A SIMULATION SUBSTRATE FOR REAL-TIME PLANNING

Scott D. Anderson

Tech. Report. 95-80

To Holly, with love

Acknowledgements

I'm filled with wonder and gratitude as I realize how much support and help I have gotten over my years in graduate school. It was, at times, a difficult task, and I truly could not have completed it without a great deal of help.

Foremost, I thank my advisor, Paul Cohen. No one could ask for an advisor with more energy and enthusiasm. Paul believed in me at times when I did not believe in myself, and more than once talked me into continuing the Ph.D. For his extraordinary support (and the occasional kick in the tail), I am very grateful.

I thank my colleagues in the Experiment Knowledge Systems Laboratory. David Hart always manages the lab with skill, humor, and grace. Michael Greenberg implemented the original PHOENIX system, writing an extraordinary amount of code in a surprisingly short time, and all the while taught me to play bridge and code Lisp. David Westbrook took over PHOENIX and has done a tremendous job improving Mike's handiwork, and helped me in innumerable ways in finishing this dissertation. Without their amazing skill, I would have had nothing to work with. I'm pleased to count these men among my friends.

I've had many friends through graduate school, all of whom helped me in ways they probably don't even realize. David Lewis taught me how to focus on the essentials of the research. David Forster taught me how to avoid research by TeX hacking, a skill that was invaluable in the final formatting. Penni Sibun showed me the importance of avoiding repetitive strain injury. Martin Herbordt demonstrated how to avoid extraneous distractions. Eric Hansen helped me read the literature. Frank Klassner reminded me to stand up for my principles when I needed it. Rob St. Amant taught me how to use JMP™ when I was at the end of my rope. Cindy Loiselle showed me that you can be a success even if you don't finish the Ph.D., while always encouraging me to finish. Dorothy Mammen inspired me with her unfailing tenacity. Steve Dropsho helped me with some crucial references, and he showed me how to lead. Paul Booth showed me how to follow my own path. In addition to being helpers and role models, these people are dear friends.

Only an autobiography would do justice to Adele Howe, Ross Beveridge, Bruce Draper, and Dan Neiman. These friends have stood by me through my years of grad school, counseling me, supporting me, helping me and spurring me on. The word "friend" seems inadequate, but it will have to do.

Finally, I thank my wife and family. They didn't always understand what I was doing or how graduate school works, but they were fiercely partisan and a bedrock of support. To all of these people, I am profoundly grateful.

Abstract

A SIMULATION SUBSTRATE FOR
REAL-TIME PLANNING

SEPTEMBER 1995

SCOTT D. ANDERSON, B.S., YALE UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Paul R. Cohen

Real-time planning is, generally speaking, problem-solving under time-pressure. In order to test and evaluate real-time planners, scientists need to have environments that pose problems to be solved. This dissertation describes a substrate, called MESS, for building environments and real-time planners. MESS relies on a second system, TCL, to replace CPU time as a way to measure the amount of thinking done by a real-time agent.

The key issue with real-time planning simulators is that some of the simulation processes represent real-world events and other processes represent *thinking* or *planning* that occurs while the real-world events do. Therefore, there needs to be a correspondence between real-world events and thinking processes. Most current simulators make the correspondence by measuring the CPU time of the thinking process, which means that every bit of thinking is "on the clock," but which also results in measurement error and portability troubles.

MESS incorporates TCL, a programming language for agents. TCL is very nearly Common Lisp, augmented by functions for interacting with the rest of the simulation. The thinking time for each primitive in TCL is determined by a platform-independent function. Thus, an agent's every thought is on the clock but in a platform-independent way. The functions representing thinking time are stored in a database that is inspectable by the agent, so that the agent can be aware of its own thinking time and can use that time for planning. The functions can model the duration of the agent's thinking at either the fine-grained level of Common Lisp primitives or at higher levels of abstraction, in which the duration model of a primitive is independent of the implementation of the primitive. The fine-grained representation is important for researchers interested in the timing properties of particular algorithms, while the abstract representation is important for those modeling other implementations of the algorithms (such as parallel hardware), those modeling different kinds of systems (such as cognitive scientists modeling brain function), or those modeling more global properties, such as the interaction of the durations of various activities.

Table of Contents

Page

Acknowledgements v

Abstract vi

List Of Tables x

List Of Figures xi

1. Introduction 1
    1.1 Cognitive Processes 1
    1.2 Duration Models of Cognitive Actions 5
    1.3 An Agent Language 7
    1.4 Time Management 8
    1.5 Summary 8

CHAPTER

2. Current Simulator Technology 10
    2.1 Computer Systems and Architecture 10
    2.2 Real-Time Systems 11
    2.3 Simulators for AI Planning 12
        2.3.1 Phoenix 13
        2.3.2 RALPH 14
        2.3.3 Truckworld 15
        2.3.4 MICE 16
        2.3.5 TileWorld 17
        2.3.6 DVMT 17
        2.3.7 Trains 18
        2.3.8 NASA TileWorld 19
        2.3.9 Ars Magna 19
        2.3.10 McDermott's Simulator 20
        2.3.11 TÆMS 20
    2.4 Summary 21

3. Design 23
    3.1 Simulation Models 23
    3.2 Architecture 24
        3.2.1 Engine 25

3.2.2 Event Streams 28

3.3 Mess Examples 32

3.3.1 Lives 32
3.3.2 Clocks 34
3.3.3 M/M/1 Queue 36
3.3.4 Registrar 37

3.4 Timed Common Lisp 40

3.4.1 Packages 41
3.4.2 Operations 42
3.4.3 Duration Models 43
3.4.4 Free Operations 45
3.4.5 Code Walking 46
3.4.6 Extending TCL 49
3.4.7 Querying the Database 49
3.4.8 Call Counting 50
3.4.9 Summary 50

3.5 Activities 51

3.5.1 Interaction 51
3.5.2 Interruptions 53

3.6 Summary 56

4. Evaluation 58

4.1 Necessity 58

4.1.1 Gabriel 58
4.1.2 Registrar 60
4.1.3 Phoenix 62

4.2 Success 65
4.3 Introspection 67
4.4 Replicability 70
4.5 Performance 73

4.5.1 Mess 73
4.5.2 TCL 76

4.6 Summary 79

5. Contributions 80

Bibliography 84

List of Tables

Table Page

4.1 Plan performance data comparing the MODEL and SHELL plans. . . 74

4.2 Relative performance of TCL on the Gabriel benchmarks. 78

List of Figures

Figure | Page

1.1 A hypothetical call tree for the **move** function of a chess agent. 6

3.1 The architecture of the MESS simulation substrate. 24

3.2 Pseudo-code for the MESS engine. 25

3.3 Algorithm to run a MESS simulation for some amount of time. 28

3.4 A simple example of co-routining. 30

3.5 A transcript of the "lives" simulation. 34

3.6 The events from a "clocks" simulation. 35

3.7 The "next event" function for the Arrival event stream. 37

3.8 The "next event" function for the Departure event stream. 37

3.9 The events from the "random walk" simulation. 38

3.10 The queue length in a simulation of an M/M/1 queue. 38

3.11 Contrasting the implementation of CPU and TCL thinking streams. 39

3.12 The Registrar's thought process. 42

3.13 A transcript of the TCL code-walker. 48

3.14 Part of the implementation of the "vacuuming" example. 52

3.15 The chess players thinking in parallel. 54

3.16 Code and pseudo-code for the chess-playing agents. 55

4.1 Raw data for the Gabriel benchmarks. 59

4.2 Log variance as a function of log mean CPU time. 61

4.3 Four runs of the CPU-time registrar for 64 events each. 63

4.4 Three sets of PHOENIX trials using identical starting conditions. . . . 64

4.5 Data from the original RTK experiment. 66

4.6 Data from the new RTK experiment. 66

4.7 Making fire projection a TCL primitive 69

4.8 Data for the RTK Introspection experiment. 71

4.9 Data from the matched-pairs experiment 72

4.10 The speed of the MESS engine, plotting all data. 75

4.11 The speed of the engine, plotting mean performance. 77

Chapter 1

Introduction

As Artificial Intelligence (AI) planning research moves away from the simple "blocks worlds" of its early years and engages complicated problems involving many objects, time pressure, sensor error, effector error, and plan failure, it's necessary to use a simulator to see whether a plan will work and how it fails. That is, the planning is done by an autonomous AI program, called an *agent*, which thinks and acts in the simulated world. Many different simulators have been implemented, with different domains and characteristics depending on the research interests of their sponsors.

A great deal of programming effort has gone into these simulators and I have not replicated or subsumed this effort. Instead, I have implemented a domain-independent *substrate* upon which to build simulators, because, despite their differences, these simulators are almost all based on discrete event simulation. My substrate, called MESS (Multiple Event Stream Simulator) employs a language, called TCL (Timed Common Lisp), for representing the thinking of agents. The design of MESS and TCL are described in chapter 3, which presents the bulk of the thesis work. I have also re-implemented the PHOENIX simulator (see section 2.3.1) on top of MESS, as a test of MESS's capabilities and to demonstrate how it supports simulators for particular domains. This introduction will describe the kinds of simulators necessary for research in real-time planning and how, in general terms, MESS and TCL will achieve those properties.

A fairly common idea in AI simulators is that the world changes while the agent is thinking: fires burn, ships and trains move, tiles appear and disappear, and so on. This results in time pressure on the agent, since windows of opportunity may close before the agent acts. Sometimes the problem definition includes an explicit deadline; for example, in the Distributed Vehicle Monitoring Testbed (DVMT), described in section 2.3.6, the agent is required to notify someone of a threatening pattern of movement within a certain amount of time [19]. In other simulators, time pressure results simply from trying to perform well; for example, in PHOENIX, the firefighting agents must act quickly so as to minimize the area of forest burned. To implement time pressure, these simulators must interleave world processes with the thinking of the agents.

1.1 Cognitive Processes

How does time advance while an agent thinks? Two approaches are used in current AI simulators: CPU time and explicit duration. This section will describe each of these approaches and explain their advantages and disadvantages. The next section will describe a hybrid approach: it uses an agent language (TCL) for all the code of an agent (like the CPU-time approach) and yet the duration of the language is CPU-independent (like the explicit-duration approach).

An agent's thinking is accomplished by executing some Lisp code, and executing code takes time, so a natural approach is to make the advance of simulation time directly proportional to the CPU time of the executed code. In short, the simulator executes some amount of the agent's code, measures the CPU time of that execution, and advances the simulation clock by a simple function of the CPU time. For example, in PHOENIX [7, 20], one CPU-second corresponds to five minutes of time in world processes (simulation time). This approach is simple and does much of what we want, because the more that the agent thinks, the more time passes. Hence, the CPU-time approach is quite popular, being used in MICE, TRUCKWORLD, RALPH and other simulators, as well as in PHOENIX.

Another approach to advancing time is to designate an explicit duration for particular cognitive actions ("thoughts"). Whenever the agent thinks one of those thoughts (that is, the simulator executes some particular piece of code on behalf of the agent), the clock is advanced by the predesignated amount. This approach is also simple and works well when the agent's thinking is dominated by a few such cognitive actions. For example, the explicit duration approach is used in the DVMT, where processing is dominated by executing knowledge sources.

Perhaps the two approaches can be clarified by means of an example. Consider a simulation of a chess-playing agent. The CPU-time approach measures the CPU time that elapses during the agent's turn and advances the clock by the correct amount. If the chess agent decides to search deeper and consider more positions, the CPU time will naturally increase and so will the simulation time. Whatever the agent spends its time thinking about, the clock will continue to tick. For the explicit-duration approach, we must choose the cognitive actions that we will assign durations to. Suppose we let a whole turn be a cognitive action, taking one minute. If the agent decides to look at more positions, the turn will still have the same duration. Every turn in the game takes exactly one minute, regardless of how much thinking occurs. On the other hand, suppose we let the consideration of a single board position be a cognitive action, taking five seconds. Now if the agent decides to look at more positions, each additional position costs it five seconds, whether the agent rejects it immediately or ponders it very carefully. Also, whatever the agent thinks about other than chess positions (such as long-term strategy, how much time remains in the game, or whether checkers would be more fun) does not advance the clock and is therefore free. Of course, if these other thinking activities are important, they can be defined as cognitive actions and given explicit durations.

What are the advantages and disadvantages of each approach? Assuming that the underlying Lisp system provides a way of measuring CPU time, the CPU-time approach is very simple and everything is "on the clock," meaning that every thought advances the simulation clock. The simplicity undoubtedly accounts for much of the popularity of the CPU-time approach. The explicit-duration approach, on the other hand, is a little more difficult, because the person implementing the agent must designate particular parts of the code as cognitive actions and specify a duration for each. Obviously, the duration is insensitive to the algorithm of the cognitive action, though this trouble can be mitigated by moving to a finer level of granularity, as in the chess example when we moved from a "turn" as a cognitive action to a "position consideration" as a cognitive action.

The insensitivity of the duration to the algorithm can be a great advantage to the explicit-duration approach when the algorithm is of no interest. For example, a researcher may be interested in modeling operations that are unrelated to current implementations. Vision, memory, and other cognitive functions may be implemented in an agent, but the researcher, particularly a cognitive scientist interested in modeling the human brain, may not want the duration of those functions to depend on the serial algorithm that implements them. A function can even be implemented with a neural net, and the researcher may want to model the function's duration depending on the number of iterations of the net but independently of the number of nodes. The explicit-duration approach allows this flexibility of modeling.

Another advantage of explicit durations is platform independence. With the CPU-time approach, on the other hand, if we run the agent on faster hardware or use a better compiler, it gets smarter relative to the environment. The environment processes (fires, ships, trucks) still proceed at the same pace (although the simulation runs quicker), but the CPU time of the agent's thinking is reduced and so the clock is not advanced as quickly. If our research concerned a particular Lisp platform, we would want this platform dependence, but as AI researchers, we are interested in a much more abstract view of an agent's thinking speed. We want, for example, the speed of our chess agent to depend on the intelligence with which moves are considered (so that little time is wasted on unpromising ones), the cleverness which which time is managed, and so forth—not on the speed of the CPU.

This abstract view is much better handled by the explicit duration approach, since that approach encourages software to be categorized in semantic chunks (such as "position consideration") and is platform-independent because the clock is advanced by the same amount regardless of the CPU. One disadvantage of the explicit duration approach is that the level of abstraction tends to be fixed, so that in implementing an agent we must commit to either a "turn" or a "position consideration" being a cognitive action. A related disadvantage is that there are often many interesting things that the agent thinks about that have not been captured as cognitive actions and hence are off the clock. For example, the DVMT, which uses the explicit-duration approach, was the testbed for Durfee's thesis on Partial Global Planning [14], where he said that

> the DVMT's current inability to simulate the time needs for control limit our ability to measure control overhead in the experiments in chapters 5 and 8 [p. 53].

This is precisely a case where an interesting cognitive activity, control overhead (deciding what cognitive action to do next), is off the clock. Later, in explaining the compromises and approximations made to estimate control overhead, Durfee says that "modifying the DVMT to simulate the cost for control activity would be a major effort" (p. 344).

Control overhead, or any kind of overhead, is particularly difficult to measure because it is defined by negation. That is, overhead is whatever remains after all the labelled cognitive actions are removed. For example, if the chess agent takes five minutes to make a move, during which time it examined 20 positions for ten seconds each, and spent 50 seconds thinking about its long-term strategy, 50 seconds is still unaccounted for. The 50 seconds may have gone into control (switching between position examination and strategy evaluation) or into other activities such as learning, time management, or anything else that the agent was thinking about.[1] That 50 seconds is reasonably called "overhead." To measure overhead, we need a way of measuring the total time of some task, inclusive of every cognitive action, as well as time for each cognitive action, so that we have all the terms of the subtraction.

A practical disadvantage of the CPU-time approach is difficulty in debugging. The same lack of exact reproducibility on different platforms means that some bugs—ones that depend on timing—must be debugged on particular machines. Even then, the bug might not be reproducible, due to inaccuracies in the measurement of CPU time. Finally, any modification of the code, say by inserting print statements, will change the timing and might make the bug change form or disappear, only to reappear later. This follows from the fact that, with the CPU-time approach, everything is on the clock, including debugging statements that are temporarily inserted. (Because PHOENIX is based on a modification of the operating system, it is possible to, roughly, suspend CPU-time accounting during debugging statements. This is not possible in most current Lisp systems, which don't provide their own operating system. Furthermore, the PHOENIX approach is only approximate; variance still remains.) Lack of reproducibility due to inaccuracies in measuring CPU time is a major concern with the CPU-time approach. The explicit-duration approach allows debugging statements to be inserted into the code and given zero duration, so that they can have no effect on the behavior of the agent.

We've seen two approaches to modeling the thinking time of agents: the CPU time of the agent code and explicit durations for particular cognitive actions. The advantages are mainly on the side of the explicit duration approach, because it allows models of thinking time that are platform-independent and flexible. The advantages of the CPU-time approach are that it is simple, since no models need to be defined, and that everything is on the clock. Having everything on the clock not only allows calculation of overhead, but avoids inadvertent cheating, where additional code that makes an agent smarter doesn't result in the agent taking any more time. In general, there is a tradeoff between speed and intelligence (reflexes are fast but stupid, while deliberation is smart but slow), and so everything that an agent thinks about should cost it something. With an explicit-duration approach, it's easy to add code that costs nothing. To combine these approaches would result in a way of modeling the computation time of an agent that can capture the detailed execution time of complex algorithms while also being able to abstract away from those details, all in a platform-independent way. This is what TCL does.

[1]Some have wondered whether garbage collection (GC) would be included in overhead. Generally no, because GC is handled by a different process in most commercial, multi-threaded Lisp systems, so any simulator (except the NASA TileWorld and TRAINS) will ignore time taken by GC. GC contributes to the non-repeatability of simulators using the CPU-time approach, but it doesn't contribute to the duration models.

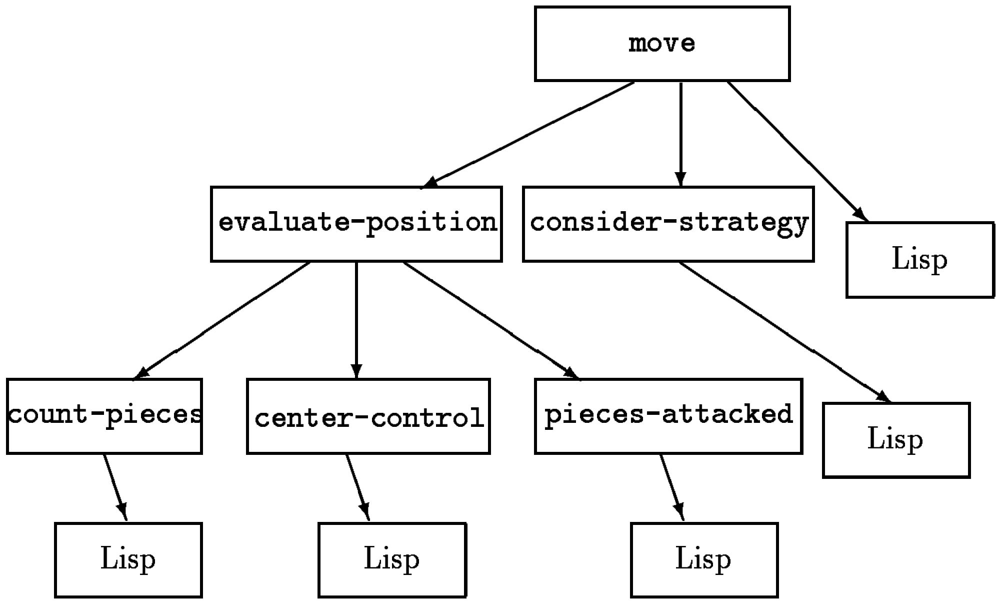


Figure 1.1. A hypothetical call tree for the `move` function of a chess agent.

1.2 Duration Models of Cognitive Actions

What is a cognitive action? If we assume that the agent's thinking is defined by a program written in Lisp,[2] cognitive actions might be the particularly important functions. TCL requires cognitive actions to be functions, since such a requirement is no real restriction and allows us to think of cognitive actions that have names, take arguments and return values. (A user can extend TCL with new cognitive actions by using the `declare-tcl-primitive` function, described in section 3.4.2.) Consider figure 1.1, which shows an imagined "call tree" for the `move` function in a chess agent. `Move` calls `evaluate-position` and `consider-strategy`, as well as calling Lisp primitives directly. Eventually, the tree grounds out in Lisp primitives, which are directly executable. We can also view the Lisp primitives as cognitive actions, albeit very low-level ones.

By assigning a duration to each Lisp primitive, we can define a duration for any cognitive action as the sum of all the durations of the Lisp primitives executed in its call tree. A better definition is recursive, defining the duration of an action as the sum over all the actions it calls. The recursive definition lets us give explicit duration models for internal nodes as well as for the leaves of the call tree. For example, rather than let the duration of `count-pieces` be defined by its Lisp code, we can define the duration to be, say, five seconds, or any other model we want. The result

[2]Throughout this dissertation, I assume that programs are written in Common Lisp. The ideas of the dissertation can be implemented using other programming languages, but I have chosen Lisp because (1) it is one of the standard languages in artificial intelligence, (2) it particularly lends itself to the kind of programming I needed to do, and (3) it's the language I'm most familiar with. The relevance of the work is not limited to Lisp.

is that the duration of **move** is a composition of the duration of Lisp primitives and other models explicitly supplied by the user. A duration model at any level preempts models from any lower level. Thus, the TCL approach can seamlessly combine the detailed, complex duration models using Lisp primitives and the simple, abstract duration models provided by the user.

Complex duration models, tied closely to the actual code that implements a cognitive action, are important for those studying particular algorithms. For example, the algorithm might do real-time path planning, perhaps by some variation on real-time A* [31]. The ability to look at algorithms in such detail will be useful to researchers on anytime algorithms [9], among others. Other researchers work at a higher level, where they don't care about the particular algorithms that implement a cognitive action but rather are interested in, for example, the scheduling and control of such algorithms. These researchers will want simple, abstract models of the duration, suitable for input to a scheduling algorithm. Other researchers will want to look at the effects of cognitive action duration; for example, they might want to explore the performance effects of plan monitoring that is of long or short duration. Another example is experiments to discover the tradeoffs between time spent on temporal projection (an agent imagining the future) versus replanning upon plan failure, possibly due to an errorful temporal projection. For such experiments, simple models of duration may be best.

A user-defined duration model need not be a constant, even though our examples have often been constants. A duration model for **evaluate-position**, for example, might depend on the number of pieces, the number of pawns, or other aspects of the current situation. For the sake of generality, duration models in TCL may be arbitrary Lisp functions, which are called when the cognitive action is executed in order to determine how long that code ran. Currently, TCL has six built-in duration models and seventeen others defined via this general mechanism (see section 3.4.3).

The duration model is supplied with the arguments and values of the cognitive action, so that it can be sensitive to the current situation. For example, the "hinders," "facilitates" and similar relationships of DVMT tasks [10] can be implemented by having the duration model check what other tasks have executed. The duration model can also be pseudo-random, generating durations from appropriate probability distributions, as is done in TÆMS, which is described in section 2.3.11. (The generation must be pseudo-random so that simulations are repeatable.) Finally, the duration model could even measure the CPU time of the cognitive action on the current platform; in this way, the duration models can subsume both the explicit-time and CPU-time approaches.

Flexibility of duration models and the ability to separate the model from the underlying implementation is particularly important to cognitive modeling, as mentioned in the previous section. For example, SOAR [32,40,50] is viewed as a serious model of human cognition, yet there are aspects of the implementation which the SOAR researchers make no theoretical commitment to, such as the RETE algorithm. They may make a commitment to certain functions, but not to the algorithms implementing those functions. This is the A|B distinction described by Cooper *et al.* [8], where the cognitive action is above the line (A) while the algorithm is below the line (B)—the line demarcates the boundary between theoretical commitment and implementation detail. The ability to use a different timing model for those cognitive actions can be crucial for attempting to replicate human timing data.

The duration models are stored in a database that is read by TCL during simulation. If TCL is given a different database, the durations of actions will be different, so multiple duration models can be used for different kinds of experiments. TCL comes with a database that specifies the duration for all Lisp primitives, so a user can, if desired, run experiments without defining any duration models at all.

1.3 An Agent Language

The database of duration models is used by an agent language that notates the agent's computation and advances the clock based on the duration models. The agent language solves one of the biggest problems with the explicit-duration method by ensuring that everything that the agent thinks about is on the clock. The agent language is called Timed Common Lisp (TCL). It has all the functions of the Common Lisp language, but each function refers to a database entry to advance the simulation clock. One can program in TCL just as in Common Lisp; the languages look the same.

How does time advance in a chess agent written in TCL? In practice, the approach looks very much like the CPU-time approach, since the agent language is executed as usual to compute the agent's thoughts. However, instead of measuring the CPU time of the code and advancing the simulation clock accordingly, the simulation clock is advanced directly by the TCL code. If the chess algorithms change, or the number of positions considered changes, the run time changes accordingly, just as with the CPU-time method. The primary difference between the TCL approach and CPU-time approach is that CPU time is platform-dependent, while TCL will give identical behavior on any computer. Thus, TCL solves one of the biggest problems with the CPU-time approach.

1.4 Time Management

The duration models, stored in a database, can be useful to the agent as well as to TCL. If the agent knows from checking the database that considering a board position is going to take five seconds, it can make a rational decision as to whether to keep searching or to stop. In this dissertation, I will refer to this ability as *introspection.* Through introspection, duration models support resource-bounded reasoning [51–54] and design-to-time algorithms [19]. Furthermore, agents will want to have simple, abstract models to work with, so there is an incentive to add such models to the database.

