation.c
extension/semantic_situation/semantic_situation-core.fu2
extension/semantic_situation/semantic_situation.fpkg
extension/semantic_situation/semantic_situation.h
extension/semantic_situation_transition/semantic_situation_transition.c
extension/semantic_situation_transition/semantic_situation_transition-
    core.fu2
extension/semantic_situation_transition/semantic_situation_transition.
    fpkg
extension/semantic_situation_transition/semantic_situation_transition.h
extension/semantic_somatosensation/semantic_somatosensation.c
extension/semantic_somatosensation/semantic_somatosensation-core.fu2
extension/semantic_somatosensation/semantic_somatosensation.fpkg
extension/semantic_somatosensation/semantic_somatosensation.h
extension/semantic_somatosensory_object/semantic_somatosensory_object.c
```

```
extension/semantic_somatosensory_object/semantic_somatosensory_object-
    core.fu2
extension/semantic_somatosensory_object/semantic_somatosensory_object.
    fpkg
extension/semantic_somatosensory_object/semantic_somatosensory_object.h
extension/semantic_temporal_object/semantic_temporal_object.c
extension/semantic_temporal_object/semantic_temporal_object-core.fu2
extension/semantic_temporal_object/semantic_temporal_object.fpkg
extension/semantic_temporal_object/semantic_temporal_object.h
extension/semantic_thought/semantic_thought.c
extension/semantic_thought/semantic_thought-core.fu2
extension/semantic_thought/semantic_thought.fpkg
extension/semantic_thought/semantic_thought.h
extension/semantic_time/semantic_time.c
extension/semantic_time/semantic_time-core.fu2
extension/semantic_time/semantic_time.fpkg
extension/semantic_time/semantic_time.h
extension/semantic_visual_object/semantic_visual_object.c
extension/semantic_visual_object/semantic_visual_object-core.fu2
extension/semantic_visual_object/semantic_visual_object.fpkg
extension/semantic_visual_object/semantic_visual_object.h
extension/timeline/timeline.c
extension/timeline/timeline-core.fu2
extension/timeline/timeline.fpkg
extension/timeline/timeline.h
extension/transframe/transframe.c
extension/transframe/transframe-core.fu2
extension/transframe/transframe.fpkg
extension/transframe/transframe.h
fu2/action.fu2
fu2/actor.fu2
fu2/actortest.fu2
fu2/assembler.fu2
fu2/bootstrap-apropos.fu2
fu2/bootstrap-array.fu2
fu2/bootstrap-boot.fu2
fu2/bootstrap-bug.fu2
fu2/bootstrap-bugs.fu2
fu2/bootstrap-cause.fu2
fu2/bootstrap-cause_group.fu2
fu2/bootstrap-command_line.fu2
fu2/bootstrap-compound_object.fu2
fu2/bootstrap-conditionlock.fu2
fu2/bootstrap-cons.fu2
fu2/bootstrap-core_extension.fu2
fu2/bootstrap-core_extension_funk.fu2
fu2/bootstrap-critic.fu2
fu2/bootstrap-critics-reactive.fu2
fu2/bootstrap-default_critics.fu2
fu2/bootstrap-defragmenter.fu2
fu2/bootstrap-dotimes.fu2
fu2/bootstrap-dynamic_library.fu2
fu2/bootstrap-fiber.fu2
fu2/bootstrap-frame.fu2
fu2/bootstrap.fu2
fu2/bootstrap-garbage_collector.fu2
fu2/bootstrap-graph.fu2
fu2/bootstrap-graph-old.fu2
fu2/bootstrap-grid.fu2
```

```
fu2/bootstrap_hash.fu2
fu2/bootstrap_largeinteger.fu2
fu2/bootstrap_list.fu2
fu2/bootstrap_math.fu2
fu2/bootstrap_nil.fu2
fu2/bootstrap_object.fu2
fu2/bootstrap_package.fu2
fu2/bootstrap_primobject.fu2
fu2/bootstrap_ptypes.fu2
fu2/bootstrap_reader.fu2
fu2/bootstrap_redblacktree.fu2
fu2/bootstrap_repl.fu2
fu2/bootstrap_set_theory.fu2
fu2/bootstrap_sort.fu2
fu2/bootstrap_string.fu2
fu2/bootstrap_surrogate_parent.fu2
fu2/bootstrap_terminal_print.fu2
