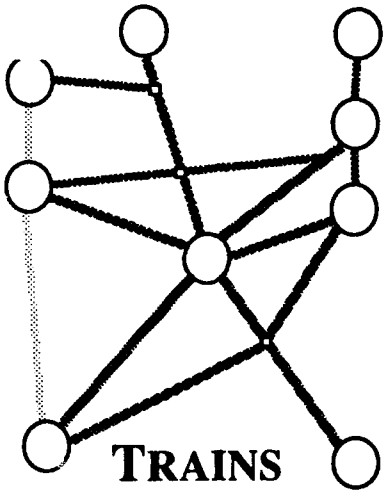

The TRAINS-90 Simulator

Nathaniel G. Martin and Bradford W. Miller

TRAINS Technical Note 91-4
May 1991

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

The TRAINS-90 Simulator

Nathaniel G. Martin and Bradford W. Miller

The University of Rochester
Computer Science Department
Rochester, New York 14627

TRAINS Technical Note 91-4

May 1991

Abstract

The goal of the TRAINS project is to design a computerized intermediary between a user and a set of agents capable of executing action in a transportation/production domain. The TRAINS-90 simulator simulates both the domain and the agents. It allows graphical editing of the world, and communication with the agents, either directly via the console or indirectly via remote procedure calls. The simulation has been designed to be easy to extend either by adding more intelligence to the agents, or by adding more actions the agents can invoke. This document describes the TRAINS-90 simulator at five levels of detail. At the first level, it describes the operation of the simulator while running the example developed over the summer of 1990. At the second level, it shows how to make modifications to the scenario used in that example. A more thorough description of the simulator at the user level is then provided. A programmer's view of the simulator follows for those who need to make modifications to the program itself. Finally, a sketch of the process of making changes to the simulator is provided.

This material is based on work supported by ONR/DARPA under grant number N0014-82-K-0193, by AF-Rome Air Development Center under contract number F30602-92-C-0010 and by ONR under grand number N00014-90-J-1811.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: Technical Note 91-4	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: The TRAINS-90 Simulator	5. TYPE OF REPORT & PERIOD COVERED: technical note
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): N.G. Martin and B.W. Miller	8. CONTRACT OR GRANT NUMBER(s): N00014-82-K-0193, N00014-90-J-1811
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Dept., University of Rochester, Rochester, NY, 14627, USA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA, 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE: May 1991
	13. NUMBER OF PAGES: 50 pages
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Office of Naval Research, Information Systems, Arlington, VA 22217	15. SECURITY CLASS. *(of this report)*: unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

planning; natural language understanding; plan execution; multi-agent reasoning; temporal reasoning

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT

The goal of the TRAINS project is to design a computerized intermediary between a user and a set of agents capable of executing action in a transportation/production domain. The TRAINS-90 simulator simulates both the domain and the agents. It allows graphical editing of the world, and communication with the agents, either directly via the console or indirectly via remote procedure calls. The simulation has been designed to be easy to extend either by adding more intelligence to the agents, or by adding more actions the agents can invoke. This document describes the TRAINS-90 simulator at five levels of detail. At the first level, it describes the operation of the simulator while running the example developed over the summer of 1990. At the second level, it shows how to make modifications to the scenario used in that example. A more thorough description of the simulator at the user level is then provided. A programmer's view of the simulator follows for those who need to make modifications to the program itself. Finally, a sketch of the process of making changes to the simulator is provided.

Contents

List of Figures

1 Introduction

The TRAINS project [Allen and Schubert, 1991] is an ongoing research effort at the University of Rochester that combines research in planning and natural language processing. The project consists of two conceptually different components: the system and the simulation. The system plays the part of a mid-level manager sending orders to agents. Simulated agents then try to fulfill the managers orders by performing actions in the domain provided by the simulation. The user converses with the system by typing natural language utterances at the console. These utterances are processed and interpreted in light of the system's knowledge of the current state of the world. Goals and means of achieving these goals are extracted from this interpretation and are used to develop a plan that will achieve as many goals as possible. The system includes two main components: a natural language understanding system, and a planning system containing both a discourse planning system and an action planning system. The natural language component is described in [Schubert, ming] and [Light, 1991]; the discourse planning component is described in [Traum, 1991]; and the action planning system is described in [Ferguson, 1991]. This document describes the simulation.