The CPU-time approach can't support time management, because the only way to tell how long a computation will take is to run it and see. Indeed, any approach suffers this problem when the resolution of the question doesn't match the resolution of the answer. In the chess example, knowing the explicit duration of a board evaluation doesn't particularly help to figure out how long a whole turn will take, unless an *a priori* decision is made about how many positions to consider and there's no variation in overhead.

One major approach to time management is to compile statistical data on the runtime of functions; these statistics are sometimes called "performance profiles" [3,9,59]. The TCL substrate supports the collection of performance profiles by recording the duration of cognitive actions at all levels of abstraction. This data can be stored in the same database with the duration models, so that it is available to agents for time management. The researchers who might benefit most from TCL are those interested in time management (anytime algorithms, design-to-time algorithms, and resource-bounded reasoning), because the computation time of the agent is a crucial part of the research. TCL gives them a great deal of control over computation time, a platform-independent standard for computation time, and automatic collection of computation time statistics.

1.5 Summary

This dissertation describes the design of MESS, the Multiple Event Stream Simulator, and Timed Common Lisp (TCL) which is the language for implementing agent thinking within the simulation. MESS is best thought of as a simulation *substrate*, because it makes no commitment to a domain, but rather supports simulation in terms of abstract events. The user implements a domain by defining classes of events and event streams that produce those events at the proper times. An agent's thought process is one kind of event stream, and the TCL language can be used for the programming in that event stream. TCL uses a database of duration models to define the duration of the agent's thinking activity and advance the clock of that event stream. The database allows the models to be detailed and tied to the particular algorithm or as simple and abstract as desired; in either case, the models are platform-independent. TCL makes it easy to integrate events that are cognitive actions with events in the simulated world and to integrate different kinds of duration models.

The primary contributions of the work are:

- the MESS substrate, which allows rapid prototyping of simulators for real-time planning and is extensible, so that it can be tailored to the needs of individual researchers.
- the TCL language, which gives platform-independent, replicatable, durations for agent thinking. The primitives can be defined along a wide range of levels, from low-level Lisp functions to high-level domain-specific functions like evaluating a chess position.
- the integration of MESS and TCL, which allows real-time agents to interact with a simulation of a dynamic world.
- support for interactions between simulated processes, particularly interactions with agent thinking, allowing an agent's thoughts to be interrupted by some simulation event.
- support for introspection by agents, so that they can reason about and control their thought processes depending on the time pressure and the duration of those thoughts.

Chapter 2 reviews the state of simulation technology relevant to real-time planning. Chapter 3 describes the design of MESS and TCL. The fourth chapter specifies the properties I claim for the research and describes the experiments that confirm those properties. The last chapter summarizes the work.

Chapter 2

Current Simulator Technology

Simulation is a way of modeling and studying any system that is too complex for mathematical analysis. Consequently, there is a vast literature on simulation, because of its nearly universal applicability. Simulation has been used to study systems as diverse as weather, manufacturing plants, galaxies, ecosystems, hospitals, jet engines, and so forth. In real-time planning, the object of study is *computation*, because we are interested in controlling and adapting the computation (planning) of an agent depending on time-pressure due to the duration of the computation.

The bulk of this chapter reviews simulators used in real-time planning research, but it begins with discussing two other areas where the object of study is computation, namely computer architecture and real-time computer systems.

2.1 Computer Systems and Architecture

When designing a new computer system, the designers often want to evaluate the design prior to implementation, to see if it will meet its goals of functionality and performance [28, 56]. For example, a company is designing a network of computer workstations that will allow access to a large database on a mainframe connected to the network. The goals may be to get acceptable response times for database queries. The questions for the designers might be "how much memory does each workstation need?" "how much disk capacity?" "what should the network connectivity be?" and so forth. It may be too expensive to wait until the system is built before answering these questions, and so the designers may model the system and simulate it to get performance estimates. This allows redesign if the performance is deemed inadequate.

One aspect of the simulation model is the processing time of various components of the system. Usually, processing time is dominated by the time for input and output (I/O), so the component's execution is modelled by estimating the number of disk accesses it does or estimating the number of remote procedure calls it makes across a network. (One recent example modeled a client/server system architecture [55].) When the processing time is less dominated by I/O time, CPU time must be estimated, usually by estimating the number of assembly language instructions and dividing by the speed of CPU in instructions per second. One way is to directly estimate the number of assembly language instructions by using some kind of cross-compiler [13, 49]; this way requires the actual code to be run, which is typically not available early in the design process of a new software system, but may be available if an existing software system is to run on new hardware. Another way is to estimate (perhaps from past experience) the number of lines of high-level code in the component and the number of assembly language instructions needed for each line of code, and then multiply to get a ballpark estimate of the amount of assembly code [56, p. 122].

The processing times of components of the software system are then used to build a simulation model. The simulation model is usually a queuing-like model, with the service times in various queues determined by these estimated processing times.

Software performance modeling is important when building very large systems, where even very rough estimates of processing time can be helpful. Because such modeling is done at design time, the simulation model is necessarily very abstract and high-level. Furthermore, they are tied to particular computer systems, disks and CPUs. MESS and TCL, in contrast, are focused on much more detailed simulations of the behavior and timing of actual code, but divorced from any particular computer system, because of the need for portability. Secondly, TCL is designed for introspection, allowing an agent to control its computation depending on its duration.

Simulation modeling is also important when evaluating novel computer architectures, such as new cache architectures or special parallel hardware for signal processing or computer vision. Martin Herbordt's dissertation [26] gives an excellent survey of techniques for simulating and evaluating computer architectures. The main idea is to devise a representative workload in terms of programs to run on the new hardware, execute the programs (in some way) to get a long "trace" of the assembly language instructions they perform, and then simulate that trace to estimate execution time, cache hit rates, or whatever performance measures are desired. The trace may be used multiple times with different configurations of the simulated hardware, to see the effect of those configurations. For that reason, the trace is often created once and stored for multiple uses; this is called trace-driven simulation. However, this is very costly in disk-space because the traces are often huge. Therefore, the trace is often created dynamically by a some kind of execution of the workload programs; this is called execution-driven simulation.

Execution-driven simulation resembles MESS and TCL, in that a program (agent) is executing in such a way that the duration of its execution affects the behavior of the simulation. The differences are fairly clear: as with modeling software systems, the duration modeling is at a low, hardware-specific level, and the simulation lacks the introspection support that TCL provides. This lack is not a criticism of simulation in computer architecture or software performance, since these research fields are not concerned with dynamic control of programs based on their duration.

2.2 Real-Time Systems

Real-time systems are, generally speaking, systems where the value or utility of a computation depends not just on what is computed but on when it's computed: a result that is late is less valuable than one that meets its deadline. A complex real-time system coordinates many tasks, with a variety of deadlines and resource usage, to try to maximize the overall value of the computations. Therefore, real-time systems research is very interested in predicting and controlling the execution time of tasks.

The SPRING project at the University of Massachusetts is a prime example of real-time systems research. Douglas Niehaus's dissertation [43] gives an excellent survey of real-time systems research and describes the SPRING project in detail. Niehaus's thesis concerns predicting the execution time of tasks and integrating those predictions into a real-time system, including a scheduler for those tasks. In particular, he introduces the SPRING-C language, a variant of C in which all loops and recursions are bounded, so that the worst-case execution time (WCET) of the program can be computed at compile time. The SPRING-C compiler finds the WCET by a sub-graph reduction technique, which begins by computing the WCET for the basic blocks of a graph representing the control structure of the program. The WCET of a basic block is computed by summing the execution times of the assembly language instructions for the block. The algorithm then successively collapses the graph[1] until it is irreducible. The WCET for the irreducible program is stored in a database, to be used by the SPRING system's task scheduler.

The SPRING-C language is similar in some respects to TCL, the primary difference being that the duration of SPRING-C programs is tied to a particular CPU rather than being measured in abstract, portable "duration units." Furthermore, there is no opportunity to introduce new, high-level primitives, whose duration does not depend on their implementation.

The SPRING-C language does not allow introspection by the tasks, so that they can decide what computational method to use based on the time-pressure, but the SPRING scheduler can make decisions like that. If a task is divided into a mandatory and optional part, the scheduler will only execute the optional part if, given the WCET of that optional part, all tasks will still make their deadlines. Thus, the real-time system as a whole has the adaptive abilities of a real-time planning agent, albeit tailored to worst-case execution time rather than expected execution time, as in real-time planning.

2.3 Simulators for AI Planning

A number of simulators have already been implemented to support various aspects of AI planning, such as reasoning and planning cooperatively, under uncertainty, under time pressure, in complex or noisy environments, and so forth. Since MESS makes no commitment to a domain or environment, most of these aspects are irrelevant to its design. Because the main contribution of this thesis is the accounting for time in the agent language, this review of the literature will only give a brief overview of each simulator. The overview will mention the simulator's strengths and weaknesses, but will concentrate on the representation and implementation of agent computation and how the simulator accounts for an agent's computation time, hence its suitability as a simulator for real-time planning.

An excellent survey article about simulators in planning was published in AI Magazine [24]. Naturally, it concentrated on the simulators developed by the authors: Steve Hanks's TRUCKWORLD, Martha Pollack's TILEWORLD, and Paul Cohen's PHOENIX. The article is also important because it discusses why we do experiments in planning. We experiment not only to find out why our agents (programs) work and whether they have the properties we intended them to have, but also to infer broader truths about how they will behave in other environments. This inference

[1]For example, an "if" node of the graph is collapsed by collapsing both arms of the branch and then computing the WCET of the "if" node as the maximum over the WCET of each arm, plus the WCET of the test expression.

from the specific experiment to the broader theory is difficult, as it is in any science. A researcher might claim that the experiment showed boldness in an agent to be better than cowardice. A more conservative researcher might say only that for the agent architecture and simulated world, "boldness" (as defined in the experiment) was better than cowardice. In either case, the challenge is given to the AI community to generalize or restrict these claims by testing the hypothesis with other agent architectures in other environments on other tasks.

A simulator doesn't make a result general: only many experiments can do that. A simulator can, however, make it easy to run many different experiments. One flaw with the TILEWORLD testbed is the commitment it makes to a particular agent architecture and environment, which limits the range of experiments that can be run. MESS makes no domain commitment and will allow more sharing of code, including allowing agents to run more easily in different domains. Thus, it greatly expands the potential number of domains that agents can be tested in. The platform-independence of thinking time is an important step in promoting replication and generalization of results.

2.3.1 Phoenix

PHOENIX [7, 20], developed by Paul Cohen's research group at the University of Massachusetts, implements agents that fight forest fires in Yellowstone National Park. There are no explicit deadlines, but the nature of fire-fighting naturally puts time pressure on the agents to control the fire before it gets out of hand. The fire is controlled by "fireline," strips of land cleared of burnable material by bulldozers. The bulldozers are agents whose efforts are coordinated by a "fireboss" agent. Other agents include watchtowers, fuel trucks, and helicopters.

The agents have a common architecture in which a "timeline" (a structure of plans and actions) is incrementally executed. This timeline gives an explicit representation of the state of a plan, so that if a plan or action fails, the failure recovery code can inspect the state and decide what to do [27]. The agent can interleave planning and execution, because the choice of how to expand a subplan is delayed until all predecessors of that subplan have been executed. The plan notation includes parallel and sequential structures, and plans can pass parameter values to their subplans.

The timeline of a PHOENIX agent is interpreted by an operating system process running on the TI Explorer™ Lisp Machine. The operating system has been modified so that when a process gives up the CPU, the CPU time for that slice is computed and the agent's clock advanced by the product of the CPU time and a constant for that agent called the "real-time knob." Thus, the computation time of an agent is determined as a linear function of its CPU time.

Although a great deal of effort has gone into controlling the swapping of processes, the remaining non-determinism affects runs of the PHOENIX simulator. Two trials from identical initial states and identical random number seeds will not necessarily act identically. (Section 4.1.3 reports just such an experiment, using PHOENIX, contrasting it with trials using MESS and TCL that do perform identically.) This difficulty in replicating behavior makes debugging hard. It also eliminates certain kinds of experiments, such as running a scenario twice, varying only a single decision. Furthermore, processes run for an entire quantum, making certain small violations of causality possible: one agent can look at the world at, say, 12:02, and then another agent will run, modifying the world at 12:01. These are minimized by keeping the time quantum small—agents can be out of step by a maximum of five minutes.

I believe that the CPU-time method in PHOENIX has been a source of a number of troubles, namely difficulty in debugging and lack of replicability across platforms. Running large experiments on multiple platforms was tainted by irrelevant variance in the results, because running PHOENIX on an Explorer II produced different behavior than on an Explorer I. In fact, there were variations even between the same CPU, possibly caused by differences in memory size or disk speed. Furthermore, CPU time is impossible to predict, making it difficult to allow the agent to have prior knowledge of the duration of its plans and actions. (This could be accumulated offline, as described in section 1.4, but that is not implemented in PHOENIX.) Finally, CPU time is difficult to control, so that the duration of one cognitive action cannot be varied independently of another. An important experiment investigating the performance of the PHOENIX planner under different settings of the real-time knob [25] had some results that bore further investigation, but it was difficult to do this within the design of PHOENIX. This is not to say that PHOENIX is a failure, merely that some kinds of experiments are difficult to run.

2.3.2 RALPH

The RALPH (Rational Agents with Limited Performance Hardware) testbed [44] was developed by Stuart Russell's research group at the University of California at Berkeley. It has a number of user-defined agents called "ralphs" moving about on a gridworld that contains food (cupcakes), walls, and moving adversaries called "nasties." Ralphs follow the rational economic model of trying to maximize their utility. Eating a cupcake raises a ralph's utility, so an optimal decision-theoretic strategy must eat as many cupcakes as possible while avoiding nasties and so forth. RALPH defines a clear, simple interface between ralphs and the world. They may move in one of the four gridworld directions, sense, eat, and so forth. The world can be extended in many ways, including being able to have tiles for the ralphs to move, allowing the testbed to mimic TILEWORLD, described in section 2.3.5, below.

In a RALPH scenario, the agents and the nasties take turns acting. During each turn, a ralph chooses an action and performs it. The default kind of ralph can think for as long as it wants before choosing an action, but real-time constraints can be imposed in two ways. The first limits the computation time of each turn to a constant, a particular number of CPU-seconds on that platform. If an agent takes too long to choose its action, it loses its turn. The second way is similar, except that the agent's computation is interrupted at the end of its turn and continued during its next turn, rather than starting again from the beginning. This forces the agent to keep track of time and allows it to allocate more than a single turn of computation to an action choice, should the agent feel the choice was crucial. In both cases, the mapping from computation time to the corresponding simulation time is from CPU-seconds to "turns," rather than time units in a simulated world, as in PHOENIX.

Clearly, the RALPH testbed is platform-dependent, because of the use of CPU time. The RALPH implementers plan to offer a means of standardizing time units across platforms by benchmarking, so that comparisons can be made, but I think that actual replication will be impossible. Indeed, it may be hard to replicate on a single platform, since operating-system interrupts and CPU-time measurement are not deterministic. Therefore, I think the RALPH testbed suffers from the same troubles that the PHOENIX testbed does. It will be hard to debug, hard to perform detailed comparisons between scenarios (since replicability is necessary), and hard to run the kinds of real-time experiments that one might want to run.

2.3.3 Truckworld

TRUCKWORLD [23,24,41,42,58] was written by Steve Hanks's research group at the University of Washington. In TRUCKWORLD, the agents are trucks with two robot arms, which they use to grasp and manipulate objects, such as fuel drums, spare tires and cargo. They also sense by grasping objects; for example, grasping a "camera" gives a truck visual input. The trucks move about in a world of roads and intersections (that is, a graph), with fuel and cargo at positions in that world. (Note the contrast to PHOENIX, where space is essentially continuous, and RALPH, which is a gridworld; TRUCKWORLD is a "graphworld.") Like the preceding testbeds, TRUCKWORLD includes uncertainty (plans and actions don't always succeed), exogenous change (events can occur other than those caused by the agent), semi-realistic sensing (sensing takes time and may be noisy), and communication between agents.

TRUCKWORLD is intended to be used in a client/server relationship, with the server process running the simulation and maintaining the world state, while the trucks run in their own processes, connecting to the server by TCP/IP[2] sockets, and sending commands, such as movement, grasping, and so forth. This design makes for a clear separation between agents and the environment.

In general, it seems that actions and events take time, but not thought processes. Indeed, since the agents are running in separate processes and merely sending commands to the simulator, it's hard to see how the simulator could limit their thought processes in any way. However, there is a "real-time" mode that the clients can be run in, where the user stipulates a mapping from CPU time to simulation units (the real-time knob for TRUCKWORLD) and a quantum for discretizing that mapping. Thinking time is reported to the simulator along with commands.

Essentially, TRUCKWORLD accounts for time using the same approach as the PHOENIX testbed. Therefore, it probably suffers from the same difficulties: behavior is platform-dependent and it's hard to replicate situations.

2.3.4 MICE

MICE [15,39] was implemented by Edmund Durfee's research group at the University of Michigan to support their investigations into coordination among cooperating agents. MICE models a group of agents moving about on a gridworld; the user determines what environmental constraints the testbed enforces (such as whether two agents may occupy the same grid square) and how violations are handled (such as sending the conflicting agents back to their previous locations). The testbed is not

[2]TCP/IP is the standard protocol for inter-process communication on the Internet.

concerned with the internal architecture of the agent, only with its interface. The agent produces commands such as movement, sensing and communication, whereupon the testbed takes care of modifying the world representation.

Because the MICE research group is concerned with concurrent action and with reproducibility, they have used a discrete-event simulator to implement parallelism, rather than using parallelism at the operating-system level (as in PHOENIX). They model computation time the same way PHOENIX does, measuring the CPU time of cognitive actions and converting them to simulation time using a real-time knob. One difference is that the agent must execute an explicit "think" action to do thinking, which then affects the world simply by advancing the clock.

There are some very good ideas in the design of MICE, but I believe that the continued dependence on CPU time will have the same ill effects it had in PHOENIX, namely difficulty in replication, unwanted variance in experiments, and incompatibility of work in different research labs. The unpredictability of CPU time makes meta-reasoning about computation time difficult.

2.3.5 TileWorld

Pollack and Ringuette's TILEWORLD [47] simulates a simple gridworld where the agent gets "points" for filling holes with tiles by pushing the tiles around. The tiles and holes appear and disappear randomly, thereby putting a premium on quick, efficient planning, action, sensing, time management and failure recovery. The testbed also simulates a particular agent architecture, called IRMA—the Intelligent Resource-bounded Machine Architecture [5]. The testbed and agent architecture are highly parameterized, to support controlled experimentation.

In its first implementation, the computation time of the agent was modeled using CPU time, as in the preceding simulators. Their experimental results confirm the experiences found in PHOENIX, namely that using CPU time can add unwanted variance to the data. In their AAAI paper [47], they state that

> The noise in the data comes, we believe, largely from our decision to use actual CPU-time measurements to determine reasoning time. If we wish to get the cleanest trials possible, we may need to use a time estimate that does not depend on the vagaries of the underlying machine and Lisp system.

Because of this trouble, later implementations of TILEWORLD abandoned CPU time and went to an explicit-duration method. Because the IRMA architecture has a look/reason/act loop, they implemented a fixed duration for the "reason" part of the loop, depending on whether the agent does any thinking during that loop:

> The remaining factor is how fast the agent thinks. We used a fixed estimate for the amount of time taking in reasoning. If the agent deliberates and/or does plan expansion, we assume that the reasoning part of the cycle takes 1780 milliseconds. Otherwise, we assume that the reasoning takes 10 milliseconds. These values are representative of the actual times taken by the reasoning modules; using actual CPU times results in non-repeatable experiments [29].

This approach gives the desired repeatability, but takes away the sensitivity to how much thinking the agent does, as discussed in the introduction. It gives an advantage to agents that can do a great deal of thinking up front in order to skip as many cycles as possible. It also doesn't allow control over different kinds of thinking: it's all one undifferentiated activity called "deliberation."

2.3.6 DVMT

Victor Lesser's research group at the University of Massachusetts implemented the Distributed Vehicle Monitoring Testbed (DVMT) [33, 34] to support their research in distributed AI. The DVMT focuses on a sensor-interpretation task, where a group of distributed nodes each receives signals about vehicles moving through its area. Processing these signals and exchanging information with other nodes allows the ensemble of nodes to track and predict the movements of vehicles. Often the task includes hard time constraints, such as noticing a suspicious movement pattern within a certain time.

The DVMT assumes a blackboard-based architecture for the agent, with the interpretation time dominated by the execution of knowledge-source instances (KSIs) that interpret base-level signals, groups of signals, or whole patterns. Research issues include load balancing (how much data to send to another node for processing) [11], sharing of intermediate results, plans and goals (and coordination algorithms to achieve this) [14], and real-time processing [19, 35].

For real-time experiments, the DVMT does not use a CPU-time approach, but instead uses explicit duration models of the execution time of KSIs in a platform-independent way. The execution time of a particular KSI is of the form $ax + b$, where x is a measure of the amount of input to the KSI, such as the number of sensor signals to be processed. (In practice, the constant a is often zero, resulting in each KSI having a constant processing time.) This approach allows the behavior of the DVMT to be easily portable; indeed, the DVMT has been implemented on VAX/VMS, Explorers and other computer systems.

The drawback, as described in the introduction, is that only the duration of KSIs is counted. Time for control is unaccounted for. The best that can be said is that it is probably small relative to the KSI duration. When researchers such as Durfee want to measure control overhead, they have to resort to measuring the total CPU time of the simulation. I believe that having a complete accounting for all aspects of computation in the DVMT would benefit their research in real-time AI.

In subsequent work by Norman Carver [6], the DVMT knowledge sources were replaced with more sophisticated algorithms for distributed reasoning under uncertainty, in a system called DRESUN. Real-time experiments with DRESUN used CPU time to measure the planner's thinking time relative to the arrival of sensor data. The measurement of CPU did allow greater detail in accounting for the activities of the planner, as Carver demonstrated.

2.3.7 Trains

TRAINS [36] simulates a transportation domain where agents store, process and transport goods. It was developed by James Allen's research group at the University of Rochester as an environment for testing their theories on planning and natural language. A user can do the planning (instructing the system using natural language such as "move the bananas to York") or the user can be replaced by an AI planner. Plans are represented as condition-action pairs. Agents may communicate as well, even sending plans to one another. The environment is one of cities and rail connections, much like the TRUCKWORLD environment.

TRAINS can run in several modes, one of which is *real-time* mode, which "coordinates the ticks in the simulation with the computer's real time clock" [36, p. 6]. What I believe this means is that the simulation of world processes is slowed down to real time, by waiting whenever the time of an event is ahead of the computer's clock. Agents have only as much time to think as they would in the real world, possibly less, and the time that they think is measured as real-time. This approach is similar to a CPU-time approach, but allows other processes to intrude, such as garbage collection, paging, the simulation of the world, and even the activity of other users on the computer. The result is an extremely realistic way to test "real-time" agents, but it sacrifices many of the advantages of a simulator, because their testbed suffers from the same non-determinism, complexity, and lack of control as the real world. Simulators are often used in order to shield the research from some of the complexity and variability of the real world, so that carefully controlled experiments can be run, but this is lost in TRAINS. Finally, the TRAINS real-time mode makes the testbed too slow to run large experiments.

2.3.8 NASA TileWorld

The NASA TileWorld (NTW) was developed by Philips, Bresina and others [45, 46] independently of Pollack and Ringuette's TILEWORLD. They are similar in that an agent moves tiles around on a grid, but in the NASA TileWorld, there is no score associated with filling holes; instead, the goals are imposed from outside. There are no holes or obstacles. Furthermore, plan failure is caused not by the disappearance of holes and tiles, but by winds that move tiles around and the failure of grippers to hold onto the tiles.

An NTW agent feels time pressure because it must grasp a tile before the tile is blown out of reach by the wind. This deadline is measured in real-world seconds. The physical movement speed of the simulated agent is measured in cells per real-world second, and, assuming that the deadline is not too tight, this leaves some amount of time for planning and other computation. The modeling of computation time is the same as in TRAINS, since it's the actual run-time of the Lisp code that matters. The authors proudly state:

> The dynamic real-time behavior of the simulator, ..., enables the construction of experiments that are more like "real world" problems in the sense that no two runs will be identical.

Like the TRAINS testbed, the NASA TileWorld is more of a real-time system than a simulator, since simulators are intended to be like the real world, but allow experimental control and reduction of variance. This testbed explicitly denies control. Certainly it allows some control over the experiment, but it aims more for complexity and realism than other simulators.

2.3.9 Ars Magna

ARS MAGNA [17] (Abstract Realistic Simulation of Mobile Robots for Analyzing Goal-achievement, Navigation and Adaptation) was developed by Sean Engelson and Niklas Bertani at Yale. It simulates an indoor domain for an abstract mobile robot. The domain has walls and doorways and such, and the robot has two arms and a storage bin for grasping objects and carrying them about. (In this respect, it is very similar to TRUCKWORLD.) Positions and objects in the world can have arbitrary properties, and objects can be affected by arbitrary exogenous events caused by "kickers." Sensors and effectors are noisy and take time. When agents initiate a sensor or effector action, they can specify what to do if the action should fail (by supplying a function to call).

ARS MAGNA uses McDermott's sophisticated Reactive Plan Language (RPL) [37] which is a Lisp-like language for robot programming. RPL includes resources, parallel threads, condition monitoring, interrupts from fluents and many other features. The agent language I describe in this thesis is simpler and lower-level than RPL but completely compatible. Indeed, RPL could be implemented using TCL, giving RPL reliable, repeatable timing behavior.

It's not clear how time pressure is implemented in ARS MAGNA, if it is at all. The simulator was built primarily for Engelson's thesis [16], which was not concerned with time pressure. The environment and the agent take turns during each moment of the simulation, so the agent does indeed think while the world changes, but it appears to be allowed to take as much time as it needs during its turn.