fu2/bootstrap_time.fu2
fu2/bootstrap_type_conversions.fu2
fu2/bootstrap_zlib.fu2
fu2/brainviz.fu2
fu2/cardgame_ai.fu2
fu2/cardgame.fu2
fu2/cause.fu2
fu2/characters.fu2
fu2/compile.fu2
fu2/emailcharacters.fu2
fu2/emotionmachine.fu2
fu2/english_eval.fu2
fu2/graph.fu2
fu2/graph_match_test.fu2
fu2/graphviz.fu2
fu2/internet.fu2
fu2/link_grammar_wrapper.fu2
fu2/miscfunks.fu2
fu2/neuralmom_brain_area.fu2
fu2/neuralmom_demo.fu2
fu2/neuralmom_nervous_system.fu2
fu2/neuralmom_occipital_cortex.fu2
fu2/opengl.fu2
fu2/pattern.fu2
fu2/planner.fu2
fu2/primfunks_apropos.fu2
fu2/primfunks_arithmetic.fu2
fu2/primfunks_array.fu2
fu2/primfunks_bug.fu2
fu2/primfunks_cause.fu2
fu2/primfunks_chunk.fu2
fu2/primfunks_command_line.fu2
fu2/primfunks_compile.fu2
fu2/primfunks_core_extension.fu2
fu2/primfunks_core_extension_funk.fu2
fu2/primfunks_cpu.fu2
fu2/primfunks_defragmenter.fu2
fu2/primfunks_dlfcn.fu2
fu2/primfunks_errno.fu2
fu2/primfunks_fcntl.fu2
fu2/primfunks_fiber.fu2
fu2/primfunks_fiber_trigger.fu2
```

```
fu2/primfunks-frame.fu2
fu2/primfunks.fu2
fu2/primfunks-garbage_collector.fu2
fu2/primfunks-gmodule.fu2
fu2/primfunks-graph.fu2
fu2/primfunks-hash.fu2
fu2/primfunks-ioctl.fu2
fu2/primfunks-largeinteger.fu2
fu2/primfunks-locale.fu2
fu2/primfunks-management_thread.fu2
fu2/primfunks-math.fu2
fu2/primfunks-memory.fu2
fu2/primfunks-object.fu2
fu2/primfunks-optimize.fu2
fu2/primfunks-package.fu2
fu2/primfunks-package_handler.fu2
fu2/primfunks-primes.fu2
fu2/primfunks-primmetros.fu2
fu2/primfunks-primobjects.fu2
fu2/primfunks-primobject_type.fu2
fu2/primfunks-primobject_type_handler.fu2
fu2/primfunks-print.fu2
fu2/primfunks-ptypes.fu2
fu2/primfunks-reader.fu2
fu2/primfunks-redblacktree.fu2
fu2/primfunks-scheduler.fu2
fu2/primfunks-set.fu2
fu2/primfunks-signal.fu2
fu2/primfunks-socket.fu2
fu2/primfunks-sort.fu2
fu2/primfunks-stdlib.fu2
fu2/primfunks-string.fu2
fu2/primfunks-surrogate_parent.fu2
fu2/primfunks-terminal_print.fu2
fu2/primfunks-termios.fu2
fu2/primfunks-time.fu2
fu2/primfunks-trace.fu2
fu2/primfunks-virtual_processor_handler.fu2
fu2/primfunks-zlib.fu2
fu2/reactive.fu2
fu2/readline-wrapper.fu2
fu2/repl.fu2
fu2/rlglue-wrapper.fu2
fu2/serialize.fu2
fu2/story.fu2
fu2/thought_process.fu2
fu2/trace.fu2
fu2/x86-compile.fu2
fu2/x86-compile-machine_code.fu2
fu2/x86-compile-mov.fu2
misc/fables.fu2
misc/frog_and_toad.fu2
misc/frog-and-toad.fu2
misc/officer_joke.fu2
misc/roboverse-blocks_world.fu2
misc/roboverse-demo.fu2
misc/simple_game.fu2
python/funk2module/c/funk2module.c
python/funk2module/c/funk2test.c
```

```
test/cairo-test/cairo-test.fpkg
test/cairo-test/cairo-test.fu2
test/concept_version_space-test/concept_version_space-test.fpkg
test/concept_version_space-test/concept_version_space-test.fu2
test/gtk-test/gtk-test.fpkg
test/gtk-test/gtk-test.fu2
test/interval_tree-test/interval_tree-test.fpkg
test/interval_tree-test/interval_tree-test.fu2
test/keyboard-test/keyboard-test.fpkg
test/keyboard-test/keyboard-test.fu2
test/keyboard-test/ncurses-test.c
test/optimize-test/optimize-test.fpkg
test/optimize-test/optimize-test.fu2
test/propogator-test/propogator-test.fpkg
test/propogator-test/propogator-test.fu2
test/timeline-test/timeline-test.fpkg
test/timeline-test/timeline-test.fu2
test/xmlrpc-test/xmlrpc-test.fpkg
test/xmlrpc-test/xmlrpc-test.fu2
```

BIBLIOGRAPHY

Bertsekas, D. (1995), *Dynamic Programming and Optimal Control*, Vol. 1, Athena Scientific Belmont, MA.