The TRAINS domain was inspired by the ARMTRAK domain [Martin *et al.*, 1989]. Like ARMTRAK, the TRAINS domain involves planning about the operation of trains. There are several notable differences, however. The ARMTRAK domain was designed to allow implementation on the Rochester Robot [Brown *et al.*, 1988], whereas the TRAINS domain is not so constrained. The TRAINS domain, on the other hand, includes multiple simulated agents allowing more complex conversations. The primary difference between the TRAINS domain and the ARMTRAK domain is one of perspective. The perspective of the ARMTRAK domain is of a single entity looking down over a train layout. The ARMTRAK domain is that of a person playing with toy trains. The TRAINS domain is a simulation of a real rail system. A planner can only gather information by requesting it from the appropriate agents, and can only effect the world by sending commands to those agents capable of executing them.

The TRAINS-90 simulator is built from two components: the world simulator and the agent simulator. The world simulator simulates the physical constraints on trains and factories. The agent simulator simulates the capabilities of the agents in the world. For example, a simulated engineer can execute commands like "Goto city I" by looking for the most direct route from its current position to city I, notifying the control center of its intentions, and adjusting the throttle so that it moves down the track to city I as fast as possible. The world simulator provides the map that the engineer looks at to find a route to city I, transfers the radio message from the engineer to the control center, and keeps track of the train's position as the train accelerates due to the increased throttle.

There are two types of agents in the TRAINS domain: engineers and managers. Engineers control trains by setting a throttle or brake, and can gather information about the world by looking around, listening, talking on the radio, etc. Managers control their plant and gather information by giving commands to workers. These actions all take time. Agents do not have privileged information about the simulated world; they have to make requests to get information. Their information can be wrong (if they figure it out themselves), or out of date (if it comes directly from the simulator). For example, an engineer only knows what is in the cars of the its train if it gets out and looks. Such an inventory takes time, so engineers perform such actions sparingly. The simulator simulates physics for the engineer and the action of factory workers for factory managers. The simulator runs in its own process and the agents execute actions by making RPC "world requests".

Figure 1 shows the lines of communication within TRAINS-90. The user communicates with the system via the natural language interface. The system communicates with each of the agents by sending "radio messages", that are generated by the executor. The agents may also send information back to the system by radio message. The system can only gather information and cause effects