2.3.10 McDermott's Simulator

Drew McDermott and his colleagues at Yale use a simulator very similar to ARS MAGNA (indeed, it is the precursor of ARS MAGNA) but with a simpler model of space. Rather than a continuous 2-D space, objects have numbered "positions" in a room: for example, a red ball at position 0, a box at position 1, and so forth. Nothing has been published about this simulator per se, but it is sketched in McDermott's technical report on transformational planning [38]. Like ARS MAGNA, McDermott's simulator is tightly integrated with RPL. The robot executes an RPL program while a planner executing in parallel tries to improve the program. If an improvement is found before the program is completed, the old one is interrupted and the robot begins executing the new, improved program. This cycle continues until the robot's program completes.

This simulator doesn't support time pressure, because there are no exogenous events and hence nothing for the agent to race against, unlike a PHOENIX agent's race against the fire or a DVMT agent's race against a deadline. To the extent that there is any racing, the planner races against the controller (the process executing the plan). After a trial, the agent's performance is measured and the utility of the planner determined by contrasting the robot's performance with and without the planner.

2.3.11 TÆMS

TÆMS [10,12] was developed by Keith Decker at the University of Massachusetts to explore issues in real-time, distributed, multi-agent systems. It is something of a departure from the preceding simulators because there is no "performance" aspect—no simulated robot moving around and doing stuff in a simulated environment. That is, rather than execute real agent software, TÆMS executes abstract "task groups," which are directed acyclic graphs of "tasks" (abstract cognitive actions), which consume time and produce "quality." TÆMS can model many interactions among tasks, including precedence (where one task must precede another) and hindrance or facilitation (where executing one task before another aids or hinders the second). These inter-task relationships can affect the duration and quality of the tasks, and many kinds of interactions can be modeled in a very general way. TÆMS can provide empirical results on real-time experiments involving, for example, methods for coordinating multiple agents or algorithms for scheduling tasks.

At a high level, the issue in real-time processing involves trading quality for time. Therefore, the tasks in TÆMS simply advance the clock and report the quality of their results, as a function of the quality of their inputs—that is, the quality that a real cognitive action such as sensor interpretation might produce if it were given real input and obeyed the model supplied to TÆMS. One way to use TÆMS, for example, is to test a scheduler that, in an effort to meet a deadline, may substitute a task that produces less quality but consumes less time. The duration of tasks is either specified by the user or drawn from probability distributions (in order to generate a space of problems). Similarly, the quality models must be either explicitly specified by the user or randomly assigned by TÆMS; they don't derive from any real problem-solving software. A wide variety of coordination and scheduling algorithms can be tested in this framework without the burden of solving domain-dependent problems. Indeed, TÆMS solves no problem at all, producing only statistics on execution time and quality.

TÆMS is clearly a useful, domain-independent tool for targeted experiments in coordination and scheduling, especially where there are interrelationships between tasks. Its control of timing allows powerful experiment designs, such as matched-paired studies (pairs of simulations that differ in exactly one way). It allows a space of coordination and scheduling algorithms to be explored in an efficient way. However, its generality may make it insufficient for many researchers who are interested in the "robot and environment" simulators, such as PHOENIX or TRUCKWORLD, described above.

I view TÆMS as complementary to MESS, in that TÆMS can use the timing and quality information generated from a specific agent and environment, allowing the researcher to explore alternative scheduling and coordination paradigms without running simulations in the particular domain. The time/quality results from such exploration can be tested by running domain-level simulations after reconfiguring the agent and environment based on the TÆMS results.

2.4 Summary

This chapter began by reviewing simulation technology in related research areas where the object of study involves the computation time of some software, namely software systems performance, computer architecture, and real-time systems. This simulation technology concentrated on computation time for particular hardware and largely omitted the need for introspection.

The bulk of the chapter has reviewed a number of simulators used in AI planning and, although they use a variety of techniques to model the time for cognition, they fall roughly in the two broad categories described in the introduction. The CPU-time approach is used by PHOENIX, TRUCKWORLD, MICE, and RALPH. It also appears to be used in NTW, TRAINS, and McDermott's simulator. (ARS MAGNA appears not to implement time pressure.) On the other hand, the DVMT and TÆMS use the explicit-duration approach. Significantly, the initial TILEWORLD implementation used CPU time, but later implementations use explicit duration, so as to eliminate the troubles described in the introduction. Unfortunately, their explicit duration approach is insensitive to the amount of reasoning and is quite unsuitable for research using multiple methods or anytime algorithms.[3] The explicit duration approach in the DVMT is better, since it admits multiple methods, but it can't measure computation other than the KSI executions, such as control overhead. TÆMS has the most flexible model of duration, but it lacks the ability to simulate an agent really doing a task.

I believe these troubles can be fixed by using TCL to define an agent's thinking, since it can advance the simulation clock in a controlled and repeatable way. TCL is extensible, allowing new timing constructs to be added if desired. Since researchers will want to experiment with different agent architectures (one of the difficulties with architecture-specific testbeds such as TILEWORLD or the DVMT), the ability to modify the agent language will be important.

Also, TCL creates a database of durations for cognitive actions, allowing introspection by the agent. This introspection ability, seemingly crucial for real-time planning, has been left out of all these simulators.

Finally, all of these simulators (with the exception of TÆMS) make some kind of commitment to a domain. With some, such as PHOENIX, the domain is quite complex and elaborate, while with others, such as MICE, the domain is simple but malleable by the researcher. MESS goes one step further in the direction of MICE, eliminating the domain commitment and instead providing tools for building up a domain on a substrate of events and event streams. This leaves some work for researchers using this substrate, but avoids a commitment that they may be loath to make.

[3]Recall that they model the duration of "deliberation" as 1780 milliseconds, regardless of how much thinking actually takes place during that cycle.

Chapter 3

Design

This section describes the design of the simulation substrate I have called MESS, for "Multiple Event Stream Simulator." Although the most interesting and novel part of MESS is the integration of MESS with the agent language, TCL, to start the description there invites confusion. Therefore, the description starts from a global perspective, discussing discrete-event simulation in general. Then, we look at the architecture of MESS and its components: events, event streams, and the engine. This will lead to several examples of modeling using MESS. With this grounding, we look at how to model thinking processes using MESS and TCL and how to model the interaction of processes.

3.1 Simulation Models

Processes and activities in the real world occur continuously and concurrently. For example, fires really burn while firefighters are trying to stop them. Computers, of course, do only one thing at a time and therefore a computer simulation of world processes must model each process as a stream of discrete, timestamped events. The simulation then interleaves these event streams so that the events happen in the correct order, yielding an approximation of the real-world processes. This is called discrete-event simulation.

MESS makes no commitment to a domain but instead supplies the materials to build any domain, namely *events* and *event streams.* For example, the ignition of a firecell is an event in PHOENIX, the appearance of a tile is a TILEWORLD event, and a train traversing a route is an event in TRAINS. Events are defined in MESS using the object-oriented programming extension to Common Lisp called CLOS [30,57], where the user supplies code that determines when the event occurs and how it modifies the representation of the world. The "how" code is the *realization* method of the event, and executing that code is called *realizing* the event. The hierarchy of event classes can be used to group kinds of events, such as all the movement events or all the fire events, so that they can be controlled and modified as a group.

The simulation literature describes two standard ways to structure a discrete event simulator [4, p. 13]:

event orientation Each event determines what subsequent events follow from it (that it "causes"). Thus, each event type has some kind of "successors" function.

process orientation Each event is part of a process, and that process determines the subsequent events.

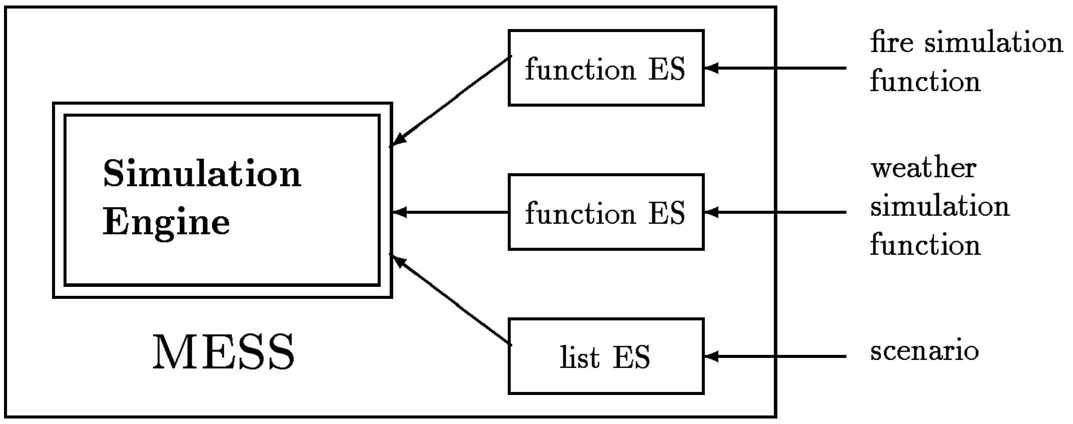


Figure 3.1. The architecture of the MESS simulation substrate. MESS is structured as a central engine, driving instances of different kinds of event stream (ES). MESS itself is domain independent; the streams listed at the right (weather, fire, scenario) are examples drawn from the PHOENIX domain.

MESS is process-oriented, because it is convenient to view as processes things like fire, weather, and particularly an agent's thinking. The representations of processes are called *event streams.* Event streams are also defined using object-oriented pclos/, so that users can add other kinds of event streams if they need a particular way of producing events.

By this software design methodology, I accomplish two things. First, I provide a common underpinning for AI simulators, which promotes sharing and collaboration among researchers. It's too much to hope that existing simulators will be re-implemented atop MESS, but new simulators will always be necessary.[1] If MESS makes implementing a simulator easier, it will become more widely used, and, gradually, a collection of classes of events and event streams will accumulate, further reducing implementation effort. Second, by implementing the agent language—which is the heart of the thesis—in a domain-independent substrate, the language is usable by more researchers than if it were tied to a domain-dependent simulator.

3.2 Architecture

Figure 3.1 shows the structure of MESS. The simulator has a central "engine," which interleaves the streams of events that represent different real-world processes. These events are drawn from and generated by "event streams" of various kinds. A very general kind of event stream (ES) is a *function* ES, where a function produces the next event upon demand. Another kind of ES is a *list* ES, which produces a pre-defined sequence of events. The MESS engine controls *instances* of these kinds of event streams, one instance for each world process. The following sections will describe these aspects of the MESS architecture.

[1]Indeed, MESS is being used for the Air Campaign simulator in the ARPA/Rome Laboratory Planning Initiative.

```
Algorithm to **Advance** the simulation:
    increment event counter
    advance time by head of PEL
    If head of PEL is an ES
        Set ES to head of PEL
        Peek ES
        Set E to event in ES
    else
        Set ES to nil and set E to head of PEL
    Check Interactions
    Realize E
    Illustrate E (optional)
    Unless ES = Nil
        Pop ES
    Do Every Event Stuff
    Check Wakeup Time Functions
    Write out E (optional)
    Change Activity
```

Figure 3.2. Pseudo-code for the MESS engine. The steps of the algorithm are explained in the text.

3.2.1 Engine

The MESS engine is so called because it's where everything happens. It controls all the events and event streams, and it invokes the realization of events. Discrete event simulators go from state to state in discrete steps, which I have called *advancing* the simulation. Figure 3.2 presents pseudo-code for the algorithm to advance the simulation. Each time the simulation is advanced, exactly one event is realized.

The event to be realized is whatever event is nearest in the future. In a queuing simulation, if we have a customer arrival scheduled for time 18 and a departure scheduled for time 13, the departure must obviously come before the arrival. These scheduled events are kept in a data structure variously called the "pending event list," "dynamic event list" or some such term. We will use the former term, abbreviated PEL. The exact representation used for the PEL is not important here; you may think of it as a totally ordered list of events. When an event is scheduled, it is inserted into the PEL in the correct place; when the simulation is advanced, the first event in the PEL is realized and removed from the list.

In MESS, there can be two kinds of object in the PEL: an event or an event stream (ES). In some ways, described below, an ES can be treated just like an event, because it always has a particular event that is the next event in the stream. If we think of an event as a sheet of paper, an ES is like a pad of paper: it has a bunch of sheets, only one of which shows at a time. The PEL in MESS contains either individual events (sheets of paper), or event streams (pads of paper). In practice, most of the objects in a MESS PEL are event streams, as the examples in later sections will show.

Let's look at the steps of the algorithm in order:

Event Counter MESS keeps a count of events that have been realized. This has no causal significance, but it can be convenient to keep track of what MESS is doing.

Time Advance The first interesting thing is to advance the time to the timestamp of the event at the head of the PEL. If the previous event was at time 13 and the next event is at time 18, the time changes atomically from 13 to 18.

Get Head of PEL Set two temporary variables based on the object at the head of the PEL. If it's a bare event, set the variable E to the event and set the ES variable to "nil." If the object is an ES, set the ES variable to the object, and set the variable E to the event that is on the top of that event stream.

In MESS, looking at the event on the top of the ES is called "peeking." To use our pad-of-paper metaphor, the sheet of paper stays attached to the pad, but the engine looks at the top sheet (the pending event). The operation of peeking at an ES depends on the kind of ES. Some kinds of ES will already have the pending event prepared, while others will delay preparing the event until the engine demands it.

Interactions Previously realized events may have started "activities" which represent continuous processes that have definite start and stop events, such as an agent's movement from A to B. Events that occur between the activity's start and stop may interact with it. For example, a surface-to-air missile site might interact with an airplane's movement.

Activities are placed on a list when they start and are removed from the list when they finish. The engine checks this list with each event, to see if the event interacts with any activity on the list. The engine checks for interactions before realizing the event, in case the activity prevents the realization of the event. (Activities are explained more fully in section 3.5, below.)

Realization Finally, the event is realized. Exactly what happens at this point is up to the event, because event realization is domain-dependent. Essentially, the MESS engine tells the event to modify the state of the world as it sees fit. Movement events will change the location of agents, weather events will change the conditions of roads, and so forth. An event's semantics may be altered or even canceled by interactions with current activities, which is why this step follows the interactions step.

Illustration Next, the event is illustrated. Again, the exact semantics of the operation depends on the event. Movement events may be illustrated by modifying the position of an icon on the user's screen, weather events may darken those regions that have been rained on, and so forth. The illustrate operation of the engine is controlled by a global parameter, so all such operations can be turned off at a stroke when they aren't wanted.

Popping the ES If the object in the PEL was an ES, "popping" it means to remove the event from the stream (tearing off the top sheet of paper). As with peeking, the popping operation depends on the kind of ES. The result of the popping determines, at least, the new timestamp of the ES, which resolves where it will be re-inserted in the PEL. Some event streams will prepare the next event right then (the timestamp of that event is the timestamp of the ES); other event streams will generate only a timestamp and defer the decision about the next event in the ES.

Every Event After every event, the engine executes a list of arbitrary functions. The user can put functions on this list, so this is a convenient way of collecting certain kinds of time-series data. (Time-series data are data collected over time during a trial.) For example, a queueing simulation might collect data on queue lengths after each event.

Wakeup Time An ES sometimes has nothing to do—there will be no events until something about the world changes. For example, in a queueing simulation, a departure process has nothing to do if there are no customers in the system. In a forest fire simulation, the fire process is dormant until some outside event starts a fire. When an ES has nothing to do, an ES can ask MESS to put it to sleep, supplying a function saying when to wake the ES back up. MESS runs these "wakeup time" functions after every event, since the event may have changed the world such that the ES should wake up.

Write Out Event A user may want to keep detailed records of exactly what happened during a simulation. If so, each event can be written out to a file for permanent storage.

Change Activity In order to make the implementation of activities convenient for users, the pair of events (starting and stopping the activity) are represented using a single object. That object has a switch in it, controlling whether it behaves like the start event or the stop event. If, during this advance of the simulation, the E event was the start of an activity, it is now switched to being the stop event of that activity. (It's completely realized now, so its usefulness as a start event is over.)

The preceding description comprises the essentials of the MESS engine, hence how a MESS simulation runs. MESS simulations are based on simply iterating the "advance" algorithm. For example, figure 3.3 shows how to run a simulation for some number of simulation minutes. In other cases, one might advance the simulation until some condition is true, such as the queue being empty or the forest fire put out. Such a simulation just puts a different stopping criterion in the loop.

The rest of a MESS simulation involves events and event streams. Since events are entirely domain-dependent, there is little more to say about them, though we will see some simple examples later. Therefore, we turn now to event streams.

```
Algorithm to **Run for Time Interval** T:
        set END to *current time* + T
        loop while *current time* < END
                **Advance** Simulation
        end loop
```

Figure 3.3. Algorithm to run a MESS simulation for some amount of time.

3.2.2 Event Streams

As we saw in the last section, an ES is passive, producing events when the engine instructs it to do so. The engine knows which ES to get the next event from because each ES has a timestamp, the timestamp of the next event in the stream.

MESS has only five kinds of event stream, but the user can implement additional kinds as long as they obey the right protocol. This protocol is essentially what we saw in the engine's use of the ES: the ES must have a timestamp, so that it can be correctly placed in the queue; it must produce an event when "peeked"; and it must either produce a new timestamp or go to sleep when "popped."

Examples are usually easier to understand than written descriptions, so MESS comes with many "demonstration" systems: toy simulations implemented to show various features of MESS. These examples are described in the next section. The remainder of this section will sketch the kinds of event stream that MESS provides.

List ES This kind of ES has a predefined list of events in it, sorted by their timestamp. These events are provided by the user before the simulation begins. The timestamp of the ES is simply the timestamp of the event at the head of the list. When the ES is "peeked," that head event is returned. When the ES is "popped," that head event is removed from the list and the timestamp of the ES changes to the timestamp of the new head of the list. When the list is exhausted, the ES is removed from the simulation.

Clearly, such an ES is very limited. The timing of the events cannot be sensitive to the evolving state of the simulation, although their realization methods could modify their behavior as a function of the state. List event streams can be useful to define the experimental conditions of a simulation: start a fire at this time, change the wind at that time, disable a vehicle at a third time, and stop the simulation at a fourth time.

Pending Function ES Unlike the List ES, *function* event streams have a great deal of flexibility. The simulation implementer provides a function that MESS calls whenever it "pops" the ES. That function calculates what the next event in the ES will be, and the event is stored in the ES. The peek operation simply returns this *pending* event (hence the name). The timestamp of the ES is the timestamp of the pending event.

Immediate Function ES This kind of ES runs an implementer-defined function whenever the peek operation is called, rather than the pop operation. It doesn't seem like this would make much difference, since the peek is before realization and the pop is after, but it can make a big difference. Let's take an example. Suppose that the ES has an event at time 25, so the pop operation happens then. Its next time is, say, 315, which is when the peek operation happens. Any number of events can happen between 25 and 315; the world might be a very different place by the time the peek happens. A Pending Function ES commits to the identity of the next event at time 25, while the Immediate Function ES doesn't commit to the event until 315. At 315, the ES is peeked, the event is created and given to the MESS engine, which immediately realizes it (hence the name).

The preceding event streams are sufficient for implementing many kinds of simulation. Below, I describe kinds of events stream specifically designed for real-time agents. The two work very similarly, differing only in whether CPU time or TCL is used to measure duration. Obviously, I believe the TCL ES should be used instead of the CPU time ES, but by implementing both kinds of ES, I have a minimal pair for purposes of experimentation. That is, I can compare the CPU ES to the TCL ES to show the differences. These experiments are discussed in section 4.1.2.

The CPU time and TCL event streams are subclasses of *thinking* event streams. A thinking ES is quite different from the previous kinds of event stream, even though it obeys the same protocol. That's because it is in a *co-routine* relationship with the engine. (This is quite common in process-oriented simulations [4, p.15].)

Co-routines are parallel threads of control that explicitly take turns executing. Contrast this with "time-slicing," which most people are familiar with, even if they don't know the name. The idea of time-slicing is that the computer runs Alice's process for a short time (say a few milliseconds) and then runs Bob's process for a short time, and then runs Alice's again, and so forth. Thus, Alice and Bob have the illusion of continuous attention by the computer, albeit one that's a little slower than the real one. The necessary condition is that the computer must be able to save the state of the two processes so that it can pick up execution exactly where it left off. Co-routines work the same way, except that instead of the computer stopping and starting each process, they explicitly say "It's her turn" or "It's his turn," to pass the CPU to the other process.[2]

The important point here is that when Alice gives Bob the CPU and, later, gets it back again, she picks up her computation exactly where she left off, rather than starting over from scratch. This point is illustrated in figure 3.4.

In MESS, a thinking event stream co-routines with the engine. That is, the engine lets the ES have its turn when it needs to get the next event from that ES, and the ES runs until it computes an event, whereupon it returns control to the engine. To be precise, a Thinking ES is like a Pending Function ES, because the ES gets its turn when the ES is popped, and when it computes an event, the event becomes the

[2]To be precise, the co-routining does not replace time-slicing but supplements it. The operating system still time-slices everything (Alice, Bob, and, Carla), but it executes a process only when it is that process's turn. Thus, if Alice and Bob are co-routining, and Carla is not involved, and it's Alice's turn, the time-slicing is between Alice and Carla, while Bob waits for Alice to give him his turn.

```
Algorithm For **Alice**:
    loop for i from 1 to 7
        print "Alice" i
        if ($i$ mod 3) = 0
            change-turn(Bob)
    end loop

Algorithm For **Bob**:
    loop for i from 1 to 7
        print "Bob" i
        if ($i$ mod 3) = 0
            change-turn(Alice)
    end loop
```

```
Alice 1
Alice 2
Alice 3
Bob 1
Bob 2
Bob 3
Alice 4
Alice 5
Alice 6
Bob 4
Bob 5
Bob 6
Alice 7
Bob 7
```

Figure 3.4. A simple example of co-routining. The two panels at left give the algorithms that Alice and Bob execute. The right panel gives a transcript of executing the co-routines, assuming that Alice goes first.

pending event in the ES. As before, the timestamp on the pending event determines when the event is realized and when the ES runs again.

How is the timestamp on the pending event calculated? Note that this is not a question we have asked before. We assume that function event streams compute the timestamp in domain-specific ways, involving, for example, models of how fast vehicles move or fire spreads. With a thinking ES, we want the timestamp on the event to be determined by the *amount of computation* that has occurred during this turn. For instance, the amount of computation between Alice's turns (see figure 3.4) is three loop iterations, each of which has a print statement, a computation of i mod 5, an equality test, and so forth. That is, the computation of the timestamp on the next event in the Alice agent is a side-effect of Alice getting her turn to think: she thinks until she does an event, and the amount of thinking determines the timestamp of the event.

Up to this point, the CPU time and TCL thinking streams have been the same, but it is in determining the amount of computation during a turn that they differ:

CPU time At the beginning of its turn, a CPU time event stream reads and records the current CPU time of the Lisp system. At the end of its turn, it reads the current CPU time again, subtracts the time from the beginning of its turn, and multiplies the difference by a constant called the "real-time knob" to get the duration of the turn. The duration of the turn, together with the time that the turn started, determines the timestamp of the pending event.

Here's an example. The thinking begins at simulation time 800, which is in seconds. The ES reads the CPU time, which reports 12,345,000. (CPU times are usually reported in some arbitrary unit that is convenient to the manufacturer of the computer or operating system. Let's suppose that they are milliseconds.) At the end of the turn, the CPU time is read again, returning 12,345,215. So, 215 CPU milliseconds have passed. Suppose the real-time knob is set to 1/5, meaning that 5 CPU milliseconds equals 1 simulation second. This means that the duration of this turn is 43 simulation seconds. Therefore, the pending event is scheduled to occur at time 843.

TCL At the beginning of its turn, a TCL event stream sets an internal counter to zero, called `*duration*`. As the computation of the turn progresses, TCL functions will increment this counter. To use the example from figure 3.4, the `print` function may increment the counter, the `mod` and equality functions may also, and so forth. At the end of the turn, let's suppose that the counter stands at 580. Let's also suppose that the real-time knob is set to 1/10, meaning that ten "duration units" equal 1 simulation second. Therefore, 58 simulation seconds have passed and the pending event is scheduled to occur at 858.

Clearly, these computations are not very different. There is a real-time knob that maps from computation units to simulation time units, and we just need to measure the amount of computation units that are consumed during the turn.

The difference, of course, is whether CPU time is measured. The effect of eliminating the measurement of CPU time is to give ourselves complete control over the measurement of time. In a TCL event stream, only functions that increment the counter take time, and they take exactly as much time as they are defined to take. There is no measurement error.

From the MESS engine's point of view, none of the preceding discussion of co-routines and measuring duration matters. All of that is hidden in the implementation of thinking event streams, and they obey the same protocol that the list and function event streams do.

The MESS functionality can be extended by defining new kinds of event stream, each obeying this protocol. For example, in the prototype Air Campaign Simulator of the ARPA/RL Planning Initiative, the implementers wanted an event stream that was a little more flexible than the List ES, but not as powerful as a Function ES. They wanted, in fact, a "Tree" ES, in which the user can specify a tree of events, with predicates (functions returning either true or false) at the nodes of the tree, to choose which branch to take. This was simple to implement in MESS: the "pop" operation returns the next event in the tree, unless the next thing is a predicate, in which case it is evaluated, and the next event is gotten from the selected branch.

3.3 MESS Examples

This section will describe a number of example simulations implemented using MESS. I have tried to minimize the amount of Lisp code presented, while still being specific about how MESS works.[3] Greater detail can be found in the MESS documentation and in the MESS "demo" code. These examples are also important because they build to the registrar experiment, which is described in the next chapter (section 4.1.2).

[3]Lisp uses a fully parenthesized prefix notation, so the first thing within parentheses is the name of the function and the other things are arguments to the function. Some of the arguments may themselves be function calls, in which case they are in parentheses.