Bowling, M. & Veloso, M. (2000), 'An Analysis of Stochastic Game Theory for Multiagent Reinforcement Learning', *Computer Science Department, Carnegie Mellon University* .

Campbell, J., Moyers, B. D. & Flowers, B. S. (1988), *The power of myth*, Doubleday New York.

Charniak, E. & Goldman, R. P. (1993), 'A bayesian model of plan recognition', *Artificial Intelligence* **64**(1), 53–79.

Cox, M. & Raja, A. (2008), Metareasoning: A manifesto, *in* 'Proceedings of AAAI 2008 Workshop on Metareasoning: Thinking about Thinking', pp. 1–4.

Cox, M. & Raja, A. (2010), 'Metareasoning: An Introduction', *Cox, M., Raja, A. (Ed.), Metareasoning: Thinking about Thinking* .

Cox, M. T. & Ram, A. (1999), On the Intersection of Story Understanding and Learning, *in* A. Ram & K. Moorman, eds, 'Understanding Language Understanding: Computational Models of Reading', MIT Press, Cambridge, Massachusetts, chapter 11, pp. 397–434.

Dongarra, J., Gannon, D., Fox, G. & Kennedy, K. (2007), 'The impact of multicore on computational science software', *CTWatch Quarterly* **3**(1), 1–10.

Doyle, J. (1978), Truth Maintenance Systems for Problem Solving, PhD thesis, Massachusetts Institute of Technology.

Džeroski, S., De Raedt, L. & Driessens, K. (2001), 'Relational reinforcement learning', *Machine Learning* **43**(1), 7–52.

Good, I. (1971), 'Twenty-seven principles of rationality', *Foundations of statistical inference* pp. 108–141.

Hume, D. (1748), *Philosophical essays concerning human understanding*, Georg Olms Publishers.

Kaelbling, L., Littman, M. & Moore, A. (1996), 'Reinforcement learning: A survey', *Arxiv preprint cs.AI/9605103* .

Kautz, H. (1987), A formal theory of plan recognition, PhD thesis, University of Rochester.

Kerkez, B. & Cox, M. T. (2003), 'Incremental case-based plan recognition with local predictions', *International Journal on Artificial Intelligence Tools* **12**(04), 413–463.

Maes, P. (1987), 'Concepts and Experiments in Computational Reflection', *SIGPLAN Not.* **22**(12), 147–155.

Maes, P. (1988), Issues in Computational Reflection, *in* 'Meta-level Architectures and Reflection', North-Holland, pp. 21–35.

Messmer, B. (1995), Efficient graph matching algorithms for preprocessed model graphs, PhD thesis, University of Bern, Switzerland.

Messmer, B. & Bunke, H. (2000), 'Efficient subgraph isomorphism detection: a decomposition approach', *Knowledge and Data Engineering, IEEE Transactions on* **12**(2), 307–323.

Minsky, M. (1975), 'A Framework for Representing Knowledge', *The Psychology of Computer Vision* pp. 211–279.

Minsky, M. (2006), *The Emotion Machine: Commonsense Thinking, Artificial Intelligence, and the Future of the Human Mind*, Simon & Schuster, New York, New York.

Minsky, M. (2011), 'Interior grounding, reflection, and self-consciousness', *Information and Computation: Essays on Scientific and Philosophical Understanding of Foundations of Information and Computation* **2**, 287.

Mitchell, T. (1997), 'Machine learning (ise)', *Recherche* **67**, 02.