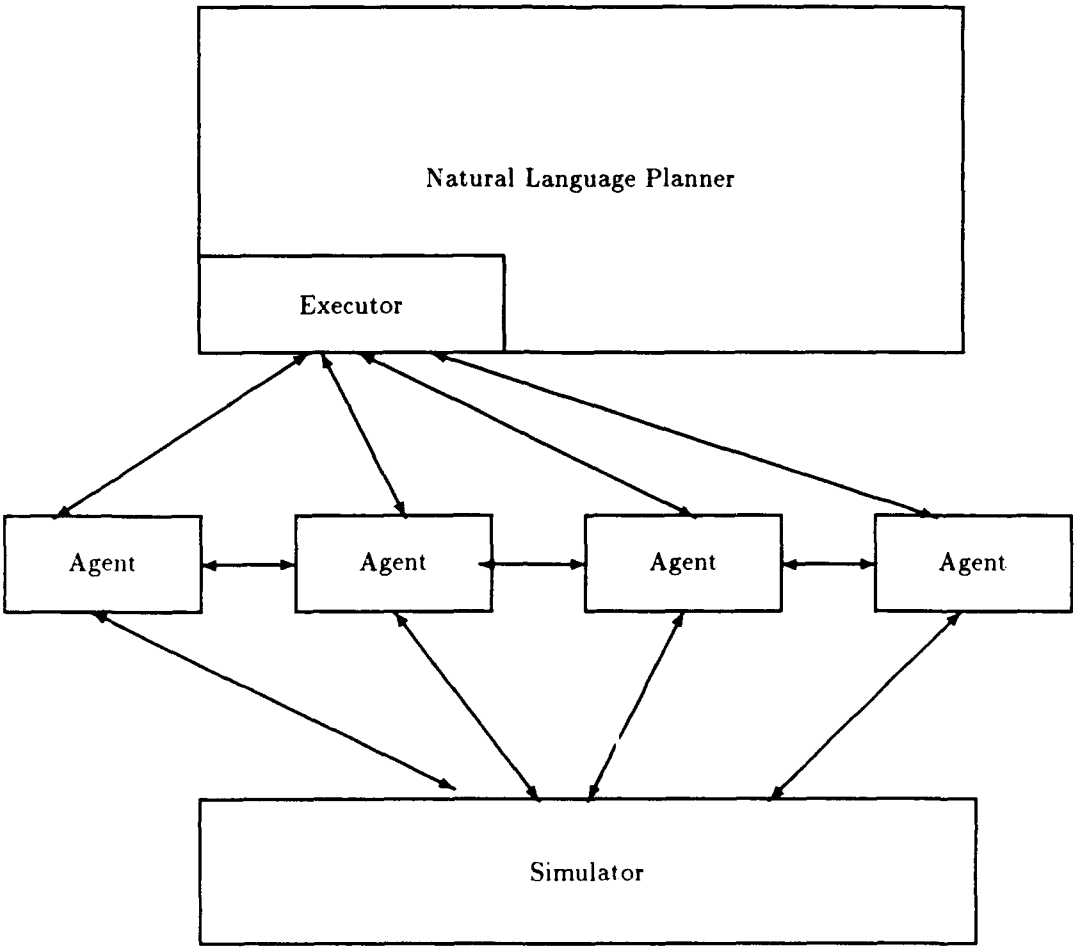

Figure 1: Communication within TRAINS-90

by radio messages to the agents; the agents can only gather information and cause effects through requests to the simulator.

The two level design of the simulator allows experimentation with a wide range of issues in both planning and natural language processing. Because the agents are entities separate from the system, it allows conversations about the knowledge and competence of a third person. Such conversations require an appropriate representation for knowledge of this type. Moreover, because the planner cannot execute its actions directly, it must reason about the competence of the simulated agents. Because there are multiple agents, the planner must reason about the best agent for a particular job. The world simulation introduces issues of time and uncertainty into planning and conversation. The conversations between the user and the system will be complicated by the lag between the issuing and the execution of commands, requiring conversations with complex tense structure. The system must also be able to plan for the time lag between issuing order and the effects occurring. It also faces deadlines as time proceeds apace in the simulator regardless of the activity of the system. Moreover, due to the strict separation of the system from the simulator–they are designed to run on separate machines–the simulation provides grounding for the knowledge representation used for both natural language processing and for planning. One may evaluate the success of the system by

Human We need to make OJ.

Human There are oranges at I and an OJ factory at B.

Human Engine E3 is scheduled to arrive at I at 3:00 PM.

Human Shall we ship the oranges?

System Yes.

System Shall I load the oranges into the empty car at I.

Human Yes.

Human We'll have E3 pick them up when it arrives.

Human OK?

System OK.

Figure 2: Example conversation between system and human

examining the effects of the system's actions.

The simulator is designed to be modified and upgraded as necessary. This flexibility allows experimentation in other aspects of reasoning. Designing different reasoning strategies for the agents allows experimentation in inter-agent communication and cooperation. Moreover, because the physics is simulated to a relatively fine grain, it allows experimentation with agents who perform routine actions in a complex world.

The TRAINS-90 simulator runs under Genera 8.0.1. It displays a scenario and allows the user to modify the scenario through mouse clicks. A modified scenario can then be saved to disk for later retrieval or commands can be sent to the agents in the scenario. Once the agents have commands the simulation begins. The simulator also allows communication with the system via remote procedure calls so the user can watch the scenario change as the system gives commands to the agents.

During the summer of 1990, a scenario in the TRAINS-90 simulator was developed. This scenario contains three cities, one engine, and two cars (see figure 3). The user instructs the system how to achieve the goal of generating oranges juice through the conversation shown in figure 2. The system processes the users instructions, develops a plan based on the user's suggestions, and executes the plan by sending messages to the appropriate agents. The next section sketches the system running *through the example scenario*. It provides an overview of the capabilities of the simulator.

The existing scenario provides a wide variety of possible conversations, but the small number of objects in the world may eventually prove limiting. The third section gives an example of altering this scenario. It shows how one could add a new city and new tracks to the existing scenario.

The fourth section sketches the capabilities of the simulator. It describes the structure of the simulator, and means of communicating with it. This section provides information for those who would like to generate scenarios that are different from the example.

The fifth section provides a overview of the workings of the program. This information is useful for those who would like to alter the behavior of the simulator. The system is designed so that different algorithms for the simulated agent's reasoning can be easily added. To a lesser extent, new actions may also be added. The section first describes the simulated physics and then the simulated agents.