3.3.1 Lives

The first example of MESS, called "lives," is very simple—so simple, in fact, that there is really little need for a discrete event simulator. However, the example lets us introduce many of the MESS language constructs without worrying about complications.

The example takes the sequence of events in two people's lives and makes an event stream of each sequence. That is, one event stream represents one person (Scott) and the other event stream represents the other person (Holly). The engine properly interleaves the realization of the events, thereby "simulating" the two lives and their interaction. In this example, the realization of a life event merely prints out some text. It's all very simple, but it demonstrates that event streams are supposed to represent world processes and the engine simulates the world by realizing these events in the correct order, allowing interactions to take place, and so forth.

We first define a "text event" which is an event whose realization just prints out a string identifying the event. (The `task-format` function is a MESS function that prints its argument, preceded by the current time and the name of the ES.) Events are classes that need to define two methods: `realize` and `illustrate`. Therefore, MESS supplies the `define-event` macro to help the user remember to do that, although in this case the `illustrate` method is empty.

```
(define-event text-event (event)
  ((text :initarg :text))
  (realize
    (task-format "text event:  ~a" (slot-value self 'text)))
  (illustrate))
```

Next, we need to define an event stream to produce these events. Because the events are pre-defined, rather than computed at run time, the `list-event-stream` is the most appropriate type. The following defines the `scott` event stream, producing the events in Scott's life. The other event stream is defined similarly.

```
(define-list-event-stream scott
  (text-event "12/4/1961" :text "Scott born")
  (text-event "9/1/1966" :text "Scott enters 1st grade")
  (text-event "6/13/1979" :text "Scott graduates from high school")
  (text-event "5/22/1983" :text "Scott graduates from college")
  (text-event "5/30/1988" :text "Scott receives M.S.")
  (text-event "4/11/1992 14:30" :text "Scott marries Holly")
  (text-event "11/11/1994" :text "Scott defends thesis proposal"))
```

Once the two event streams are defined, we can define a function to run the simulation. The following function makes a simulation, using the two event streams named `scott` and `holly`, resets the simulation (which creates the initial state of the world and gets the simulation ready to run) and runs it to the finish time, which in this case is 6/1/95.

```
12/4/1961  0:00 SCOTT text event:  Scott born
 9/1/1966  0:00 SCOTT text event:  Scott enters 1st grade
 8/4/1967  0:00 HOLLY text event:  Holly born
 9/1/1973  0:00 HOLLY text event:  Holly enters 1st grade
6/13/1979  0:00 SCOTT text event:  Scott graduates from high school
5/22/1983  0:00 SCOTT text event:  Scott graduates from college
6/1/1985   0:00 HOLLY text event:  Holly graduates from high school
5/30/1988  0:00 SCOTT text event:  Scott receives M.S.
5/22/1990  0:00 HOLLY text event:  Holly graduates from college
4/11/1992 14:30 HOLLY text event:  Holly marries Scott
4/11/1992 14:30 SCOTT text event:  Scott marries Holly
9/5/1994   0:00 HOLLY text event:  Holly starts law school.
11/11/1994 0:00 SCOTT text event:  Scott defends thesis proposal
```

Figure 3.5. A transcript of the "lives" simulation.

```
(defun show-lives ()
  (run-forever
    (reset (make-simulation
             'lives-sim 'simulation-state "6/1/1995"
             :stream-specs '(scott holly)))))
```

Figure 3.5 shows a transcript of the simulation; you can see that it's just the interleaving of the events in the two streams.

3.3.2 Clocks

Like the previous "lives" demonstration, the "clocks" demonstration is also small. It implements a set of clocks, each of which has a different length cycle between rings. The clocks are also distinguished by their rings: some are little chimes and others are big booms. The primary difference between "clocks" and "lives" is that the clocks simulation uses pending function event streams to produce potentially infinite streams of events.

A function event stream requires the user to provide a function that will be called whenever a new event is needed. For a *pending* function event stream, a new event is generated right after the previous event in the stream is realized. The event stream and the event are put back into a priority queue (kept in the engine) and the timestamp of this new event determines when it gets realized (and the next event generated).

I have omitted the `define-event` form for this example; it's similar to the one for the preceding example, differing only in formatting the output based on the kind of ring the clock has. The function that will generate the pending event is called `next-chime` and it takes two arguments: the type of the ring and the time-interval between rings.

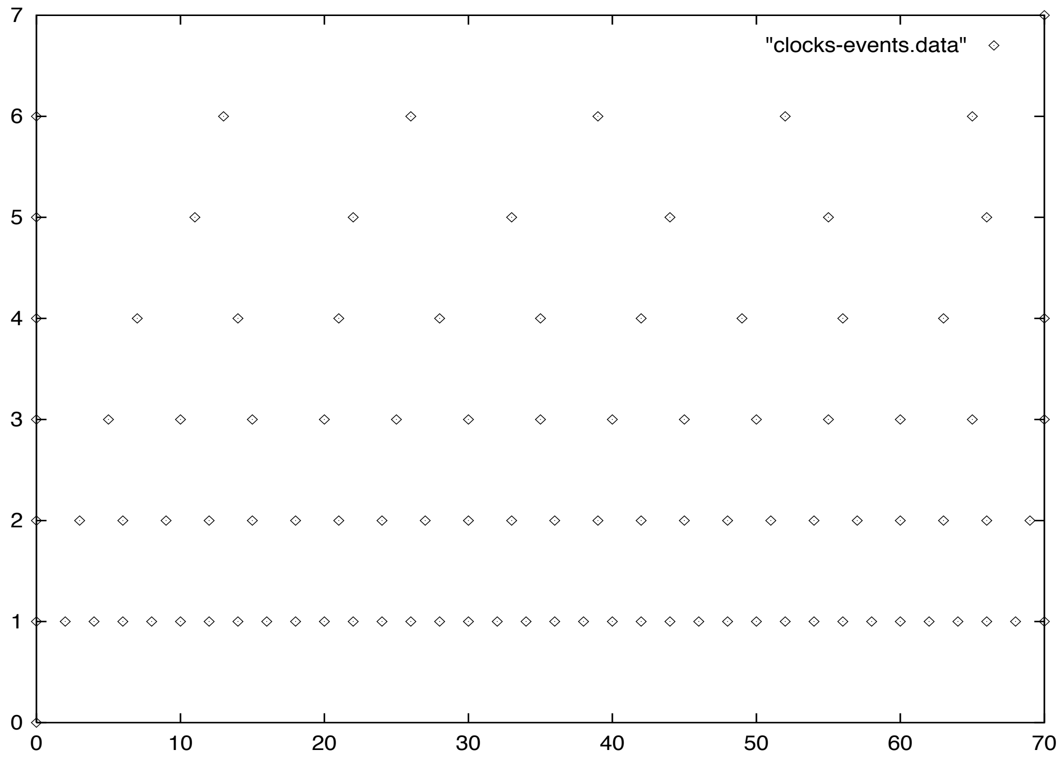


Figure 3.6. The events from a "clocks" simulation. The horizontal axis is time, and the events have been partitioned along the vertical axis by the event stream they came from.

```
(defun next-chime (type interval)
  (make-event 'clock-chime
              (+ (this-time) interval) (this-event-stream)
              :chime-type type))
```

The `make-event` function creates an event object for MESS to schedule and realize. The function's first argument is the type of event (the simulator implementer must define these), the second argument is the time that the event must occur, the third argument is the source of the event (this feature is primarily for accountability, so that the user can know where an event came from).

The event streams for this demonstration are defined using the `next-chime` function, supplying different arguments for different streams, as follows. Thereafter, the engine will always call this function when it "pops" the ES.

```
(define-pending-function-event-stream ching 0 #'next-chime :ching 2)
(define-pending-function-event-stream  ding 0 #'next-chime :ding  3)
(define-pending-function-event-stream  ring 0 #'next-chime :ring  5)
(define-pending-function-event-stream  dong 0 #'next-chime :dong  7)
(define-pending-function-event-stream  bong 0 #'next-chime :bong 11)
(define-pending-function-event-stream  gong 0 #'next-chime :gong 13)
```

Figure 3.6 shows the first few events of a sample run that was ended at time 71.

MESS provides software to easily produce graphs like this, which can aid in debugging and understanding simulations.

Periodic event streams can be used for periodic data collection during a simulation, or for monitoring some condition, and so forth. Furthermore, the time of the next event is calculated by the sum of **(this-time)**, which is the current time, and **interval**. The interval could easily be non-constant. Indeed, it can be stochastic. Our next example will demonstrate this.

3.3.3 M/M/1 Queue

The simplest queueing system has a single queue, where customers arrive with some distribution of times, receive service with some distribution of times, and depart. If the arrival times and service times are exponential, this system is known as an M/M/1 queue. Our next example is such a system. We will model this system with two event streams, one for arrivals and one for departures. (There are other, more efficient ways of modeling an M/M/1 queue, but those can't easily be transformed into the more complex examples to be described later.)

One aspect of this example is getting a source of random numbers. MESS implements its own portable random number generators, and these generators yield the same results on different platforms.[4] Each event stream has its own random number generator; this is so that the behavior of different event streams will be independent, or, rather, interact only through the simulation and not because they are both drawing from the same random number generator. This is all hidden from the user, but the user must call the MESS generators, such as **uniform-random**, as shown below, rather than the Common Lisp **random** function. The user may specify "seeds" that are used to initialize the random number generator in a stream, so that random numbers can be replicated.

Both the event streams are implemented as Immediate Function event streams, which means that they generate an event just before it's realized. The "arrival" event stream is fairly easy to implement. Its code is in figure 3.7. The arrival rate is controlled by a parameter, so it's easy to modify that aspect of the simulation. Note how the function calculates the timestamp of the next arrival, by getting an exponential random number, rounding it up to an integer and coercing it to be of the correct type for a simulation time. The "departure" event stream (figure 3.8) is similar, but it can't calculate the next departure time if the customer departing is the last one in the queue. In that case, it sets its *status* to "waiting," which signals the engine to put the ES to sleep. (The ES has previously specified a wakeup-time function, which will awaken the ES when a customer arrives.)

Figure 3.9 shows the events from one run of the simulation, with the random number seeds set to 2 and 3. Different kinds of events are labelled with different

[4]The random number algorithms have been adapted from Numerical Recipes in C [48] and therefore are trustworthy. The Numerical Recipes algorithms used 32-bit integers for the calculations, but Lisp implementations typically only have 29-bit integers. Therefore, MESS currently uses double-floats in the computation, since those have 53-bit mantissas in our Lisp implementations. If those double-float routines prove to be a problem in some Common Lisp implementation, MESS also has generators using exact arbitrary-precision integer arithmetic (bignums) that are completely portable, albeit not as efficient as double-floats.

```
(defparameter *arrival-rate* .1
  "The parameter of an exponential distribution controlling the
arrival process.  Its reciprocal is the mean inter-arrival time.")

(defun next-arrival ()
  (setf (timestamp (this-event-stream))
        (+ (this-time)
           (ceiling (exponential-random *arrival-rate*))))
  (make-event 'arrival (this-time) (this-event-stream)))
```

Figure 3.7. The "next event" function for the Arrival event stream.

```
(defparameter *service-rate* .2
  "The parameter of an exponential distribution controlling the
service process.  Its reciprocal is the mean inter-departure time.")

(defun next-departure ()
  "Called to make the departure event, which it does last because that
value must be returned.  It is called at the time of the departure, so
the event's timestamp is always (this-time).  It must calculate the
time of the next departure, for rescheduling purposes, or it must wait
for an arrival."
  (if (= (queue-length (this-simulation-state)) 1)
      (setf (status (this-event-stream)) :waiting)
      (setf (timestamp (this-event-stream))
            (+ (this-time)
               (ceiling (exponential-random *service-rate*)))))
  (make-event 'departure (this-time) (this-event-stream)))
```

Figure 3.8. The "next event" function for the Departure event stream.

letters, as shown in the legend. The "C" event indicates when the departure ES puts itself to sleep. Figure 3.10 shows the history of the queue length for that same simulation. Note how the "C" events correspond to the queue length dropping to zero.

3.3.4 Registrar

The next example is a variation on the M/M/1 queue, except that instead of exponential service times, we will assume that a "registrar" agent is serving the customers, using some deterministic, constant time algorithm. (Thus, the system is an M/D/1 queue.) Therefore, we replace the Immediate Function ES with a Thinking ES, either a CPU-time or a TCL thinking stream. Actually, we will do both, so as to contrast their implementation. Section 4.1.2 will show the contrast in their behavior.

Figure 3.11 displays a side-by-side comparison of the two definitions. (Some irrelevant code has been omitted, such as the "wakeup-time" functions. These are

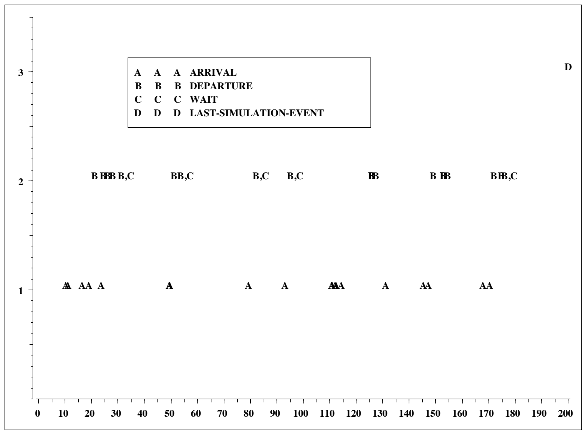


Figure 3.9. The events from the “random walk” simulation.

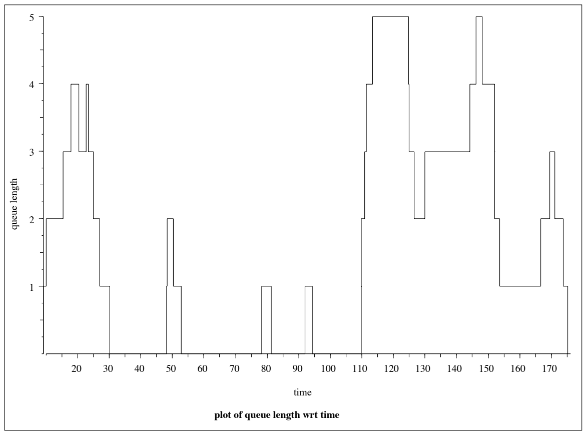


Figure 3.10. The queue length in a simulation of an M/M/1 queue.

```
(define-event-stream reg-cpu
  cpu-event-stream
  :es-function 'reg-cpu-thought-process
  :real-time-knob *reg-cpu-rtk*)

(defun registrar-cpu-thought-process ()
  (loop-forever
    (if (zerop
          (queue-length
             (this-simulation-state)))
        (make-event-during-thinking 'wait)
        (progn
          (cl-user:arbitrary-thought-stuff)
          (make-event-during-thinking
             'departure)))))

(in-package :cl-user)

(defun arbitrary-thought-stuff ()
  (loop repeat 3 do
        (sort *big-array* #'>)
        (sort *big-array* #'<)))
```

```
(define-event-stream reg-tcl
  tcl-event-stream
  :es-function 'reg-tcl-thought-process
  :real-time-knob *reg-tcl-rtk*)

(defun registrar-tcl-thought-process ()
  (loop-forever
    (if (zerop
          (queue-length
             (this-simulation-state)))
        (make-event-during-thinking 'wait)
        (progn
          (tclu:arbitrary-thought-stuff)
          (make-event-during-thinking
             'departure)))))

(in-package :tclu)

(defun arbitrary-thought-stuff ()
  (loop repeat 3 do
        (sort *big-array* #'>)
        (sort *big-array* #'<)))
```

Figure 3.11. Contrasting the implementation of CPU and TCL thinking streams. The code on the left defines the registrar that calculates duration using CPU time, while the code on the right calculates duration using TCL. They are completely parallel except that for the TCL agent, its thinking is defined in the `tclu` package. These differences have been highlighted with boxes.

identical, anyhow.) They are quite similar, with the primary difference being the `in-package` form preceding the definition of `arbitrary-thought-stuff` for the TCL agent. The next section will discuss the implementation of TCL and the implications of that `in-package` form.

The first forms, `define-event-stream`, define a name and a type for the agents. The form also specifies a function that will be the co-routining process; this is the `-thought-process` function. Finally, it specifies a value for the event stream's real-time knob.

The next form defines the `-thought-process` function. As will often be the case, the thought process is an infinite loop that computes and, every so often, calls `make-event-during-thinking` (MEDT). Of course, the thought process doesn't have to strictly follow this structure; any computation that occasionally calls MEDT, so that other event streams can take their turn, will be suitable as a thinking process.

Agents should use the MEDT function any time they want to interact with the world. This is because using MEDT gives control back to the engine and thereby to other event streams. Recall that the agent co-routine runs when the engine "pops" the thinking ES that has that co-routine. So, if the "pop" happens at time 800, the co-routine computes for a while and calls MEDT, which returns a pending event at time 858. Nothing else has gotten to run while the co-routine is running. If a movement event is scheduled for time 825, it is still pending. Control is given back to the engine so that the other events that should happen between 800 and 858 can happen, before realizing the agent's next event. If the co-routine were to interact with the world directly, it would see the world as it is at time 800. On the other hand, suppose the agent's next event, generated using MEDT, is a sensory event; that event happens at time 858 and will see the world as it is at that time.

Returning to our registrar example, the bulk of the computation between calls to MEDT is used in `arbitrary-thought-stuff`, which is called from the `-thought-process` loop. In this simple example, the arbitrary thought stuff is silly: it just sorts a big array several times. In any realistic registrar implementation, there would be some interesting computation concerning the customer, but that is irrelevant for this example.

Even those unfamiliar with Lisp programming can see that implementing agents using TCL is only slightly different from CPU-time agents. The differences above are due to renaming (for clarity's sake) and using the TCL package. It's time to see what that difference means.

3.4 Timed Common Lisp

Earlier, in describing TCL event streams, I said there is a special counter, called `*duration*`, that accumulates "duration units," and that TCL functions increment that counter. The key idea of TCL is how that counter can be incremented in a straightforward, systematic manner.

Let's start with an example. Suppose we have a function in the Common Lisp language, such as the cosine function, `cos`, and we want it to have a duration of 100 "duration units." (Remember that these duration units will get multiplied by the event stream's real-time knob to yield simulation time units, so the actual units of duration are arbitrary. Their relative magnitude is all that matters. If we want `cos` to take ten times as long as the square root function, `sqrt`, we set the latter to take 10 duration units.) Let's call duration units "ticks." To make `cos` take 100 ticks, we could define a different function, named `tcl-cos`, in terms of `cos`:

```
(defun tcl-cos (x)
  (incf *duration* 100)
  (cos x))
```

The `tcl-cos` function increments the counter by 100 and then calls `cos` to do the actual computation. We have thus defined a kind of "wrapper" function for a Common Lisp function. All we have to do now is call `tcl-cos` wherever we would have called `cos`. One can similarly define TCL versions of `sin`, `tan`, `sqrt`, `exp`, and, in fact, any Common Lisp function, of which there are about 600; TCL has implemented all those definitions.

3.4.1 Packages

The names we have generated for the TCL wrapper functions are a little cumbersome to use. For example, the name of the addition function in Common Lisp is, not surprisingly, `+`, but it would be awful to have to type `tcl-+` to use the TCL version of `+`. We would prefer to have the `+` symbol mean the TCL version of addition. This problem has an immediate solution in Common Lisp, through the use of *packages.*

A symbol in Common Lisp is a little like a person, in that it has two names: a family name and given name. The family name of a symbol is its *package*, while the

```
(defun registrar-tcl-thought-process ()
  (loop-forever
    (if (zerop (queue-length (this-simulation-state)))
        (make-event-during-thinking 'wait)
        (progn
          (tclu:arbitrary-thought-stuff)
          (make-event-during-thinking 'departure)))))

(in-package :tclu)

(defun arbitrary-thought-stuff ()
  (loop repeat 3 do
        (sort *big-array* #'>)
        (sort *big-array* #'<)))
```

Figure 3.12. The Registrar's thought process.

given name is the name specified by the programmer. All the symbols of the Common Lisp language, such as **cos**, are in the **common-lisp** package. When a program is *in* or *uses* a package, the program may specify a symbol in that package by just its given name, omitting the package name. Thus, the following function defines the **sam** function to call the normal, Common Lisp **cos** function, since **cl-user** *uses* that package.

```
(in-package cl-user)

(defun sam (x)
   (cos x))
```

To refer to a symbol in another package, you must specify both names. The syntax is the package name, followed by either one or two colons, and the given name. Thus, the symbol **cl-user:sam** will always refer to the symbol above, regardless of what package the program is in.

There are other packages than the **common-lisp** and **cl-user** packages; indeed, users may create them for their own purposes. TCL creates a package called **timed-common-lisp**, which is analogous to the **common-lisp** package. So, we can now define our TCL version of the cosine function as follows:

```
(in-package timed-common-lisp)

(defun cos (x)
  (incf *duration* 100)
  (common-lisp:cos x))
```

The TCL package redefines all the Common Lisp functions, while retaining the original definitions, by using a different family name. To call the TCL version of these functions, we just have to *use* the TCL package. There is already such a package defined, called **timed-common-lisp-user**, which thankfully has a nickname of **tclu**.

Let's look again at the thinking functions from figure 3.11, shown in figure 3.12. We can see now that the `-thought-process` function refers explicitly to the `arbitrary-thought-stuff` function in the `tclu` package, by using the package name and given name. That function doesn't seem to modify the `*duration*` variable, but, in fact, the `sort` function is the TCL version of the `sort` function, which increments the clock by the amount appropriate for `sort`. This is why porting agent code to TCL is so trivial: the language looks just like normal Common Lisp.

3.4.2 Operations

By how much should a function increment the clock? It depends on the function, of course, but we can group them into categories. For some, like `cos`, a constant time duration model seems correct. In the notation of complexity theory, which studies how long algorithms take, these functions would be described as $O(1)$.[5] The complexity of other functions depends on their inputs. For example, the `+` function takes any number of arguments and returns their sum, so the duration might reasonably depend on the number of arguments.

To be precise, the number of *operations* performed by the `+` function depends on the number of arguments. The duration is the product of the number of operations and a constant stored in the TCL database. Modeling duration as an operation count and a constant mirrors the analyses from complexity theory, which separates the complexity class of the algorithm from the proportionality constant that gives its actual, asymptotic duration on a particular machine, compiler, and so forth. The complexity of an algorithm is determined by choosing a "basic operation," and counting the number of such operations that are performed as a function of the inputs. For sorting, the basic operation could be the comparison of two elements, while for a Fast Fourier Transform, the basic operation could be a floating point multiplication.

So, the example of `cos` given above was slightly incomplete. The TCL `cos` function is defined more like:

```
(declare-primitive cos :constant 100)

(defun cos (x)
  (call-book cos x)
  (common-lisp:cos x))
```

The `call-book` function does the bookkeeping for a call to the `cos` function with arguments `x`. The bookkeeping looks up the duration model for `cos` in a database

[5]This "big-oh" notation defines the complexity of an algorithm in a way that ignores minor variations on the essential nature of the algorithm. For example, suppose we analyze two sorting algorithms. One uses a pair of nested loops, in which the inner loop finds the smallest element of the items remaining and the outer loop puts that smallest element in the correct place. The other algorithm uses a fancy divide-and-conquer technique in which the items are divided into two equal halves, each of which is sorted, and the resulting sorted lists merged. If the number of elements is n, as n goes to infinity, the duration of the first sorting algorithm is proportional to n^2, while the duration of the second is proportional to $n \log n$. Ignoring the constants of proportionality, we say the first algorithm is $O(n^2)$ and the second is $O(n \log n)$. See Baase for an introduction to complexity theory and analysis of algorithms. [2]

and increments the clock based on the duration model. The duration model in this case is `:constant` and the constant to use is 100. It is the `declare-primitive` that adds new primitives to the TCL language, and it is available to users who want to extend TCL.

3.4.3 Duration Models

The following duration models are built-in as primitives of the TCL implementation. One advantage of the built-in primitives is that the bookkeeping function knows specially about them, so they execute faster, because they entail no function-call overhead.

`:constant` The function does one operation, so it always takes a constant duration, regardless of the number and nature of its arguments. An example is `cos`.

`:length` The number of operations is proportional to the length of the first argument, and so, in "big O" terms, equals its length. This model is used for many of the Common Lisp sequence functions, such as `find`, which traverses a list to find an element, or `butlast`, which copies all but the last element of a list.

`:arity` The number of operations is the number of arguments to the function (its *arity*). This model is used for only a few TCL functions, such as `list`, which constructs a list of its arguments, and `vector`, which constructs a vector of its arguments.

`:arity-1` The number of operations is one less than the function's arity. This model is used for a lot of arithmetic functions, such as `+`, `-`, `*` and so forth, because `(+ 1 2 3)` does *two* additions, not three. In general, to add n terms, the computer must do $n - 1$ additions, and similarly with subtractions and multiplications.

`:direct` The function's first argument is the number of operations it does. No Common Lisp functions quite meet this description, but this duration model is useful when *implementing* certain kinds of primitives, where the implementation of the primitive knows how many operations were performed and can call the bookkeeping function with that value. For example, the `tcl:length` function is defined as follows:

```
(declare-primitive length :direct 2)

(defun length (sequence)
  (let ((val (common-lisp:length sequence)))
    (call-book length val)
    val))
```

What this means is that the Common Lisp `length` function is called on the argument, and that value is supplied to the bookkeeping function. The bookkeeping code looks up the duration model, `:direct`, which means to take `val`, multiply it by 2, and add that to `*duration*`. Thus, the duration of the TCL length function is proportional to the length of the list it must traverse, which is only reasonable.