Morgan, B. (2009), 'Funk2: A distributed processing language for reflective tracing of a large critic-selector cognitive architecture', *Proceedings of the Metacognition Workshop at the Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems* .

Morgan, B. (2012), 'Funk2 programming tutorial'.
URL: *https://github.com/bunuelo/funk2/wiki*

Pryor, L. & Collins, G. (1995), Planning to Perceive, *in* A. Ram & D. Leake, eds, 'Goal-driven learning', MIT Press, Cambridge, Massachusetts, chapter 10, pp. 287–296.

Pryor, L., Collins, G. et al. (1992), Planning to perceive: A utilitarian approach, *in* 'Proceedings of the AAAI 1992 Spring Symposium on Control of Selective Perception', Citeseer.

Radul, A. & Sussman, G. J. (2009), The art of the propagator, *in* 'Proceedings of the 2009 International Lisp Conference'.

Rao, S. (1998), Visual routines and attention, PhD thesis, Massachusetts Institute of Technology.

Rapoport, A. (2001), *N-person game theory: Concepts and applications*, Courier Dover Publications.

Russell, S. & Wefald, E. (1991), *Do the right thing: studies in limited rationality*, MIT press.

Saxe, R., Schulz, L. & Jiang, Y. (2006), 'Reading minds versus following rules: Dissociating theory of mind and executive control in the brain', *Social Neuroscience* **1**(3-4), 284.

Simon, H. (1957), *Administrative behavior*, Vol. 4, Cambridge Univ Press.

Simon, H. (1982), 'Models of bounded rationality: Behavioral economics and business organization, vol. 2'.

Singh, P. (2002), 'The panalogy architecture for commonsense computing'.

Singh, P. (2005), EM-ONE: An Architecture for Reflective Commonsense Thinking, PhD thesis, Massachusetts Institute of Technology.

Singh, P. & Minsky, M. (2005), An architecture for cognitive diversity, *in* D. Davis, ed., 'Visions of Mind', Idea Group Inc., London.

Singh, P., Minsky, M. & Eslick, I. (2004), 'Computing commonsense', *BT Technology Journal* **22**(4), 201–210.

Smith, D. & Morgan, B. (2010), Isisworld: An open source commonsense simulator for ai researchers, *in* 'Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence'.

Sodan, A., Machina, J., Deshmeh, A., Macnaughton, K. & Esbaugh, B. (2010), 'Parallelism via multithreaded and multicore cpus', *Computer* **43**(3), 24–32.

Stoytchev, A. (2005), Toward learning the binding affordances of objects: A behavior-grounded approach, *in* 'Proceedings of AAAI Symposium on Developmental Robotics', pp. 17–22.

Sussman, G. (1973), A computational model of skill acquisition, PhD thesis, Massachusetts Institute of Technology.

Ullman, S. (1984), 'Visual routines', *Cognition* **18**(1), 97–159.

Velez, J., Hemann, G., Huang, A., Posner, I. & Roy, N. (2011), Planning to perceive: Exploiting mobility for robust object detection, *in* 'Proc. ICAPS'.

Wilensky, R. (1981), 'Meta-planning: Representing and using knowledge about planning in problem solving and natural language understanding', *Cognitive science* **5**(3), 197–233.

Winograd, T. (1970), Procedures as a Representation for Data in a Computer Program for Understanding Natural Language, PhD thesis, Massachusetts Institute of Technology.

Winston, P. (1970), Learning structural descriptions from examples, PhD thesis, Massachusetts Institute of Technology.

Winston, P. (2011), The strong story hypothesis and the directed perception hypothesis, *in* 'Papers from the AAAI fall symposium, Technical report FS-11-01', pp. 345–352.

Zilberstein, S. (2011), Metareasoning and bounded rationality, *in* M. T. Cox & A. Raja, eds, 'Metareasoning: Thinking about thinking', MIT Press, Cambridge, Massachusetts, pp. 27–40.

COLOPHON

This thesis was typeset with LaTeX 2_ε using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera." (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.) The typographic style was inspired by Bringhurst as presented in *The Elements of Typographic Style* (Bringhurst 2002). It is available for LaTeX via CTAN as "`classicthesis`."

Final Version as of August 6, 2013 at 12:55.

DECLARATION

I, Bo Morgan, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Cambridge, June 2013

Bo Morgan