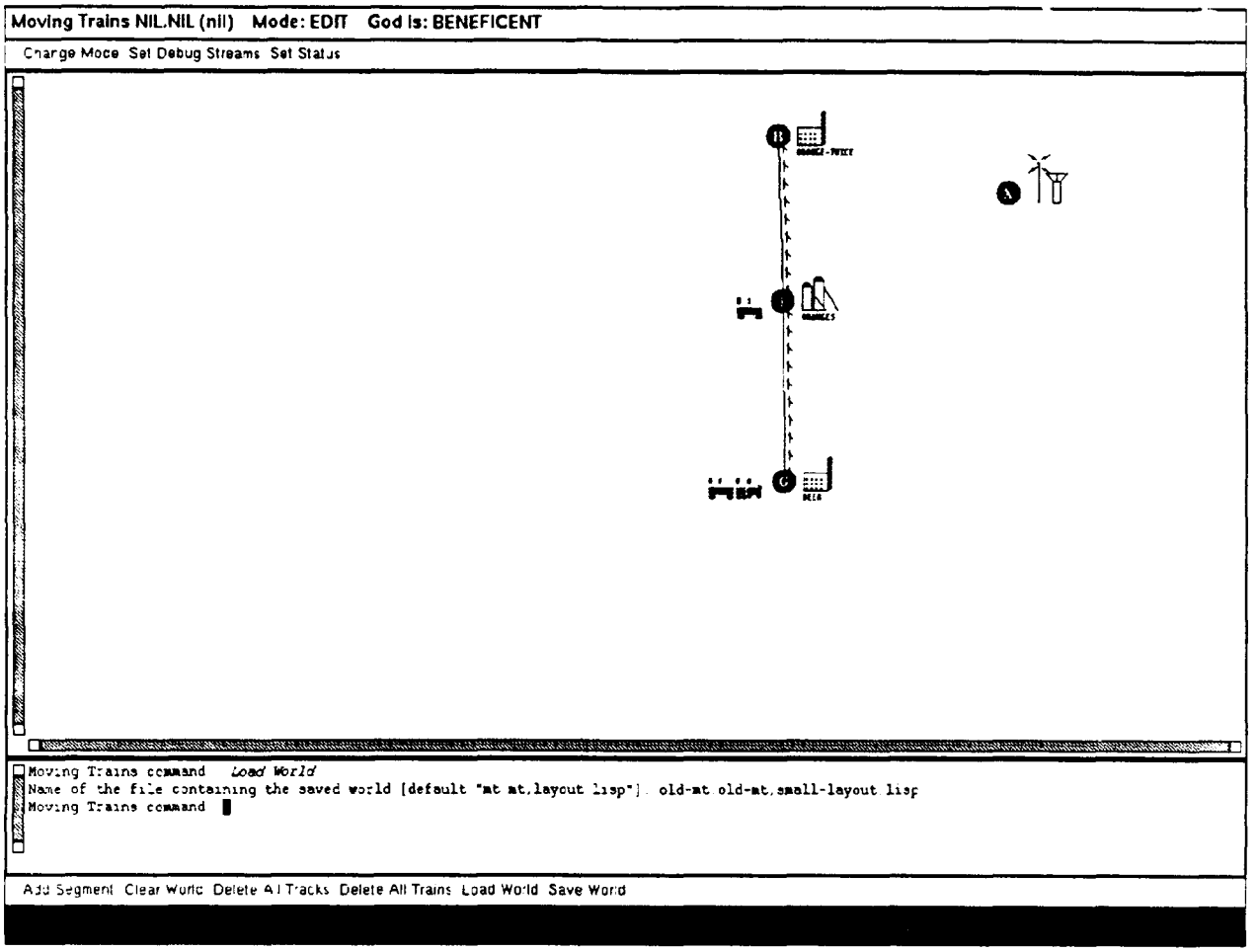

Figure 3: Scenario for the example conversation in edit mode

In the sixth section, the fifth section is elaborated with information about how to alter the system for different applications. This section shows how to add new features to the simulation, first by describing how to add new features to the world, then by describing how to add new features to the agents. Such features might be different commands the agents could respond to, or different primitive actions the agents could choose. For example, one might want to have engineers who could plan routes that are not the shortest possible. One might also want to add a new primitive action the agents could attempt. For example, one might want agents to unload cars by dumping them. Such an action would require a special car and a special unloading station, but would result in much quicker unload actions.