The more general purpose of the `:direct` model is for modeling the duration of a function where it is hard to cheaply compute the number of operations by looking at the arguments to the function (as all the preceding models have done), but it is straightforward to count the operations *while* the function is running. To do so, however, means reimplementing the function. Take the Common Lisp function `tree-equal` as an example. This function takes two trees and recursively traverses them until it finds a difference, whereupon it returns false; if it finds no difference, it returns true after finishing the traversals. An operation in this algorithm is comparing two tree nodes. To count the operations during the tree traversal is easy, but to do so beforehand is equivalent to traversing the tree again. Tree traversal is far too expensive to do twice. Therefore, TCL re-implements the Common Lisp `tree-equal` function, rather than calling the Common Lisp version, but the TCL version counts operations as it goes. It reports those operations by using `call-book`, which knows that its argument is not something that must be converted into an operation count, as with `:length` or `:arity`, but is already an operation count (directly, hence the name of the `:direct` duration model). `Call-book` then multiplies the operation count by the proportionality constant in the database to produce a duration, just as with the other duration models.

We can now see the reason for using `:direct` with the `length` function, because the obvious alternative duration model is `:length`. The `:length` model would (1) measure the length of the list in order to count operations, and (2) call the Common Lisp `length` function to compute the real value of the TCL `length` function. Clearly, it's foolish to compute the length of the list twice, and so we use the `:direct` model because `length` happens to count the number of operations so conveniently.

It's easy to extend the set of duration models by defining a duration-model function and supplying its name instead of one of these primitives. Indeed, the TCL implementation does that in a number of cases, detailed below. For example, the duration of `concatenate` is implemented something like the following (certain confusing parts of the Lisp syntax have been elided).

```
(define-duration-model concatenate-operations (x y)
  (+ (length x) (length y)))

(declare-primitive concatenate 3)

(defun concatenate (x y)
  (call-book concatenate x y)
  (common-lisp:concatenate x y))
```

The duration model for `concatenate` accounts for the fact that it must copy both arguments in order to concatenate them, and so its duration should be proportional to the sum of their lengths. TCL defines sixteen other such duration models, so there are many examples for those who want to use this ability to extend TCL.

3.4.4 Free Operations

When implementing an agent whose thinking time should be "on the clock," we can implement it in the TCL package (or a package that uses it), and any function we refer to will increment the clock in some way. But suppose we don't want to increment the clock. Suppose that we are inserting some code to help in debugging the agent, to collect data, or to measure performance or quality, and we don't want that code to affect the timing behavior of the agent. That is, we want to let the code execute "for free." Let's call the code "insertion" code. Executing insertion code for free is quite straightforward in TCL, using one of two techniques.

The first technique is used if the insertion code does not increment the clock. For example, the following function will not increment the clock because it is defined in the `cl-user` package, and so it doesn't call any TCL functions:

```
(defun insertion (x)
  (format "The variable is ~s" x))
```

Since it doesn't advance the clock, it is already free, and so all we need to do is call it. Because the agent is defined in the TCL package, this will probably mean that we must specify the symbol's package name as well as its given name:

```
(in-package tclu)

(defun agent-function-5 (x y)
  ...
  (cl-user:insertion x)
  ...)
```

The problem is slightly harder if the insertion code was defined using TCL functions, since they will increment the clock. The solution is to wrap the call to the TCL function with a `free` form:

```
(in-package tclu)

(defun agent-function-3 (x)
  ...)

(defun agent-function-5 (x y)
  ...
  (free (agent-function-3 x))
  ...)
```

Under normal circumstances, calling `agent-function-3` would advance the clock by some amount, depending on its code. The `free` form lets the function go ahead and advance the clock, but then restores the clock to its original value. Note that the `free` form is for code that is not part of the agent's normal thinking, so `make-event-during-thinking` should not be called within its scope.

Of the two kinds of insertion code, using the `free` function is clearly less efficient, since the enclosed code does all the work of TCL only to throw the results away. Nevertheless, the `free` function is invaluable if the insertion code calls TCL primitives.

3.4.5 Code Walking

Looking up the duration model takes a little time, so for the sake of speed, TCL allows users to specify that the lookup should happen at compile time instead of run time. In that case, the code of the TCL version of **cos**, say, does indeed become a simple increment to ***duration***, as shown at the beginning of this section. The resulting speed advantage comes at the cost of having to recompile all the TCL code if the duration database changes. But we can do better, squeezing even more overhead out of TCL to make it faster, by a technique called *code-walking.*

Suppose we have a function that computes the hypotenuse of a right triangle.

```
(defun pyth (x y)
   (sqrt (+ (* x x) (* y y))))
```

Suppose, further, that we have the following duration models for the functions involved:

```
(declare-primitive + :arity-1 1)
(declare-primitive * :arity-1 10)
(declare-primitive sqrt :constant 100)
```

Executing the **pyth** function involves two calls to *****, one call to **+**, and one call to **sqrt**. Each of these calls results in a lookup of a duration model and an increment to ***duration***, for a total duration of 121 (1+10+10+100). Furthermore, each call results in a call to the Common Lisp function of the corresponding name. (That is, **tcl:sqrt** calls **cl:sqrt** and so forth.) Since we can figure out that 121 is the duration of **pyth**, so can TCL.

The idea of code-walking is that a function (called a code walker) analyzes the code of the **pyth** function, just as we did, and transforms it. The **pyth** example is the best case, because the code walker can transform it to the following:

```
(defun pyth (x y)
  (incf *duration* 121)
  (cl:sqrt (cl:+ (cl:* x x) (cl:* y y))))
```

The code-walker works recursively, analyzing each function and then the arguments of that function, until it gets to the atomic elements.[6] Along the way, it converts functions to their free, Common Lisp, equivalents, if possible. When it does so, it increments its own counter ("compile-time duration") by the duration of that function. Figure 3.13 is an actual transcript of the TCL code-walker working on the **pyth** function, in its "verbose" mode.

The first step, labeled 0, processes the **sqrt** function call, which increments the compile-time duration by 100. It then recursively calls the code-walker on the **+** call, which increments the compile-time duration by 1 and calls the code walker recursively on each argument—the two multiplies, each of which increments the compile-time duration by 10. The result is shown below: the **pyth** function now only increments the clock once instead of four times, for a substantial savings. We can't do better on this function.

The **pyth** function was a best case for the code-walker. Unfortunately, many things can get in its way. All have essentially the same character: the duration depends on information that won't be known until run-time. For example, the

[6]Lisp syntax is ideal for code-walkers, because the fully parenthesized notation makes it easy to find functions and their arguments.

```
TCLU 102 > (defun pyth (x y) (sqrt (+ (* x x) (* y y))))
 0> Calling 'TCL-CW-FUNCTION' on function call:
    (SQRT (+ (* X X) (* Y Y))) with model (:CONSTANT 100)
  1>SQRT (:CONSTANT) increments compile-time duration by 100
  1> Calling 'TCL-CW-FUNCTION' on function call:
     (+ (* X X) (* Y Y)) with model (:ARITY-1 1)
    2>+ (:ARITY-1) increments compile-time duration by 1
    2> Calling 'TCL-CW-FUNCTION' on function call:
       (* X X) with model (:ARITY-1 10)
      3>* (:ARITY-1) increments compile-time duration by 10
    2> Returned (COMMON-LISP:* X X)
    2> Calling 'TCL-CW-FUNCTION' on function call:
       (* Y Y) with model (:ARITY-1 10)
      3>* (:ARITY-1) increments compile-time duration by 10
    2> Returned (COMMON-LISP:* Y Y)
  1> Returned (COMMON-LISP:+ (COMMON-LISP:* X X) (COMMON-LISP:* Y Y))
 0> Returned (COMMON-LISP:SQRT
               (COMMON-LISP:+ (COMMON-LISP:* X X)
                              (COMMON-LISP:* Y Y)))
There were 4 increments, and code-walking saved 3 of them:  75.0%
PYTH

(DEFUN PYTH (X Y)
  (PROGN
    (MESS:INCR-DURATION 121)
    (COMMON-LISP:SQRT
     (COMMON-LISP:+ (COMMON-LISP:* X X) (COMMON-LISP:* Y Y)))))
```

Figure 3.13. A transcript of the TCL code-walker.

duration of the `length` function depends on the length of the list, which won't be known until the program runs. If a loop appears in the code, the code-walker typically won't know how many times the loop will be executed until run-time. The code walker can process the *body* of the loop, to try to maximize speed there, but it can't pull duration code out of the loop. The code-walker does analyze all the code, looking for whatever opportunities it can.[7]

[7]The TCL code-walker is reminiscent of Niehaus's computation of the irreducible program graph (see section 2.2), since both involve a depth-first traversal of the code in order to compute execution times. However, because Niehaus is interested in worst-case execution time, his algorithm can collapse over branches and loops by simply assuming the worst. The TCL code-walker is stopped by branches and loops because its purpose is to preserve the actual execution time of the code while speeding it up. The TCL code-walker could be easily modified to collapse over branches and loops, to compute worst-case execution time (or some similar function of the execution time).

3.4.6 Extending TCL

TCL currently implements every Common Lisp function, and any functions that can reasonably have a duration model have one. Thus, it can be treated entirely as Common Lisp. These default duration models are defined in a file delivered with TCL, so one way to modify TCL is to change these default duration models, either by changing the proportionality constant for a function or its type of duration model (from `:constant` to `:arity`, for instance).

Users will also want to extend the language, adding new primitives with particular duration models. This should be fairly straightforward, given only the description earlier in this section. To define a new primitive, we have to declare the primitive and its duration model, and we have to write the code for it. Here's an example from the implementation of PHOENIX:

```
(declare-primitive object->string :constant 20)

(defun object-to-string (x)
  (call-book object-to-string x)
  (eksl-utilities:object-to-string x))
```

In fact, the preceding pattern is so common that TCL provides a shortcut form that writes the function definition for the user. Using it, the preceding becomes:

```
(deff-primitive object-to-string eksl-utilities:object-to-string
   :constant 20)
```

3.4.7 Querying the Database

Agents will want access to the duration database, so that they can reason using the durations of various primitives. For example, a scheduler will need this information. This is easily done with the MESS function `primitive-duration`. The duration, of course, depends on the real-time knob of the agent, so that is the first argument of `primitive-duration`. Here are some examples:

```
(primitive-duration 1 'cos) => 100
(primitive-duration 2 'cos) => 200
(primitive-duration 1 '+ 3 4) => 1
(primitive-duration 1 '+ 3 4 5) => 2
```

The last example reminds us that, in general, the duration of a function may depend on its arguments. Here, the `+` function costs 1 tick for each operation, and the number of operations is one less than the number of arguments. Therefore, with the real-time knob set to 1, the first addition takes one simulation time unit, while the second takes two. In fact, the first two examples should, for consistency, have supplied the argument to `cos`. We could get away with omitting it because the duration model of `cos` was `:constant`, so the argument was not needed.

An agent will want to supply its own real-time knob setting, which can be easily done as follows.

```
(primitive-duration (real-time-knob (this-event-stream)) '+ 3 4)
```

The value of this call depends on the event stream (agent) and its real-time knob.

3.4.8 Call Counting

Given that a body of code takes some amount of time to run, users will often want a breakdown of that time, perhaps because they want to know why the code takes as long as it does. For this purpose, TCL provides a form that reports the duration of code, broken down by the TCL primitives that were called. Here is an example using the simple **pyth** function.

```
MESS 59 > (duration (:verbose t)
              (pyth (length '(1 2 3))
                    (length '(1 2 3 4))))
duration: 135
  TOTAL
DURATION | CALLS | OPERATIONS | OP-COST |   KIND   | FUNCTION-NAME |
  100    |   1   |     1      |   100   | CONSTANT |     SQRT      |
   14    |   2   |     7      |     2   |  DIRECT  |    LENGTH     |
    1    |   1   |     1      |     1   | ARITY-1  |       +       |
   20    |   2   |     2      |    10   | ARITY-1  |       *       |

Grand total of partitioned duration:  135
5.0
```

The **duration** form first prints out the total duration (121), then partitions it by the function name. Each row of the table gives, from right to left: the function name, the kind of duration model and the cost of each operation (these two are obtained from the duration database), the number of times each operation was performed and the number of times the function was called (these are collected at run time), and, finally, the product of the operation cost and the number of operations. The column total of the leftmost column is the grand total of the duration, and should equal the overall duration given earlier. Finally, the form returns the value of the computation (in this case, 5.0).

3.4.9 Summary

TCL is a complete programming language, duplicating all of the functionality of Common Lisp, while allowing each TCL function to advance the clock in a deterministic, controlled way. TCL has been used as the agent language for PHOENIX agents in the new implementation of PHOENIX, which is approximately 16,000 lines of code. This gives us good reason to believe that TCL works.

In addition, the language can be extended with new primitives, and users can trade off flexibility for speed by specifying various parameters.

3.5 Activities

Simulation involves more than just agents thinking; it also has agents and world processes acting over time, and these actions can interact. For example, one event stream can be a train moving along a track and another event stream might produce an event that destroys a section of the track. The nature of such interactions is,

```
(define-activity vacuuming (activity) ()
  (:documentation "Vacuuming activity")

  (activity-start
    (task-format "Start vacuuming"))

  (activity-finish
  ;; This is a percentage of the job, which is an hour
  (let ((amount (/ (- (finish-time self) (start-time self)) 3600)))
    (task-format "Finished vacuuming; did ~f percent~%"
                 (* 100.0 amount)))))

(define-event power-failure (event) ()
  (realize
    (task-format "Zot!  Power failure!")))

(defmethod interaction ((a vacuuming) (e power-failure))
  (task-format "Interaction detected")
  (setf (finish-time a) (timestamp e))
  (reschedule (this-simulation-state) a))
```

Figure 3.14. Part of the implementation of the “vacuuming” example.

of course, domain-dependent, but the MESS engine supports simulations that have interactions; it does so by defining *activities.*

Activities are processes that happen over time, with a start event and a stop event, such as traveling from place to place. Activities exist as simulation objects so that they can be notified of events that happen between the start and stop events. As we saw in the description of the MESS engine (section 3.2.1), this notification happens because activities are placed on a list when they start (and removed when they stop), and intervening events check this list to see if an interaction occurs.

3.5.1 Interaction

In more detail, what happens is that, for every activity in the engine's list, a MESS function, called `interaction`, is called with that activity and the current event. By default, that function does nothing, but users may *extend* the definition of `interaction` to do specific things with their domain-dependent activities and events.[8]

This is best explained with a simple example, drawn from the demonstration systems of MESS. In this example, one event stream is performing a “vacuuming” activity. Another event stream causes a power failure during the activity. Figure 3.14 presents part of the necessary code for the demo. The first form defines the `vacuuming`

[8]This specialization of behavior is a feature of object-oriented programming. Keene explains how this works in Common Lisp [30].

activity, which does nothing interesting except report when it starts and finishes. The `power-failure` event is similarly uninteresting. The work happens in the `defmethod` of `interaction`. By naming `vacuuming` and `power-failure` in its second argument, the `defmethod` extends the `interaction` function to execute the following code *only* when its arguments are those kinds of objects. Thus, the MESS `interaction` function is extended to domain-specific behavior. The code in this case is very simple—it just changes the finish time of the vacuuming to be the time that the power failure occurs, and reschedules the activity. (When the time of an event changes, the MESS engine needs to be notified so that the PEL is kept in the correct order. The `reschedule` function does this.)

The following is a transcript of running the "vacuum" demonstration.

```
3,601 VACUUMING Start vacuuming
4,000 POWER-COMPANY Interaction detected
4,000 POWER-COMPANY Zot!  Power failure!
4,000 VACUUMING Finished vacuuming; did 11.083333 percent
```

What about the interaction of two activities? For example, what if one train is moving from New York to Boston on the same track on which another train is moving from Boston to New York? These two activities should interact (spectacularly). In fact, this kind of interaction is a special case of the interaction described above, since activities are a special case of events. In the trains example, the start event of the second train will interact with the first train's activity, which is in the engine's list of current activities.

3.5.2 Interruptions

An agent's thought process ought to be something that interacts with intervening events. MESS supports this too, although the mechanism is slightly different from that of the previous section. The user marks a portion of the agent's code that is interruptible and marks places in the agent's code where an interruption is allowed to occur. When an interruption occurs, an "interruption function" is called to define the semantics of the interruption.

Consider two computer chess programs playing one another. They play the following game, a classic fool's mate.

1.	P-K4	P-K4
2.	B-B4	B-B4
3.	Q-B3	N-QB3
4.	QxP mate	

The algorithm of the players is designed to imitate a probabilistic anytime algorithm [3], in which, during each iteration, the agent has some probability of finding a move better than its current best. (Anytime algorithms, remember, are those that have an iterative improvement nature.) For this example, I gave each agent only one move, but on each iteration, the agent tosses a coin to see if "finds" the move. Thus, the improvement of quality is from having no move to having a move, and the improvement is probabilistic in that the *a priori* probability that the agent has a move monotonically increases with time.

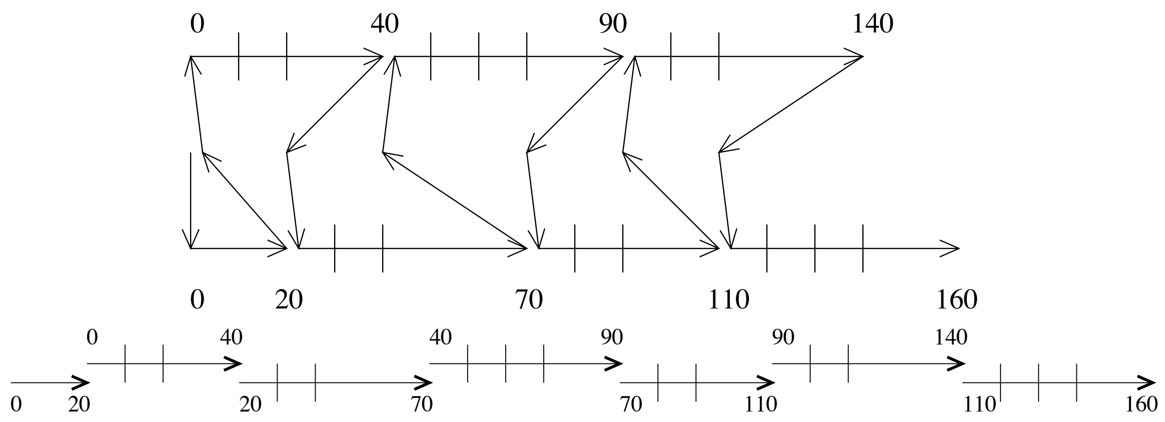


Figure 3.15. The chess players thinking in parallel. The chess players thinking in parallel. White is below and Black above. Control starts in the center with the Mess engine, which transfers control to White, which thinks for 20 time units and transfers control back to the engine. The engine then transfers control to Black, which thinks from time 0 to time 40 and transfers control back, and so on. The short lines perpendicular to the arrows indicate *interruption points*, which are described in the text.

For this example, let's fix the times that the agents find their moves as follows. This assumes that the cycle-time for finding a move is 10 seconds.

1.	20	40
2.	70	90
3.	110	140
4.	160	

The game is depicted in figure 3.15. Take Black's first move as an example. It takes Black 40 seconds to come up with its first move, the first 20 of which coincide with White's turn.

So far, there is no role for interruptions. However, suppose that the players lower their threshold for move quality when it's their own turn, so that they are more willing to accept a candidate move, thereby decreasing the time until they find a move. To change the threshold, an agent's thinking must be interrupted by the opponent's move. (Furthermore, a real chess-playing agent would interrupt its thinking to observe its opponent's move and take that into account.)

In the Mess demonstration, when the opponent makes its move, this interrupts a move searching loop and changes to one that uses a lower threshold for accepting a move. Part of the code is presented in figure 3.16. The first function defines how the chess agent works: it thinks while the opponent is thinking until that is interrupted, looks at the opponent's move (in this case only to see if it lost), and thinks again until it comes up with a move. (The agent playing White must start with the second loop; that code has been omitted.) The important part of this code is the `ithink` form, which marks a part of the code that is an interruptible thinking activity. The thinking activity starts when the form is entered and stops when the form completes.

```
(defun chess-thinking (who-am-i)
  (loop
  While opponent is thinking, establish interruptible loop
        (catch 'stop-thinking
             (ithink *[function to jump out of loop]*
                 (loop-forever (move-search nil))))
     Opponent has moved, so...
     If they just won, exit
     Note their move and finish thinking of my next move:
           (loop until (move-search t))
     Make my move))

(declare-primitive move-search :constant 10)

(defun move-search (my-turn?)
  (ip) **Allow interruptions here**
 This is the probabilistic anytime algorithm
 If I already have a good move
    return it
 else
    (call-book move-search)
    if it's my-turn
        flip low-threshold coin
    else
        flip high-threshold coin
    if coin flip says I found a move
        return the move
    else
        return false)
```

Figure 3.16. Code and pseudo-code for the chess-playing agents.

The first argument to the `ithink` form is an *interrupt handler*, that is, a function saying what to do if the thinking activity is interrupted. In this case, the interrupt handler will jump from the point of the interruption to the `stop-thinking` tag, from which the player goes on to check the opponent's move and so forth. Note that, in this case, the thinking activity comprises an infinite loop, so the thinking activity *must* be interrupted.

Where can the thinking activity be interrupted? Those points are marked with the `ip` function, which stands for "interruption point." The interruption points are depicted with short lines perpendicular to the arrows in figure 3.15. We see a call to `ip` at the top of the `move-search` function. That function is called to search for a move, whether it is the player's move or the opponent's. If a good move has already been found, it is returned immediately; otherwise, the function advances the clock and flips a coin to try to find a move.

A real chess program would not be a simple coin-tossing loop like this; it would probably be an elaborate search procedure. MESS can handle such procedures using just the forms we have described: surround the entire search procedure with the `ithink` form, mark appropriate interruption points with the `ip` form, and set up the interrupt handler to do whatever is necessary should an interruption occur.

How is interruptible thinking activity implemented? To understand this, we have to take a slightly different view of time, because thinking that happens simultaneously in the real world must happen sequentially in the simulation. The thinking of the chess agents happens as follows:

1. White thinks until it comes up with its first move, at time 20. The move event is scheduled but not realized.

2. Black's interruptible thinking activity starts (with its clock at time 0) and is allowed to think until the interruption occurs at time 20.

3. White's move occurs at time 20, interrupting Black's thought activity.

4. Black switches to its other thinking loop, and thinks until time 40, when it finds a move. Again, the move is scheduled but not realized.

5. White's interruptible thinking activity starts (with its clock at 20) and thinks until 40, when it's interrupted.

6. Black's move occurs at time 40, interrupting White.

7. and so on, as in the first step.

It seems strange to have agents thinking "in the past," as Black does in its first thinking phase. In all our previous example simulations, an event stream started at the current time and thought until it came up with an event at a future time. Thus, event streams were always thinking "in the future," and the engine worked by advancing time to the timestamp of whatever event-stream was earliest, and running that event stream so that it would move forward in time. This principle has not changed, even though it seems that it has. When White schedules its first move at time 20, simulation time does not advance to that time. Simulation time remains at

0, because that move event has not yet occurred. The engine then runs the Black agent, which sets up a thinking activity that is interrupted when the move event does occur.

3.6 Summary

The details of a discrete event simulator can be complicated, but the overall architecture is fairly straightforward. Event streams are sources of events, and the engine ensures that these events occur in the correct order. The timestamps of the events are determined by the kind of event stream: the list event streams have the timestamps specified *a priori*, the function event streams calculate the timestamp explicitly, and the thinking event streams determine the timestamp by the amount of computation since the last event. We saw that the inner loop of a simulation is the *advance* algorithm (figure 3.2), which selects the next ES and event from the pending event list (PEL), realizes and illustrates the event. It uses the "peek" and "pop" operations to get the event from the ES and to run the ES forward to its next timestamp. This simple protocol allows for user-defined extensions to MESS.

Two aspects of MESS set it apart from most simulation systems for real-time planning: support for thinking event streams and handling of interactions.

Thinking event streams are accomplished by an event stream that sets up a co-routine relationship between the ES and engine and by the TCL language. We saw how TCL replaces CPU time by defining functions that have the semantics of ordinary Common Lisp functions and also advance the clock in a well-defined way. This allows us to program in TCL just as in Common Lisp, while yielding a clean, platform-independent and agent-accessible measure of computation time. We also saw that the set of TCL primitives and the set of duration models are both easily extended by the user, to handle different kinds of real-time planning problems. We also saw how easy it is to have computations be "off the clock," something which is virtually impossible when CPU time is used.

Interactions are handled by the MESS engine's management of a list of current activities, checking whether events that occur during an activity interact with it. Interactions of events with thinking activities is similar, aided by the `ithink` form, which establishes a thinking activity, and the `ip` form, which allows for thinking to be interrupted at that point.

Chapter 4

Evaluation

This chapter presents experiments that (1) justify MESS and TCL, (2) prove that they provide the claimed replicability over experimental trials, (3) show some example experiments that can be run thanks to the properties of MESS and TCL, and (4) demonstrate the speed of the software.

4.1 Necessity

The main innovation of MESS is using TCL to replace the measurement of CPU time, eliminating the variability of CPU time. The following sections will show the variability of CPU time and its effects on simulators for real-time research.

4.1.1 Gabriel

In the mid 1980s, when Common Lisp implementations were just coming out, Richard P. Gabriel, who was a renowned Lisp programmer and member of the Common Lisp design committee, distributed a set of programs intended to benchmark aspects of Lisp implementations [18]. For example, the "Tak" benchmark computes a recursive mathematical function within the integers. (It is named for Ikuo Takeuchi who used the function as a benchmark before Gabriel.) "Tak" primarily tests speed of function calling and secondarily tests the speed of integer arithmetic. The function is defined as follows:

$$f(x,y,z) = \begin{cases} z & \text{if } x > y \\ f(f(x-1,y,z), f(y-1,z,x), f(z-1,x,y)) & \text{otherwise} \end{cases}$$

You can see that the function doesn't compute anything interesting, but takes a long time to do it. The benchmark program calls $f(18,12,6)$, which returns 7. Other benchmark programs include "Boyer," which is based on the Boyer-Moore theorem prover; "Deriv," which computes derivatives of symbolically represented functions; "FFT," which computes the Fast Fourier Transform of some data; and so forth. Many of these benchmarks appear in several variants. In all cases, the program is, like "Tak," completely deterministic and computed on constant arguments. There is no variability in the task posed by a benchmark program.

These benchmark programs were run 100 times each on a single computer, thereby controlling for differences in swap space or memory, even though those aspects of computer hardware should not affect CPU timings. The data are displayed in figure 4.1. If there were no variation in the CPU timings, there would be only one dot on each line, marking the time taken by the algorithm. Clearly, the timings are variable.

CPU seconds	mean	variance
ctak	0.05	0.0010
takr	0.06	0.0013
takl	0.07	0.0010
destru	0.08	0.0019
destru-mod	0.08	0.0031
stak	0.10	0.0022
tak-mod	0.15	0.0011
tak	0.16	0.0012
dderiv	0.21	0.0028
div2	0.22	0.0031
deriv	0.24	0.0070
boyer	0.73	0.1483
puzzle-mod	1.03	0.0020
puzzle	1.03	0.0033
browse	1.16	0.1238
fft	1.94	0.0109
fft-mod	1.96	0.0361
frpoly	2.31	0.0337
traverse	3.45	0.1506
triang	17.05	0.4961
triang-mod	17.12	0.4896
CPU seconds		

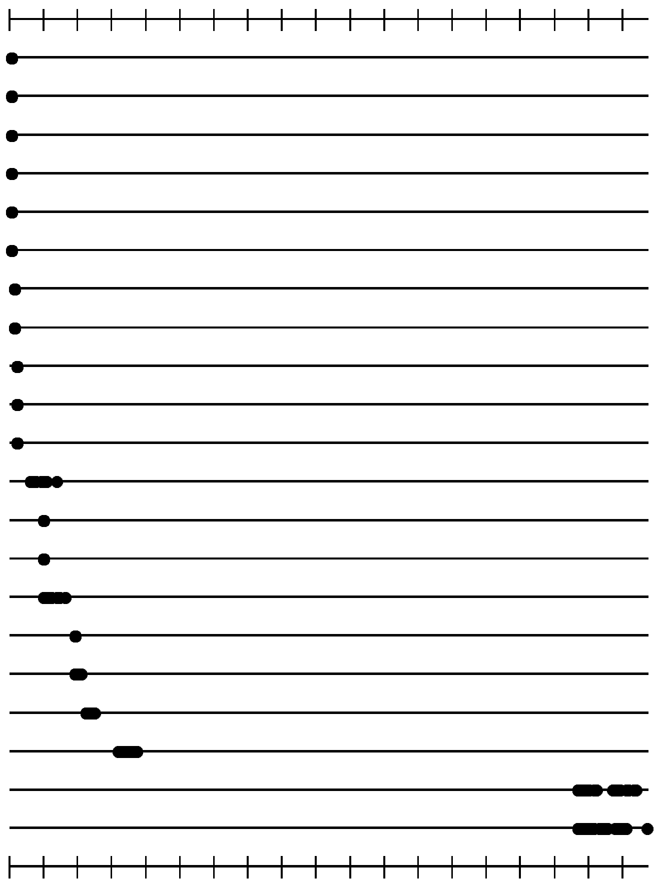
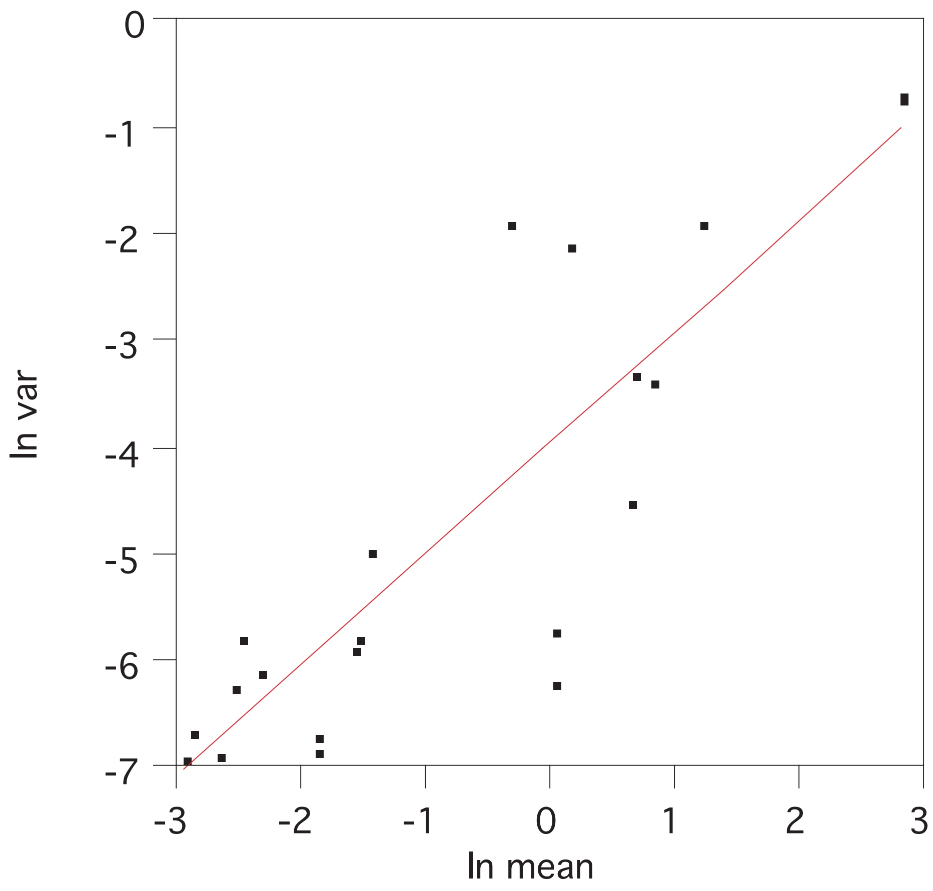


Figure 4.1. Raw data for the Gabriel benchmarks. They were run in Harlequin Lispworks release 3.2.1 on a Digital Equipment Corporation (DEC) Alpha. One hundred timings were done of each program, but many of the dots overlap, so fewer than 100 are visible. They have been arranged in order of increasing CPU time.



Summary of Fit

R^2	0.757793
Root Mean Square Error	1.079307
Mean of Response	-4.7214
Observations	21

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio	Prob
Model	1	69.248109	69.2481	59.4453	0.0000
Error	19	22.133185	1.1649		
Total	20	91.381294			

Parameter Estimates

Term	Estimate	Std. Error	t Ratio	Prob
Intercept	-3.978	0.2545	-15.63	0.0000
ln mean	1.0405701	0.13496	7.71	0.0000

Figure 4.2. Log variance as a function of log mean CPU time. A regression analysis indicates a good linear fit to the data.

Indeed, it seems that the variance may be proportional to the magnitude of the timing, since the "triang" and "triang-mod" benchmarks, which took by far the most time (16-18 CPU seconds), also have the widest spread.

Why should variance be proportional to the mean CPU time? We saw earlier that the CPU time required to execute a piece of code is computed by finding the difference between two readings from a "CPU-time accumulator" provided by the computer system. That is, CPU derives from, essentially, a running sum. It's well known that the variance of a sum of independent random variables is the sum of the variances:

$$\mathrm{VAR}\left(\sum \xi_i\right) = \sum \mathrm{VAR}\,(\xi_i)$$

Therefore, if each addition that the computer makes to the running sum is somewhat random, independently of other additions, the total should indeed be variable in proportion to its magnitude.

To confirm this hypothesis, we would like to use linear regression to find variance as function of the mean. Unfortunately, we can't do that directly with this data, because of the two outlier points, with high mean and variance. These "high leverage" points might give a good linear fit that is unwarranted by the data. Therefore, we take the log of both mean and variance. Figure 4.2 shows this transformation of the data, together with a linear regression analysis, which yields the following equation, where y is the variance of the benchmark and x is the mean CPU time of the benchmark:

$$\ln y = 1.04 \ln x - 3.978$$

If we round off the $\ln x$ coeffient to 1.0 (which is well within the standard error), and take the log of the constant term, we can simplify this equation a lot:

$$\begin{aligned} \ln y &= \ln x - \ln 53.41 \\ \ln y &= \ln(x/53.41) \\ y &= x/53.41 \end{aligned}$$

So, as we suspected, variance indeed seems to be a linear function of mean CPU time. Of course, as figure 4.2 shows, this is not the whole story, since the linear fit is far from perfect. For our purposes, it's sufficient to note that CPU time is variable, and long-running programs are more variable than short-running ones.

4.1.2 Registrar

In section 4.1.2, two "registrar" examples were introduced, one based on CPU time and the other on TCL. This section describes a short experiment contrasting those two simulations. The experiment will show that the small variations in CPU time that we saw in the previous section can result in qualitative changes in the behavior of a simulation, even a simple queuing simulation.

The registrar serves customers as they arrive, exponentially with mean inter-arrival time of 9 simulation time units. Each customer is served in constant time of 9 time units. This is done essentially by "inverting" the real-time knob. Recall that the real-time knob is a constant that maps some amount of computation (CPU or TCL)

```
CL-USER 100 > (registrar-cpu-experiment 64 4)
AADDADADADADADAAADAADAADADDADAADDDADAADADAAADAAAAD 276.1029, 1.9379
AADDADADADADADAAADAADAADADDADAADDDADAADADAAADAAAAD 277.1680, 1.9791
AADDADADADADADAAADAADAADADDDAAADDDADAADADAADAAAAAD 274.8755, 1.8644
AADDADADADADADAAADAADAADADDADAADDDADAADADAAADAAAAD 277.1649, 1.9780
NIL                                              ↑
                                                 Ordering change
CL-USER 101 >
```

Figure 4.3. Four runs of the CPU-time registrar for 64 events each. The numbers following each trial are the time of the last departure and the mean queue length for the trial. Note the next to last departure in the third trial, in which there is a change in event ordering.

into some amount of simulation time: $s = kc$, where s is simulation time, k is the real-time knob, and c is the amount of computation. If we want the simulation time of a computation to be a particular s, and we can measure the computation time to be c, we can just set the real-time knob to be $k = s/c$. For example, the real-time knob of the TCL registrar was set using the following code:

```
(defparameter *registrar-tcl-rtk*
              (load-time-value
                (/ (/ *arrival-rate*)
                   (duration (:stream nil :value :duration)
                             (tclu::arbitrary-thought-stuff)))))
```

The TCL **duration** function, described in section 3.4.8, is used here to find the duration of the **arbitrary-thought-stuff** function. The first argument to **duration** tells it not to print anything, but to return the duration as a value. The code for the **arbitrary-thought-stuff** function was given in figure 3.12; it just sorts a big array three times.

Code in the registrar example collects data on the queue length at each time, so that mean queue length can be reported at the end of each trial. In addition, the **illustrate** methods of the arrival and departure events print out an A or D, respectively, to show the order of events.

The output from four runs of the CPU registrar is displayed in figure 4.3. Every run had the same seed for the random number generator, so the arrivals happen at exactly the same time in each trial. Note, though, that in every trial, the time of the final departure and the mean queue length are different, due to variance of CPU-time. Even worse, in the third trial, a departure actually is early enough to precede an arrival. Thus, we see that CPU time variance can affect important characteristics of our simulations.

4.1.3 PHOENIX

The preceding sections showed that CPU times vary (which we expected), and that if these CPU times are used in a simple queuing simulation, we can see the

CPU variance infecting our queuing data, even to the extent of changing the order of events.

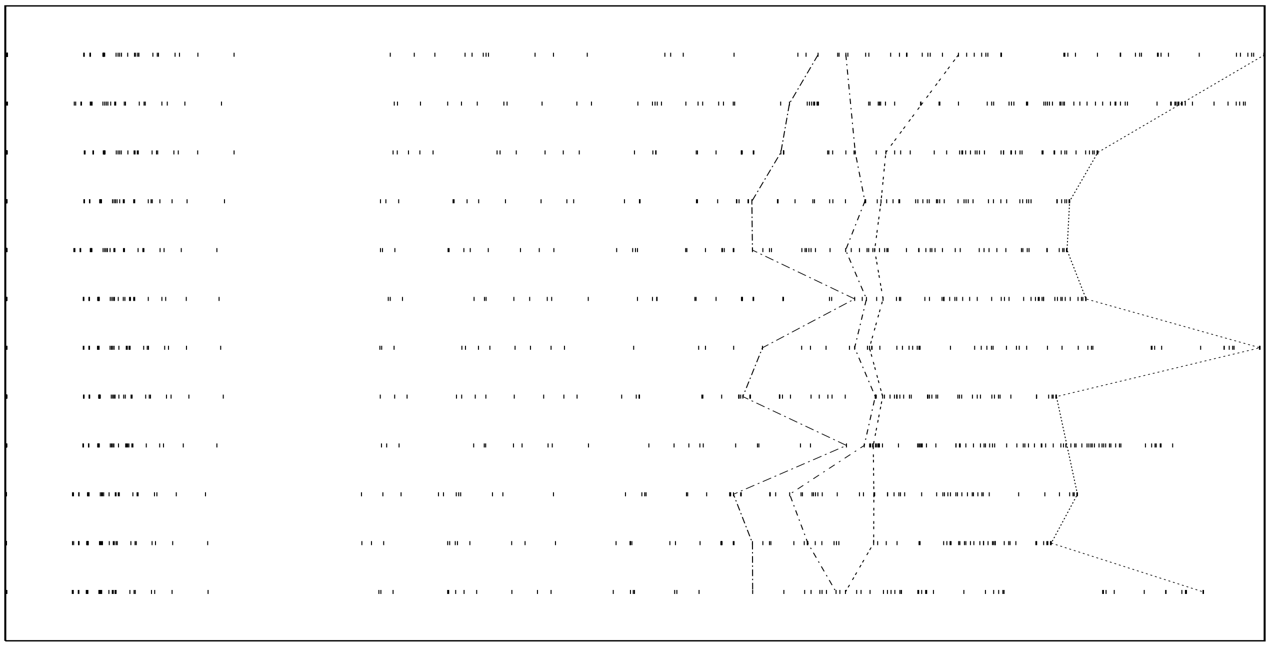
Does this variance continue to scale up? Does CPU variance really affect a large simulator like PHOENIX? Figure 4.4(a) shows that it does. Let me explain where the data comes from. As the PHOENIX simulator runs, the code prints out various messages, indicating what is going on. Each of those messages has a timestamp. I collected those timestamps for a dozen runs of PHOENIX with *identical* starting states. The fire was set at the same time, at the same place, in the same way, and the random number state was identical on every trial. All the trials were run on the same Lisp Machine on the same day, with the same software loaded. The runs were as identical as I could make them. The data were then plotted as (t, y), where t is the timestamp on the message and y is the trial number. Since there were so many messages that the lines became muddy, I uniformly deleted all the "monitoring," "plan info," "activating," "deactivating" and "calculating" messages. Also, I cut off all messages after time 41,000, since all of the fires that were successfully put out were put out by then.

In figure 4.4(a), the trials are clearly not identical. To make this easier to see, I have connected the points corresponding to the messages "low on gas," which bulldozers send to the Fireboss when they need to refuel, and "Marking fire as under control" which the Fireboss prints when it believes the task is complete. This is the jagged line at the right edge of the graph, showing how different the timestamps of those messages could be. Even worse, in trials 4 and 11, the fire was not successfully put out, and in trial 3, a bulldozer was killed by the fire. These trials came from the original PHOENIX, which did not use MESS or TCL in any way, but instead represented each agent with a process running on the Lisp Machine and measured the duration of agent thinking using the CPU time of that process. The Lisp Machine's operating system's scheduler was modified to ensure the processes didn't get too far out of step (usually less than five simulation minutes). It's important to remember, thought, that all the computation ought to be deterministic, given these identical starting conditions.

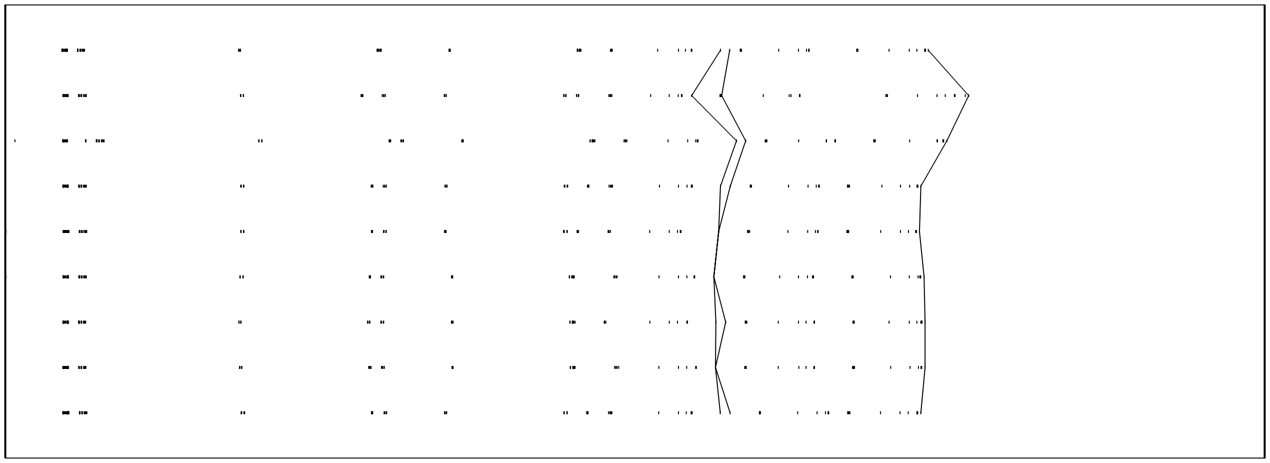
Figure 4.4(b) shows the new implementation of PHOENIX using MESS. The agents are modeled as thinking event streams, co-routining with the engine. Thinking time is measured using CPU-time, rather than TCL. Clearly, the trials are still not identical, although the variance is much reduced over the original PHOENIX trials. This variance reduction can be attributed to (1) changing from the Explorer to a Unix workstation, and (2) changing to co-routines from parallel processes running with a modified scheduler.

Finally, figure 4.4(c) shows the MESS-based implementation of PHOENIX using TCL instead of CPU-time. The behavior is identical in every trial. The variations in (b) can unquestionably be attributed to CPU-time variance because the only difference between the (b) and (c) trials is TCL versus CPU-time, just as in the "registrar" experiment above.

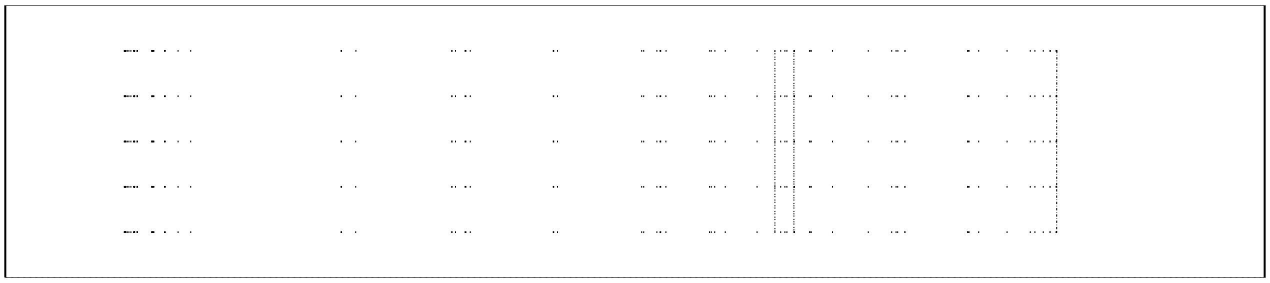
I think figure 4.4 is the single most important figure of this entire dissertation. Indeed, it is the alpha and omega of this thesis, because 4.4(a) is the reason that I embarked on the research, and 4.4(c) shows that I have accomplished what I intended: a platform-independent simulation substrate for real-time planning, with exact replicability of behavior.



(a)



(b)



(c)

Figure 4.4. Three sets of PHOENIX trials using identical starting conditions. The original, non-MESS CPU-time PHOENIX is at the top; MESS-based PHOENIX using CPU-time is in the middle; and MESS-based PHOENIX using TCL is at the bottom.

4.2 Success

One intended benefit of the MESS and TCL research is to resurrect the PHOENIX testbed, allowing it to run portably across machines. The preceding section showed that PHOENIX does indeed work, but does it have qualitatively the same behavior? One way to show this is to re-run an experiment that was run on the old PHOENIX system, to show that the data are similar.

That experiment is called the "Real Time Knob" (RTK) experiment [25]. It was a large, elaborate experiment that varied the wind speed and the value of RTK for the Fireboss agent, to show how the Fireboss responds to changes in problem difficulty and time pressure. Since the algorithms that the Fireboss uses to handle wind speed have not changed other than being coded in TCL, I did not replicate that aspect of the RTK experiment. I did vary the RTK over a similar set of values as in the original RTK experiment. Many dependent variables were measured; the one I report below is "shutdown time," which is the time at which the Fireboss realizes the fire is successfully contained, triggering the experiment-running software (CLIP [1]) to shut down the trial and go on to the next trial. If the fire is not successfully contained after some number of hours, CLIP shuts down the trial anyway.

Some data from the original RTK experiment is shown in figures 4.5(a) and (b). One feature of the experiment is that RTK was varied so that some values made the Fireboss think slower than usual while others made it think faster. Its "usual" thinking speed is the value of RTK that prevailed during the development and implementation of PHOENIX over the preceding years. The usual value is coded as 1 in 4.5(a) and the log transform makes that zero in 4.5(b), so negative numbers represent slower thinking, while positive numbers represent faster thinking.

One conclusion of the original RTK experiment, clearly visible in figure 4.5, is that the Fireboss's behavior is qualitatively different when it thinks slower than usual. While it almost always succeeds in fighting the fire, it performs much worse than in the "smart" conditions (thinking speed at least normal). Given additional thinking speed, its performance continues to improve, but not as markedly. The original experimenters said that "this shift in slope and value range for SD suggests a threshold effect in PHOENIX as the Fireboss's thinking speed is reduced below the normal setting of RTK." [25].

Data from the new RTK experiment is shown in figure 4.6(a). Again, the shutdown times have been converted to speeds, with cut off trials encoded as zero speed, plotted against the log of the RTK. Fifteen trials were run in each of six test conditions (values of RTK). We immediately notice that, when thinking speed is below normal, the Fireboss almost never succeeds. When thinking speed is at least normal, the Fireboss always succeeds. Figure 4.6(b) zooms in on those successful trials, showing a steady improvement in shutdown speed as thinking speed increases. Note clear separation into two groups of data. I partitioned the data by the plan that the Fireboss chose to use, and discovered that the lower speed data comes from the "MBIA" plan (Multiple Bulldozer Indirect Attack), while the higher speed data comes from two variants on MBIA, called "model" and "shell." Regression lines are plotted through the partitioned data.

There are obvious differences as well as similarities between the two versions of PHOENIX. In some respects, the old PHOENIX is more robust, since it usually succeeds

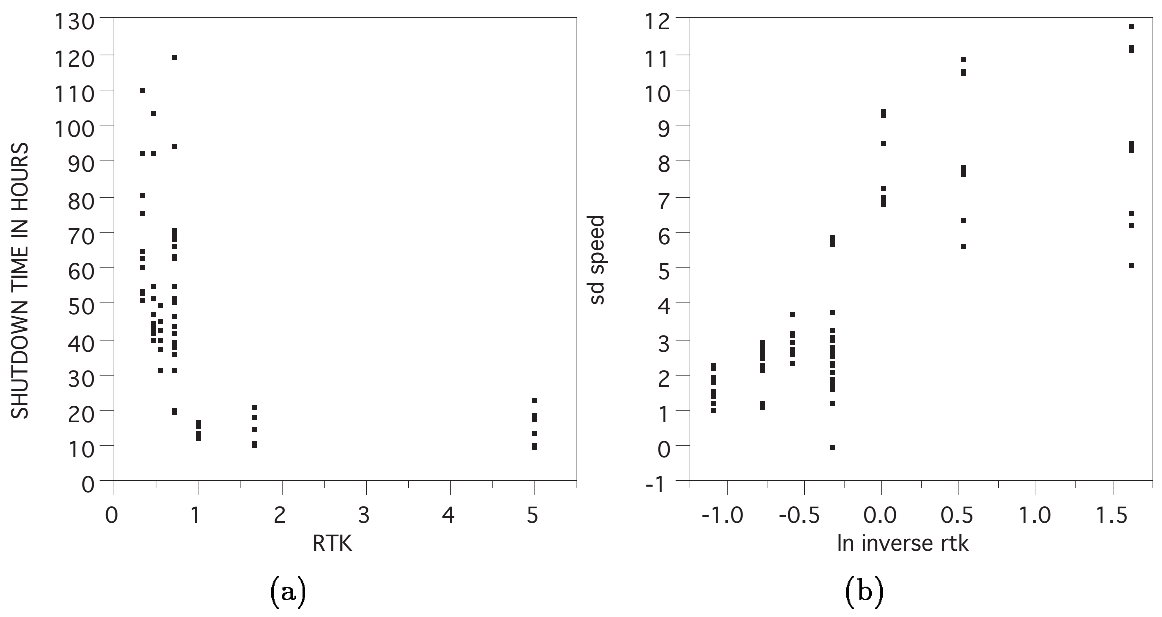


Figure 4.5. Data from the original RTK experiment. This is just the data with wind speed equal to 3kph. Both graphs show the same data, but in different ways. In (a), the raw shutdown time and RTK values are given, as presented in the original report [25]. In (b), the shutdown time has been recoded as speed, by dividing the shutdown time into the cutoff time of 120 hours. Trials that were cut off are recoded as zero speed. A log transform of the RTK spaces the x-coordinates better.

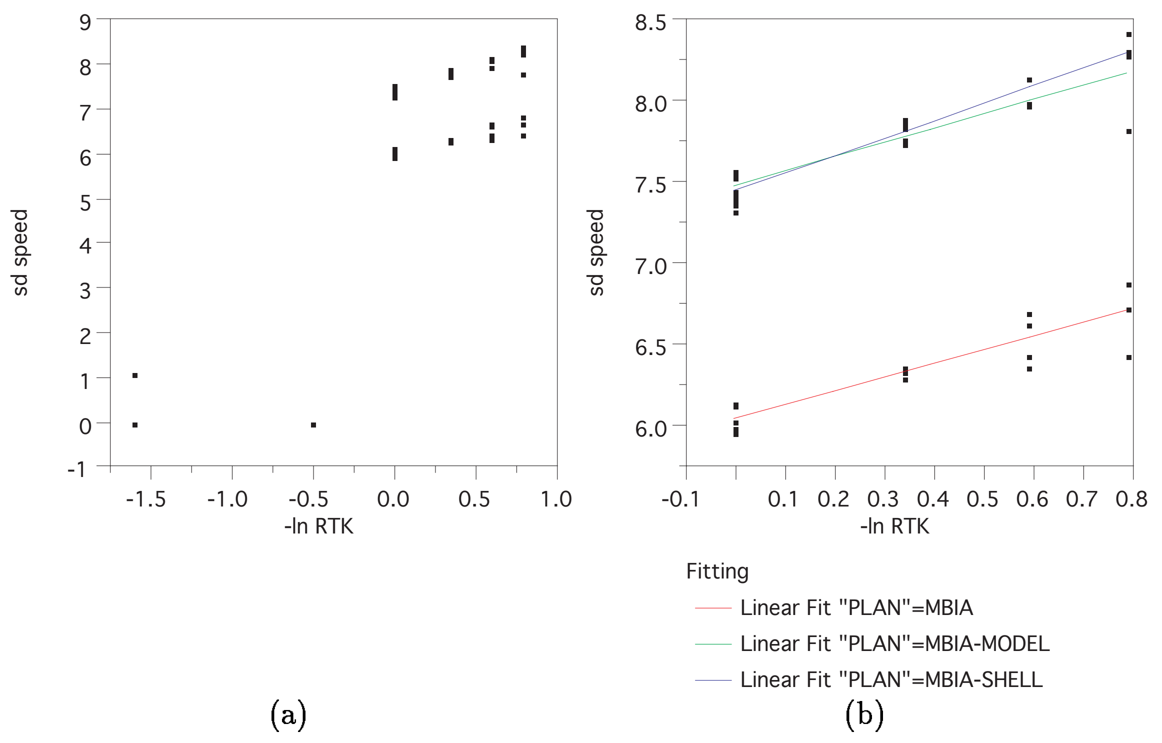


Figure 4.6. Data from the new RTK experiment. All the data are shown in (a). In (b), the graph focusses on the data in the upper right of (a).

in the slow-thinking conditions, while the new one almost always fails. I believe this may be due to the default setting of the RTK in the new PHOENIX. Since duration is measured in an entirely different way in the new PHOENIX, the default value of the RTK had to be set, without the benefit of years of subsequent tuning, in some way that mimics the old PHOENIX. The new RTK was set (to two simulation seconds per thousand duration units) based on comparing several trials of the old PHOENIX (using its default RTK) with several trials of the new PHOENIX. The new RTK may simply be too challenging. Additional tuning may be warranted.

Perhaps the most important and striking difference between the old and new data is the sharp drop in variance. For example, in figure 4.5(b), the data for the shutdown speed in the normal setting ($x = 0.0$) ranges from below 7 to above 9, which is the most tightly clustered of the conditions, while in figure 4.6(b), the corresponding data ranges from 6 to 7.5, and the range hardly increases in the other conditions. Thus, we have learned something about PHOENIX: a lot of the variance in the behavior of the old PHOENIX comes from variability of CPU time measurement, at least in cases where the environmental conditions don't change, as in these experiments.

4.3 Introspection

The old RTK experiment raised the question of why the Fireboss's performance dropped sharply when RTK was lowered below its normal value. One possibility, suggested by several people (including me), is that the Fireboss's estimates of "overhead" for various cognitive tasks include the latency of cognitive tasks. When RTK changes, those estimates do not change accordingly, so the Fireboss over- or under-estimates the time needed to think about certain things. Overestimating may hurt performance somewhat but will not usually cause failure, while underestimation may indeed cause failure.

Let's take an example. One of the standard ways to fight a fire in PHOENIX involves temporal projection [21, 22], which essentially means predicting the future state of the world, assuming that things continue as they are. In PHOENIX, the Fireboss projects the future size and shape of the fire, given its current size and shape, and assuming that wind and other such factors don't change. How long into the future? One method is to view firefighting as a race between the bulldozers digging fireline at rate c (cutting rate) and the fire increasing its perimeter at rate s (spread rate), with an initial lead of p_0 (initial perimeter). The race is won when the bulldozer's fireline exceeds the perimeter of the fire, since the fireline must enclose the fire. That occurs at time:

$$t = \frac{p_0}{c - s}$$

The preceding model is a simplification, because the bulldozers cannot start digging immediately. The time delay for fire projection, path planning, and bulldozer travel, can all be summed up in an increased initial lead for the fire:

$$t = \frac{p_0 + sd}{c - s}$$

Given t, the Fireboss can create a temporal projection of the fire for t time in the future.

The time it takes to create the temporal projection depends on t, since it takes longer to project the fire farther into the future. Therefore, d can depend on t as well as the reverse.

In PHOENIX, this complication is ignored, assuming that temporal projection doesn't take very long. Instead, some constant "slack time" is added to d. Generally, this works, especially since for simple, small fires, temporal projection for hundreds of simulation minutes usually takes less than a minute of simulation time, which is negligible. Unfortunately, there are cases of large, complicated fires where temporal projection can take up to an hour. This usually precedes catastrophic failure, since an hour is not negligible, and the inaccuracy results in a fire-fighting plan doomed to failure.

These "slack time" estimates may be why the Fireboss fails at slower settings of RTK, since these estimates do not depend on RTK, but were derived from experience with running PHOENIX at the normal setting of RTK.

One way to test this hypothesis is to take control of the time required by fire projection and allow the agent to introspect about the duration in its computations. To do this, we make fire projection a TCL primitive, with its own duration model. The code is given in figure 4.7. The first function defines a duration model for fire projection. This model depends on the perimeter of the fire, which is multiplied by 750 and rounded off to the nearest integer. Next, this duration model is used to declare `fire-projection` to be a TCL primitive, using that duration model and a proportionality constant of 1.

These two definitions allow the simulation clock to be advanced by the function `call-book` and also permits the agent to determine how long fire-projection will take by calling `primitive-duration`. (These functions were described in section 3.4.) In this case, the duration of fire projection is added to the `slack` variable, which is supplied to the `estimate-projection-time-based-on-model` function. That function returns `FST`, which stands for Fire Spread Time—the t from the equations earlier. The fire is then projected for that length of time, within the scope of `free` and just after calling `call-book`, so that the fire projection takes exactly as long as it is supposed to.

The results are shown in figure 4.8. They show a slight decrease in performance for the below-normal thinking speed, giving weight to the hypothesis, but converging to near identity when the RTK rises to its usual value. The conclusion is that allowing additional time for projection hurts performance at low thinking speeds and has no effect at higher thinking speeds.

However, it doesn't matter whether the hypothesis is right or wrong; what matters is that we were able to test it. The experiment has two points: (1) It shows how to define durations for high-level, domain-specific cognitive actions, such as fire projection, rather than low-level cognitive actions like the `cos` function with which we first introduced the notion (section 3.4.2). (2) It shows how an agent can obtain and use the duration of a cognitive action in its own thinking, thereby enabling meta-reasoning and scheduling.

4.4 Replicability

By use of MESS and TCL, we have complete control over the trajectory of a simulation—its sequence of states. If in two trials, the initial state is the same,

```
(mess:define-duration-model
 fire-projection-operations (fire-frame &rest ignore)
 (round (* 750 (fire-perimeter fire-frame))))

(mess:declare-primitive fire-projection fire-projection-operations 1)

(defmethod em-estimate-projection-based-on-model ((self fireboss))
  "Uses the linear model to estimate FST.  Projection problems cause
em-errors."
  ⋮
  (let ((slack (+ 60
                  (round (mess:primitive-duration
                          (mess:real-time-knob (this-event-stream))
                          'fire-projection
                          fire)
                  ;; Slack has to be in minutes
                  60))))
  (multiple-value-bind (fst dig-time minimal-dig-time)
      (estimate-projection-time-based-on-model
           fire num-bd rftt btm env-info slack)
      ⋮
    (let ((time (exact-time)))
       (task-format "Starting model-based projection at ~s" time)
       (call-book fire-projection fire)
       (free
          (project-fire-not-stupid fire ap firemap fst env-info
              spoke-res search-res fast? max-bd-seg nil))
    (task-format "Finishing model-based projection at ~s; took ~s"
       (exact-time) (- (exact-time) time)))
    ⋮
    :completed))
```

Figure 4.7. Making fire projection a TCL primitive The first two forms define the primitive, and the long function uses `primitive-duration` to look up the duration and add it to the slack time in a fire-fighting plan.

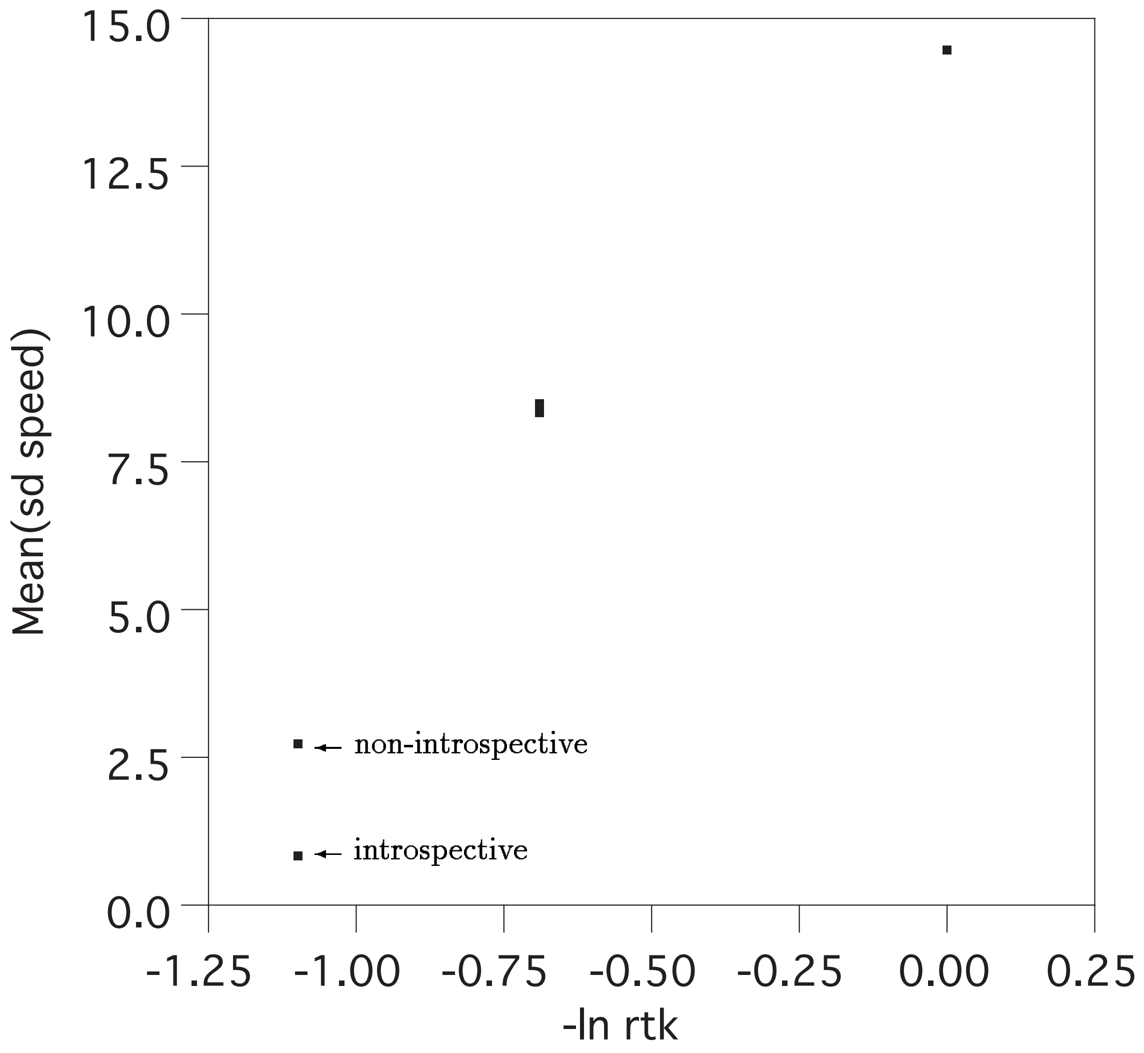


Figure 4.8. Data for the RTK Introspection experiment.

including the state of the random number generators, the simulation trajectories will be the same; the second trial is superfluous. However, the two trials may differ in exactly one respect, thereby allowing *matched-pairs* experiments. Because only one thing is different between matched-pairs trials, the experiment design is especially sensitive, allowing subtle effects to be discovered. This section explains how to do matched-pairs experiments using MESS and TCL, using a simple example of finding the difference between different fire-fighting plans.

The key idea for matched pairs experiments is that the random number seed becomes an independent variable: those trials that have the same value of the random number seed are the matched pairs. For example, in the following experiment, we varied three factors:

factor	levels	values
wind speed	3	3, 6, 9
fire plan	3	MBIA, MBIA-MODEL, MBIA-SHELL
random seed	5	1, 2, 3, 4, 5

How is this accomplished in MESS? As described before, each event stream has its own random number seed, so that the event streams are independent. However, for running experiments, MESS needs to vary the random number seed over the trials. This is done via a global variable, `*random-seed-modifier*`, which each event stream adds to its own base seed to yield the seed for this trial. Typically, `*random-seed-modifier*` is set to the trial number, so that all trials will be different. In a matched-pairs experiment, `*random-seed-modifier*` is set depending on the value of the independent variable, so that it will be the same in matched trials.

Figure 4.9 shows the shutdown speeds for all 45 trials. Again, failures are coded as zero speed. Table 4.1 gives the data comparing just the MODEL and SHELL plans, giving the differences in their shutdown speeds. (The numbers in the table have been rounded to two decimal places, but the computations were done on the original data.) A one-sample, two-tailed t-test of the 15 differences yields $t = -0.7501472$, which is not significant ($p < 0.46$). Excluding the pair of trials in which the MODEL plan fails, $t = 0.5503609$, which is still not significant ($p < 0.59$). If there is a difference between these plans, more trials will have to be run, because there is no better way to control for the variance. Such a result, however interesting, is not germane to this dissertation.

4.5 Performance

In addition to the ability to control durations, run matched-pairs experiments, and run simulations of PHOENIX, there is a practical side to MESS and TCL: how efficient are they? While the bulk of time spent in running a simulation is spent in executing domain-specific code, we still want to minimize the overhead spent in the engine and in TCL. This section describes the performance of MESS and TCL.

Table 4.1. Plan performance data comparing the MODEL and SHELL plans.

wind	random	plan	sd speed	difference
3	1	MBIA-MODEL	9.04	
3	1	MBIA-SHELL	8.04	
				0.99
3	2	MBIA-MODEL	9.05	
3	2	MBIA-SHELL	7.82	
				1.23
3	3	MBIA-MODEL	9.04	
3	3	MBIA-SHELL	8.04	
				0.99
3	4	MBIA-MODEL	9.04	
3	4	MBIA-SHELL	8.03	
				1.00
3	5	MBIA-MODEL	8.10	
3	5	MBIA-SHELL	7.82	
				0.27
6	1	MBIA-MODEL	7.68	
6	1	MBIA-SHELL	7.60	
				0.078
6	2	MBIA-MODEL	7.83	
6	2	MBIA-SHELL	7.80	
				0.03
6	3	MBIA-MODEL	7.60	
6	3	MBIA-SHELL	7.65	
				-0.05
6	4	MBIA-MODEL	7.67	
6	4	MBIA-SHELL	7.66	
				0.01
6	5	MBIA-MODEL	0	
6	5	MBIA-SHELL	7.59	
				-7.59
9	1	MBIA-MODEL	6.32	
9	1	MBIA-SHELL	6.77	
				-0.46
9	2	MBIA-MODEL	6.12	
9	2	MBIA-SHELL	6.54	
				-0.43
9	3	MBIA-MODEL	6.21	
9	3	MBIA-SHELL	6.51	
				-0.30
9	4	MBIA-MODEL	5.71	
9	4	MBIA-SHELL	6.70	
				-0.99
9	5	MBIA-MODEL	5.66	
9	5	MBIA-SHELL	6.56	
				-0.91

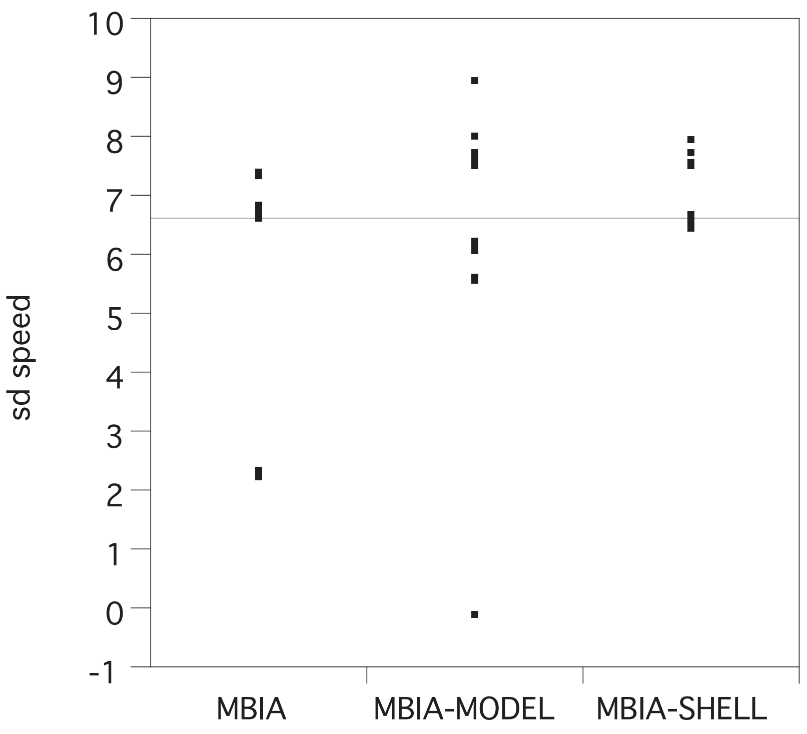


Figure 4.9. Data from the matched-pairs experiment The data is partitioned by fire-fighting plan, and the light gray line is the grand mean of the three groups.

4.5.1 MESS

As described in section 3.2.1, simulations consist primarily of calls to the **advance** function, so its efficiency primarily determines the efficiency of simulation. The best measure of this efficiency is events per second, since each event represents one complete execution of the **advance** function, including getting the event stream from the pending event list (PEL) and reinserting it afterwards.

Much of what the **advance** function does is fixed-cost overhead, such as managing the clock, peeking and popping event streams, and so forth. Managing the PEL, however, depends a lot on the user's simulation code, particularly (1) the number of event streams, which determines the size of the PEL, and (2) the pattern of timestamps produced by the event streams, which determines where they will be inserted into the PEL.

Let's look more closely at the last point. Assume the PEL is represented by a sorted list. The next event is simply the one at the beginning of the list, and so it is a small, $O(1)$, cost to delete that event from the PEL. After advancing the simulation, a new event must typically be inserted into its correct place in the PEL. If it has a timestamp in the near future, it will go near the front of the list, an operation which is convenient and quick to compute. If the new event goes into the far future, it will go to the end of the list, an operation that takes a good deal of time: $O(n)$ steps in the worst case, where n is the size of the PEL. Furthermore, ties must be handled; in MESS, ties are broken by priority of the event stream, and then arbitrarily, if there is also a tie in priority. Looking at priority takes time, and so ties can slow down management of the PEL.

In MESS the PEL is not represented as a sorted list, but as a "heap," which can be thought of as a partially sorted tree. Heaps are often ideal for PEL representation because an element can be inserted at the very end in $O(\log n)$ time, rather than the $O(n)$ for a sorted list. Thus, their worst case time is much better. Unfortunately, this comes at the cost of slowing down the operation of removing the first element from the PEL, from $O(1)$ to $O(\log n)$. Nevertheless, a heap optimizes the worst-case performance, which is the correct performance goal for MESS, since it is a domain-independent substrate. Some simulators might be better off with other data structures for their particular pattern of timestamps, but a heap is a good compromise.

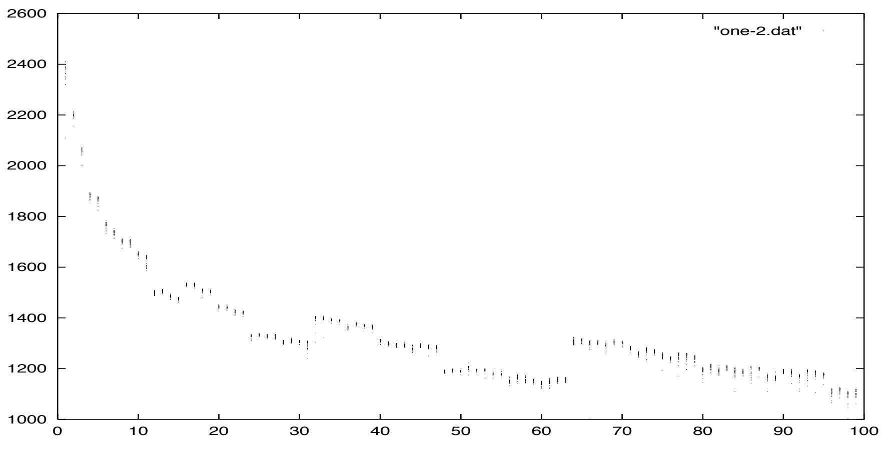
The actual efficiency of a heap still depends on the relative positions of the elements inserted and deleted, because it's still cheaper to insert near the front rather than near the back. To measure the performance of MESS's heap, I ran three simulations with different patterns of timestamps, all variations on the "clocks" example. The clocks example is used because it spends a minimal amount of time in the user's code (just computing the next clock event), so the run time of a simulation is determined entirely by the efficiency of MESS. The following variations were run:

- All the clocks have the same period and the same starting time, so ties are ubiquitous. The engine processes all the events at time 1, then all those at time 2, and so forth. That is, the PEL always has two classes of events in it: those at timestamp t and those at $t + 1$. Insertions will tend to be in the middle of the PEL, because the new event will be at time $t + 1$ and will be inserted between t events and the $t+1$ events. (MESS breaks ties in the way that is computationally most efficient.)

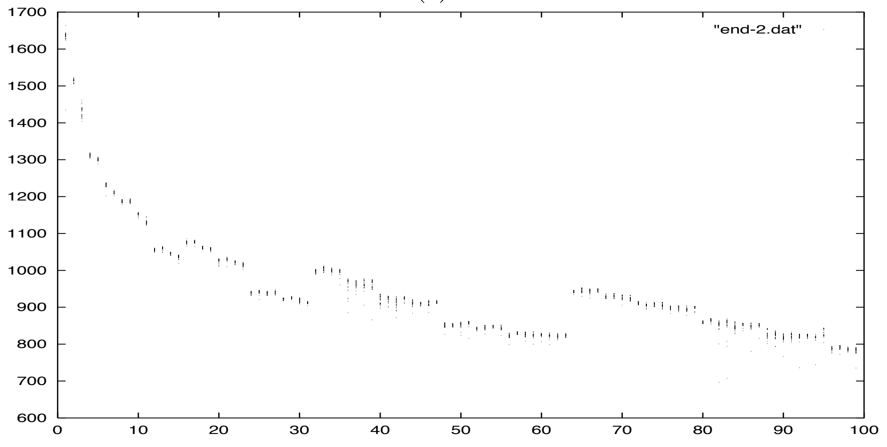
- All the clocks have a period equal to the number of event streams, and they all have different starting times. For example, if there are five event streams, A–E, A will have timestamps 1, 6, 11, ..., while B will have 2, 7, 12, ..., and so forth. In this case, there are no ties, and an event stream always is inserted at the very end of the PEL. This scenario represents something of a worst-case pattern of timestamps.

- The clocks all have different periods, which are pairwise relatively prime, so ties among timestamps are relatively infrequent. This also means that short-period clocks will tend to be inserted into the middle of the PEL, while long-period clocks will be inserted at the end. This scenario represents a kind of average case pattern of timestamps.

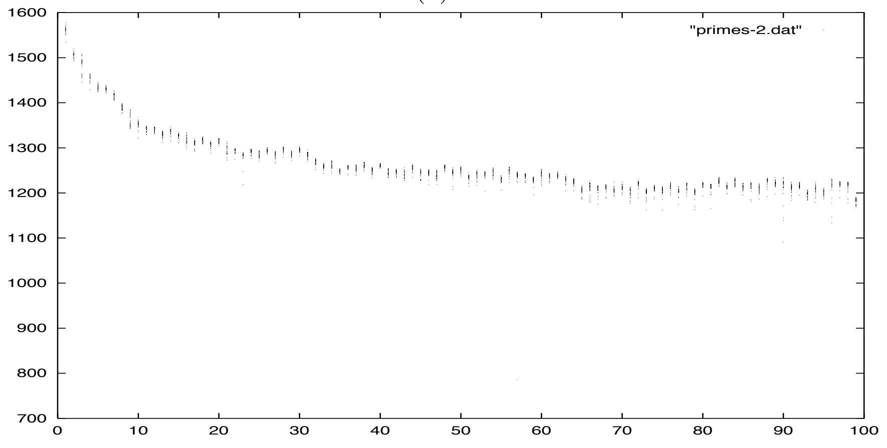
The speed of the simulation, in events per second, was calculated by running the simulation for n events, measuring the run time in seconds, t, and recording the ratio n/t. In these experiments, $n = 1000$. The number of clocks (event streams) was varied from 1 to 100, to see how MESS performs in a variety of relevant conditions (PHOENIX, for example, has just ten event streams) as well as seeing how its performance would scale up for simulations with many event streams. Thirty trials were run in each condition.



(a)



(b)



(c)

Figure 4.10. The speed of the MESS engine, plotting all data. The speed is reported in events per second as a function of the number of event streams. The conditions are: (a) equal periods and starting times, so there are many ties and insertions are typically in the middle of the PEL; (b) equal periods and different starting times, so there are no ties and insertions are always at the end of the PEL; and (c) prime periods, so there are infrequent ties and event insertions are distributed over the front, middle, and end of the PEL.

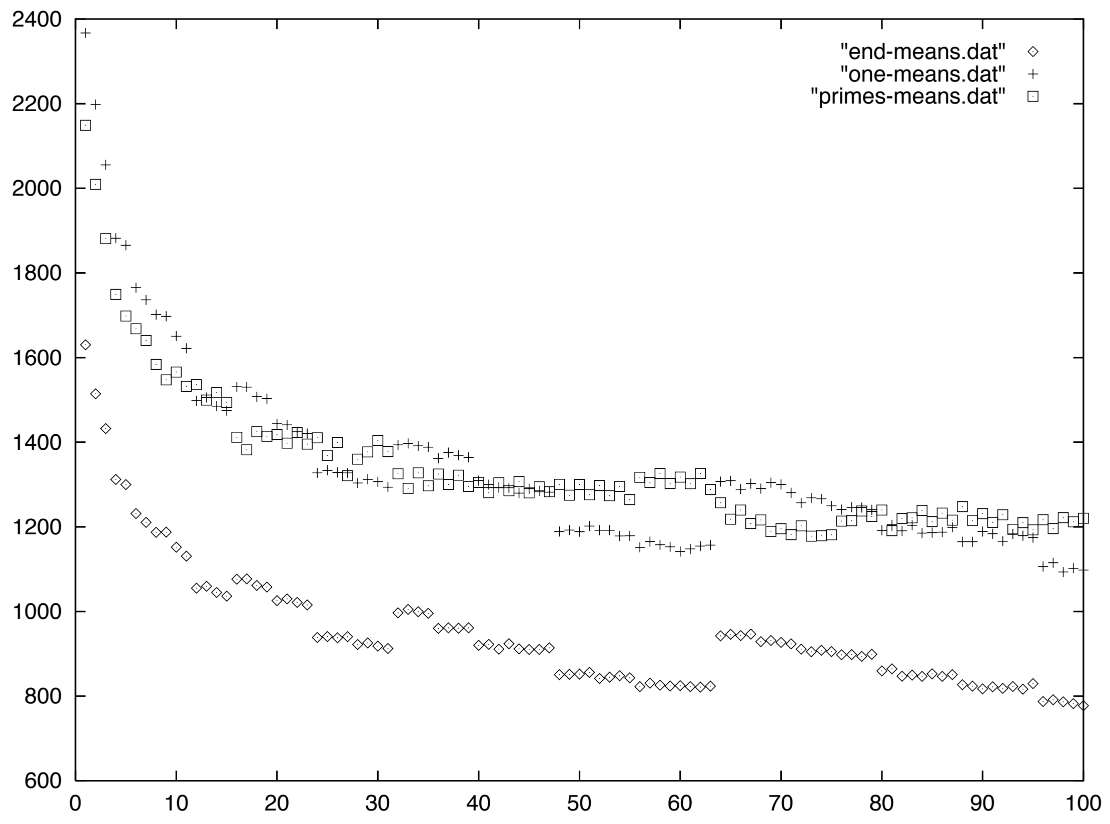


Figure 4.11. The speed of the engine, plotting mean performance. This graph just collapses the subfigures of figure 4.10.

The data for the three experiments are presented in figure 4.10. The graphs are combined in figure 4.11, which plots the mean for each condition. This graph clearly shows the lower performance in the condition that we expected to be the worst case. The graphs also show the diminishing decline behavior that we would expect of a simulation whose performance is dominated by the $O(\log n)$ PEL representation. We can expect that MESS will continue to scale well to larger simulations.

4.5.2 TCL

We have seen that TCL is just like Common Lisp, except that each function advances the clock in addition to calling its Common Lisp twin. Because each TCL function has a bit more work to do than its twin, we expect a TCL program to run slower than the equivalent Common Lisp program. To show how much slower, the Gabriel benchmarks were run using both Common Lisp and TCL. Table 4.2 gives the timing of the TCL functions *relative* to the Common Lisp. The first column gives the relative timing with code-walking turned off, while the second column has code-walking turned on. The table shows that code walking always helps a little bit, and sometimes helps a lot. In general, it looks like TCL runs over twice as slowly as Common Lisp, without code walking, and half again slower with code walking.

Why is TCL's performance so bad? After all, TCL only adds an increment to a variable, which ought not to be computationally expensive. There are several reasons:

Table 4.2. Relative performance of TCL on the Gabriel benchmarks. Programs were run on a DEC Alpha running Harlequin Lispworks.

Benchmark	without CW	with CW
boyer	2.6	2.2
browse	3.8	2.5
ctak	1.9	1.3
dderiv	1.2	1.0
deriv	1.2	1.0
destru-mod	2.5	1.2
destru	2.4	1.2
div2	2.1	1.2
fft-mod	1.4	1.0
fft	1.2	1.0
frpoly	1.6	1.3
puzzle-mod	1.5	1.1
puzzle	1.6	1.1
stak	1.6	1.3
tak-mod *	3.6	2.3
tak *	3.2	2.4
takl *	3.8	2.6
takr *	2.4	1.5
traverse	2.2	1.8
triang-mod	1.3	1.0
triang	2.1	1.2
Mean	2.15	1.49
Mean w/o *	1.9	1.3

- The Tak benchmark is somewhat over-represented in the suite, since there are four variants. Tak is so extremely simple, consisting only of additions or subtractions on integers, that incrementing a clock represents a significant increase in its computation. Eliminating Tak and its variants improves the relative performance to 1.9 (without CW) and 1.3 (with CW). Tak is not particularly representative of programs in artificial intelligence, and so may be a poor benchmark.

- Generally, many of the benchmark programs are not like AI programs, emphasizing low-level numeric computations. Two counterexamples are Puzzle and Triang: the first is a search program to fit 3-D puzzle pieces together, and the second solves a puzzle like Chinese Checkers. TCL performs fairly well on these benchmarks, especially with code-walking. Of course, there is still room for improvement, and that work is ongoing.

- Compilers are designed to recognize and optimize common code patterns, and TCL spoils those patterns by inserting increments. This is probably one reason that code walking is so effective, because pulling the increments out of the code restores the patterns the compiler is looking for. Still, there are undoubtedly many patterns being spoiled.

Finally, these timings are essentially the worst case for TCL. TCL is designed to mimic Common Lisp, but it is also designed to let users define their own durations for functions, as we did in the "introspection" experiment above, where we defined fire projection as a TCL primitive. This is a very high-level primitive, compared to the low-level functions that the fire projection function calls. Once fire projection is a primitive, it can call the faster Common Lisp functions rather than their TCL equivalents. Therefore, using higher-level primitives can speed up TCL.

4.6 Summary

The main objective with MESS and TCL was to build a simulation substrate with integrated support for real-time planning, using the TCL language to replace CPU time. This chapter has shown that that goal has been achieved, because the PHOENIX simulator runs using MESS and TCL, but the haphazard behavior caused by CPU variance has been eliminated. Eliminating variance allows matched-pairs experiments just by making the random number seed one of the independent variables. The control over duration allows the agent to reason about its own thinking. The performance of both MESS and TCL is good, but TCL could use some improvement. The bottom line is that it is currently useful for implementing practical simulators.

CHAPTER 5

CONTRIBUTIONS

MESS and TCL dovetail to produce a substrate for conducting empirical research in real-time planning. They are the primary contribution of this dissertation. MESS, by itself, doesn't know anything about the duration of thought, but it can be used without TCL for ordinary discrete event simulation. Indeed, it is currently being used for such simulations. TCL, by itself, doesn't know anything about simulation, but it can be used without MESS to run code to yield precise, replicatable, and introspectable durations. For example, researchers in anytime algorithms or design-to-time scheduling can run agents that can easily know about the duration of their cognitive primitives. The combination of MESS and TCL integrates these durations with other events in a simulation, so that the agent can interact with a simulation. The simulation provides the time-pressure on the agent's thinking (planning), and so the simulator is useful for real-time planning research.

MESS and TCL make the following contributions:

- A simulation substrate for discrete event simulation in AI. MESS makes no commitments to environment representation. Researchers will always want to use their own domains, since those domains emphasize issues that are of personal interest. Having a portable substrate for implementing those domains will foster better collaboration between sites and allow results to be easily replicated, extended and generalized. That MESS provides an effective substrate for discrete event simulation is shown by its use in supporting PHOENIX and other simulators in the Experimental Knowledge Systems Laboratory (EKSL) at the University of Massachusetts.

- A platform-independent way of advancing time in a computational process, so that the process can be integrated with other event streams in a discrete event simulator. Several kinds of duration models can be stored in the database. This platform-independent way of timing algorithms is also useful to anytime algorithm and design-to-time researchers, since their models of thinking time do not depend on which computer, compiler, and so forth, the code runs on. The platform-independence of TCL is shown primarily by its design: the characteristics of the underlying computer system are simply irrelevant to its models of duration, which instead are closely tied to the functions that the user employs. Replicability and introspection are straightforward consequences of that design.

- Support for detecting interactions between continuous processes, such as vehicle collisions, and particularly interactions between an agent's thinking and concurrent events that may interrupt that thinking. The examples of interrupting a vacuuming activity and a chess-player's thinking activity showed these aspects of the MESS design.

- An expressive, extensible agent language for describing the thinking processes of the agent. TCL is a full-featured programming language, since it duplicates all of Common Lisp. TCL is also modifiable, because the user can change the duration model for any of the TCL primitives, including defining new kinds of duration models. In addition, the user can extend TCL by adding new primitives to the language. These can be general utility functions or domain specific functions like the fire projection primitive in PHOENIX.
- Support for research on decision-making in complex environments, because the simulation can replicate the state of a decision, allowing the decision to be made several ways, so that outcomes can be directly compared. Replicating the state of a decision comes from the basic replicability of MESS and TCL, but by making the random number seed an independent variable in an experiment, a matched-pairs design is possible, allowing close analysis of any branch point in a simulation trajectory. We saw this in the experiment analyzing the effects of the three fire-fighting plans. Ordinarily, the agent chooses randomly among the three plans; by forcing it to choose the plan specified by the independent variable, we can analyze the effects of that decision.

MESS has allowed the reimplementation of PHOENIX, so that this complex, realistic real-time environment is again available to the AI community. The previous unavailability of PHOENIX was unfortunate, since its simulation of forest fires in Yellowstone National Park is one of the most interesting environments supplied by current AI testbeds. The source of time pressure is a realistic world process. When a PHOENIX plan fails, it fails for reasons resulting from natural environmental change (such as a change in the wind). Furthermore, plans may fail gradually, as the effects of, for instance, changed wind speed accumulate. The gradual failure provides greater opportunities for plan monitoring (to see whether it's going to fail), plan steering (to try to prevent its failure), and failure recovery (to figure out why it failed). The continuous space in the two-dimensional world enables interesting geometric reasoning and path planning. I believe there is a great deal to be gained by further experiments using the PHOENIX simulator.

Finally, the most important use of MESS and TCL will be in building new simulators, built from the beginning with the idea of modular, reusable event streams for modeling world processes and agents that can reason about their thinking time in order to act effectively under time pressure.

I consider MESS and TCL to be essentially finished work. They are tools for the particular purpose of supporting research in real-time planning, and I have argued that they fulfill that purpose. They certainly work sufficiently well to support the PHOENIX testbed.

The next step is to distribute these tools to the real-time planning community, so that they can be used. Once they are in active use, opportunities for enhancement will undoubtedly spring up. Some possible enhancements are:

process event streams In the current implementation of MESS, the only event streams that use co-routines are the TCL and CPU-time event streams, which are used for implementing agents. There's no reason that the co-routine control relationship with the engine must be bundled with thinking agents. Therefore, it would be straightforward to add an event stream to support co-routines for event streams that represent arbitrary environmental processes.

duration of non-primitives Currently, TCL's database can report the duration of any primitive operation, but the duration of higher-level functions is not known: the only way to determine the execution time of a function is to run it. TCL could be modified so that the duration of a function is recorded in the database each time it runs. These durations can be used in subsequent runs as estimates of duration, increasing the possibilities for agent introspection. Essentially, this is the automatic collection of "performance profiles" [59].

worst-case execution time The TCL code-walker could easily be modified so that it computes the worst-case execution time (WCET) for bounded code, as in SPRING-C [43]. The WCET estimates can be stored in the TCL database as additional information available to the agent for meta-reasoning. Syntactic constructs for bounded loops and recursion would have to be added to the TCL language in order to write bounded code.

average-case execution time Given either empirical or *a priori* estimates of (1) the probability that a branch is taken, (2) the number of loop iterations, and (3) the depth of recursion, the TCL code walker could be modified to compute average-case execution time as well. This is a straightforward generalization of the preceding algorithm, taking advantage of additional information about the run-time characteristics of the code.

efficient, agent-specific duration models Currently, TCL allows different agents to use different duration databases if desired, but the implementation is not as efficient as if all agents use the same database. If they all use the same database, the duration models can be looked up at compile time. If a researcher wants several agents that are substantially similar but vary slightly, there is no alternative but to use the less efficient run-time lookup of duration models. However, a reorganization of the TCL implementation could improve the efficiency of this situation by allowing compile-time lookup of duration models for primitives that are the same over all agents, using run-time lookup only for primitives that differ between agents.

parallel simulation Multiprocessor workstations are increasingly common, presenting the possibility of executing agents in parallel, with the MESS engine controlling concurrent processes instead of co-routines. This could substantially accelerate simulations.

efficiency There may be additional speed that can be squeezed out of the TCL language, so that the overhead of its use can be further reduced, thereby speeding up simulations.

These and other enhancements to MESS and TCL will be made as the need arises. For now, the research directions of this work are in real-time planning, deliberation scheduling and control, meta-reasoning, plan monitoring, failure recovery, and other issues of real-time artificial intelligence.

Bibliography

[1] Anderson, S. D., Carlson, A., Westbrook, D. L., Hart, D. M., and Cohen, P. R. Tools for empirically analyzing AI programs. In *Proceedings of the Fifth International Workshop on Artificial Intelligence and Statistics* (1995), The Society for Artificial Intelligence and Statistics and The International Association for Statistical Computing.

[2] Baase, S. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, 1988.

[3] Boddy, M., and Dean, T. Solving time-dependent planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (1989), pp. 979–984. Detroit, Michigan.

[4] Bratley, P., Fox, B. L., and Schrage, L. E. *A Guide to Simulation.* Springer-Verlag, 1983.

[5] Bratman, M., Israel, J., and Pollack, M. Plans and resource-bounded practical reasoning. *Computational Intelligence 4* (1988), 349–355.

[6] Carver, N. F. *Sophisticated Control for Interpretation: Planning to Resolve Sources of Uncertainty.* PhD thesis, University of Massachusetts at Amherst, 1990.

[7] Cohen, P. R., Greenberg, M. L., Hart, D. M., and Howe, A. E. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine 10*, 3 (Fall 1989), 32–48.

[8] Cooper, R., Fox, J., Farringdon, J., and Shallice, T. Towards a systematic methodology for cognitive modeling. Technical Report UCL-PSY-ADREM-TR5, University College London, November 1992.

[9] Dean, T., and Boddy, M. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence* (1988), American Association for Artificial Intelligence, Morgan Kaufmann, pp. 49–54. St. Paul, Minnesota.

[10] Decker, K. S. TÆMS: A framework for environment centered analysis & design of coordination mechanisms. In *Foundations of Distributed Artificial Intelligence*, G. O'Hare and N. Jennings, Eds. Wiley Inter-Science, 1994, ch. 17. To appear.

[11] Decker, K. S., and Lesser, V. R. Analyzing the need for meta-level communication. Technical Report 93-22, University of Massachusetts at Amherst, Computer Science Department, 1993. This technical report combines and expands the authors' IJCAI-93 and AAAI-93 papers.

[12] Decker, K. S., and Lesser, V. R. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance and Management* (1993).

[13] Dickens, P., Heidelberger, P., and Nicol, D. Timing simulation of paragon codes using a workstation cluster. In *Proceedings of the 1994 Winter Simulation Conference* (1994), J. D. Tew, M. S. Manivanna, D. A. Sadowski, and A. E. Seila, Eds., Institute of Electrical and Electronic Engineers, pp. 1347–1353.

[14] Durfee, E. H. *A Unified Approach to Dynamic Coordination: Planning Actions and Interactions in a Distributed Problem Solving Network.* PhD thesis, University of Massachusetts at Amherst, September 1987. Also available as Computer Science Department Technical Report 87-84.

[15] Durfee, E. H., and Montgomery, T. A. MICE: A flexible testbed for intelligent coordination experiments. In *Intelligent Real-Time Problem Solving: Workshop Report* (Palo Alto, CA, 1990), L. Erman, Ed., Cimflex Teknowledge Corp.

[16] Engelson, S. P. *Passive Map Learning and Visual Place Recognition.* PhD thesis, Yale University, New Haven, CT, May 1994 1994. Also available as technical report YALEU/CSD/RR #1032.

[17] Engelson, S. P., and Bertani, N. Ars Magna: The abstract robot simulator manual, version 1.0. Technical Report 928, Yale University, New Haven, CT, October 1992.

[18] Gabriel, R. P. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.

[19] Garvey, A., and Lesser, V. Design-to-time real-time scheduling. *IEEE Transactions on Systems, Man and Cybernetics 23*, 6 (1994). Special Issue on Planning, Scheduling and Control.

[20] Greenberg, M. L., and Westbrook, D. L. The Phoenix testbed. Tech. Rep. COINS TR 90–19, Computer and Information Science, University of Massachusetts at Amherst, 1990.

[21] Hanks, S. Practical temporal projection. In *Proceedings of the Eighth National Conference on Artificial Intelligence* (1990), American Association for Artificial Intelligence, MIT Press, pp. 158–163.

[22] Hanks, S. *Projecting Plans for Uncertain Worlds.* PhD thesis, Yale University, Department of Computer Science, 1990.

[23] Hanks, S., Nguyen, D., and Thomas, C. The new Truckworld manual. Tech. rep., Department of Computer Science and Engineering, University of Washington, 1992. Forthcoming. Contact `truckworld-request@cs.washington.edu`.

[24] Hanks, S., Pollack, M. E., and Cohen, P. R. Benchmarks, testbeds, controlled experimentation, and the design of agent architectures. *AI Magazine 13*, 4 (1993), 17–42.

[25] Hart, D. M., and Cohen, P. R. Predicting and explaining success and task duration in the Phoenix planner. In *Proceedings of the First International Conference on AI Planning Systems* (1992), Morgan Kaufmann, pp. 106–115. Technical Report 93-19.

[26] Herbordt, M. C. *The Evaluation of Massively Parallel Array Architectures.* PhD thesis, University of Massachusetts at Amherst, 1994.

[27] Howe, A. E. *Accepting the Inevitable: The Role of Failure Recovery in the Design of Planners.* PhD thesis, University of Massachusetts at Amherst, February 1993. Also available as Computer Science Department Technical Report 93–40.

[28] Jain, R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling.* Wiley and Sons, 1991.

[29] Joslin, D., Nunes, A., and Pollack, M. E. TileWorld user's manual. Technical Report 93-12, Department of Computer Science, University of Pittsburgh, 1993. Contact `tileworld-request@cs.pitt.edu`.

[30] Keene, S. E. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[31] Korf, R. E. Real-time heuristic search: New results. In *Proceedings of the Seventh National Conference on Artificial Intelligence* (1988), vol. 1, American Association for Artificial Intelligence, Morgan Kaufmann Publishers, pp. 139–144.

[32] Laird, J. E., Newell, A., and Rosebloom, P. S. Soar: An architecture for general intelligence. *Artificial Intelligence 33*, 1 (1987), 1–64.

[33] Lesser, V. R., and Corkill, D. D. The Distributed Vehicle Monitoring Testbed: A tool for investigating distributed problem solving networks. *AI Magazine 4*, 3 (Fall 1983), 15–33.

[34] Lesser, V. R., Corkill, D. D., and Durfee, E. H. An update on the Distributed Vehicle Monitoring Testbed. Technical Report 87-111, University of Massachusetts at Amherst, Computer Science Department, October 1987.

[35] Lesser, V. R., Pavlin, J., and Durfee, E. H. Approximate processing in real-time problem solving. Technical Report 87-126, University of Massachusetts at Amherst, Computer Science Department, December 1987.

[36] Martin, N. G., and Mitchell, G. J. A transportation domain simulation for debugging plans. Obtained from the author, `martin@cs.rochester.edu`, 1994.

[37] McDermott, D. A reactive plan language. Technical Report YALEU/CSD/RR #864, Yale University, New Haven, CT, Aug. 1991.

[38] McDermott, D. Transformational planning of robot behavior. Technical Report YALEU/CSD/RR #941, Yale University, New Haven, CT, Dec. 1992.

[39] Montgomery, T. A., Lee, J., Musliner, D. J., Durfee, E. H., Damouth, D., So, Y., and the rest of the University of Michigan Distributed Intelligent Agents Group. MICE users guide. Tech. rep., Department of Electrical Engineering and Computer Science, University of Michigan, Mar. 1994.

[40] Newell, A. *Unified Theories of Cognition.* Harvard University Press, 1990.

[41] Nguyen, D. The Truckworld project. Distributed with Truckworld software.

[42] Nguyen, D., Hanks, S., and Thomas, C. The TRUCKWORLD manual. Technical Report 93-09-08, University of Washington, Department of Computer Science and Engineering, 1993. Contact `truckworld-request@cs.washington.edu`.

[43] Niehaus, D. *Program Representation and Execution in Real-Time Multiprocessor Systems.* PhD thesis, University of Massachusetts at Amherst, 1994.

[44] Parr, R., Russell, S., and Malone, M. The RALPH system. Manual with the distribution of RALPH `ftp::snake.cs.berkeley.edu/pub/ralph.tar.gz`, September 1994.

[45] Philips, A. B., and Bresina, J. L. NASA tileworld manual: System version 2.0. Tech. Rep. TR-FIA-91-11, NASA Ames Research Center, AI Research Branch, Mountain View, California, May 1991. Contact `tileworld@ptolemy.arc.nasa.gov`.

[46] Philips, A. B., Swanson, K. J., Drummond, M. E., and Bresina, J. L. A design rationale for NASA Tileworld. Technical Report TR-FIA-91-04, NASA Ames Research Center, AI Research Branch, Mountain View, California, 1991. Contact `tileworld@ptolemy.arc.nasa.gov`.

[47] Pollack, M. E., and Ringuette, M. Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proceedings of the Eighth National Conference on Artificial Intelligence* (1990), American Association for Artificial Intelligence, MIT Press, pp. 183–189.

[48] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. *Numerical Recipes in C: The Art of Scientific Computing*, 2 ed. Cambridge University Press, 1992.

[49] Rihar, M. The software simulator as an effective tool for testing control algorithms. *Simulation 63*, 1 (1994), 6–14.

[50] Rosenbloom, P. S., Laird, J. E., and Newell, A. A preliminary analysis of the soar architecture as a basis for general intelligence. *Artificial Intelligence 47*, 1–3 (1991), 289–325.

[51] Russell, S., and Wefald, E. Decision-theoretic control of reasoning: General theory and an application to game playing. Technical Report UCB/CSD 88/435, UC Berkeley, October 1988.

[52] Russell, S., and Wefald, E. *Do the Right Thing: Studies in Limited Rationality.* Artificial Intelligence Series. The MIT Press, Cambridge, Massachusetts, 1991.

[53] Russell, S. J., and Wefald, E. H. On optimal game-tree search using rational meta-reasoning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (1989), pp. 334–340. Detroit, Michigan.

[54] Russell, S. J., and Wefald, E. H. Principles of metareasoning. *Artificial Intelligence 49* (1991), 361–395.

[55] Shen, J., and Butler, S. Performance modeling study of a client/server system architecture. In *Proceedings of the 1994 Winter Simulation Conference* (1994), J. D. Tew, M. S. Manivanna, D. A. Sadowski, and A. E. Seila, Eds., Institute of Electrical and Electronic Engineers.

[56] Smith, C. U. *Performance Engineering of Software Systems.* Addison-Wesley, 1990.

[57] Steele Jr, G. L. *Common Lisp: The Language*, second ed. Digital Press, 1990.

[58] Thomas, C., Hanks, S., and Nguyen, D. Multiprocess and multiagent operation of the Truckworld. Technical report, University of Washington, 1993. (Forthcoming; supplied with Truckworld code.).

[59] Zilberstein, S. *Operational Rationality through Compilation of Anytime Algorithms.* PhD thesis, University of California at Berkeley, 1993. Department of Computer Science.