2 A First Example

The scenario for the example conversation is shown in figure 3. This scenario contains three cities: City G, City B, and City I. There is an orange juice factory at City B, a production center, which produces oranges, at City I, and a beer factory at City G. There is a single engine, E3, initially at City G, and two box cars, one connected to E3 at City G, and the other at City I. There are tracks

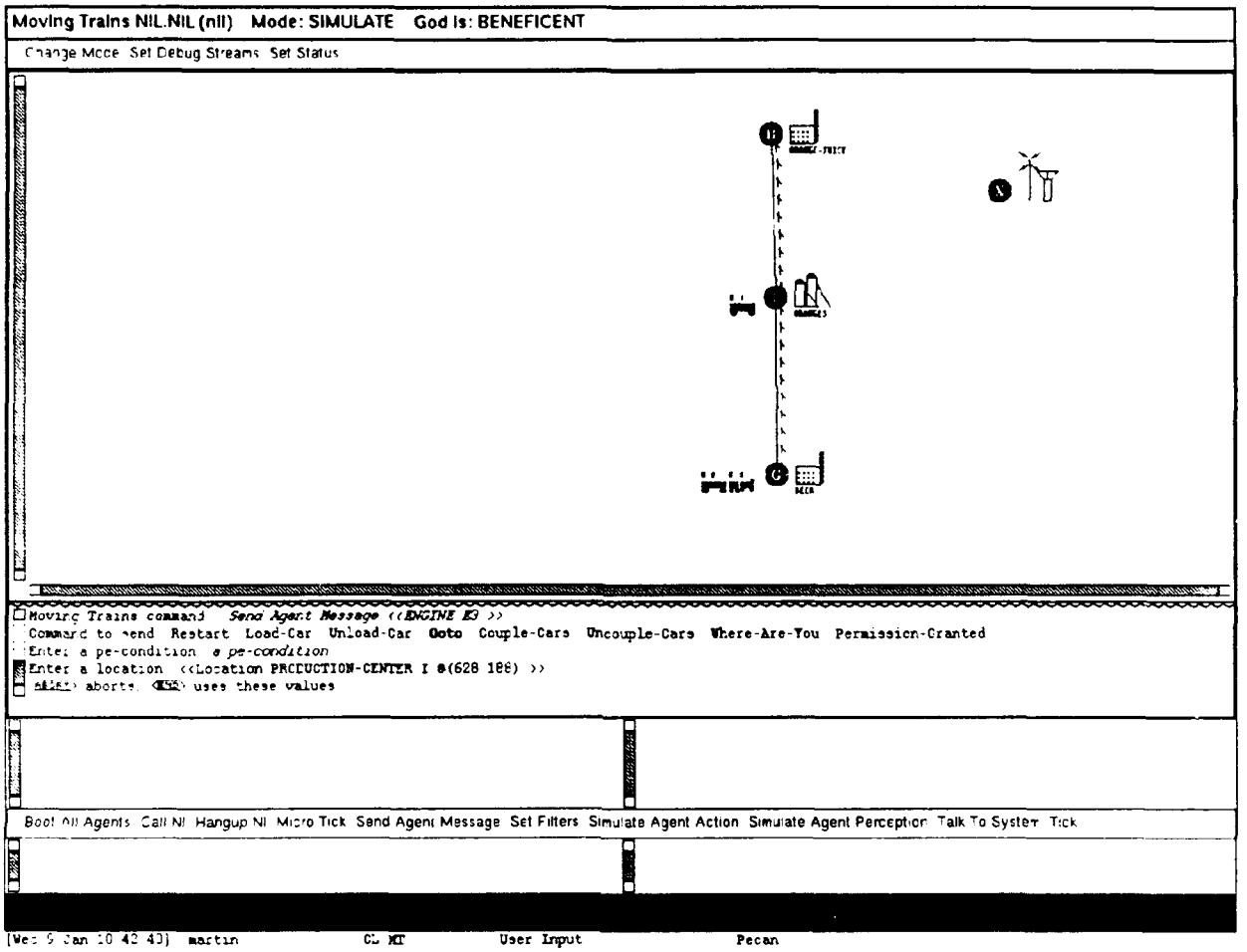

Figure 4: Scenario for the example conversation in simulate mode

between City G and City I, and tracks between City I and City B. City I has 200 tons of oranges in its silos, but both factories are idle. Both cars are empty.

The initial example conversation between the user and the system is shown in figure 2. From this conversation the system infers that the user wants to make oranges juice using the oranges at city I, the orange juice factory at city B, the engine E3, and the car B1. It generates the following plan from these inferences.

```
((((AT ENG3 CITYI)) (COUPLE ENG3 ENG3 C1))
 (((IN C1 O1) (AT C1 CITYB)) (UNLOAD F1 C1))
 (ANYTIME (LOAD W1 C1 1))
 (((IN C1 O1) (AT C2 CITYI) (COUPLED ENG3 C1)
   (AT ENG3 CITYI))
  (GOTO ENG3 CITYB))
 (((HAS_ORANGES F1)) (RUN F1)))
```

The executor takes this plan and generates a sequences of messages to individual agents in the scenario. It sends the following forms via TCP connection to be evaluated by the simulator.

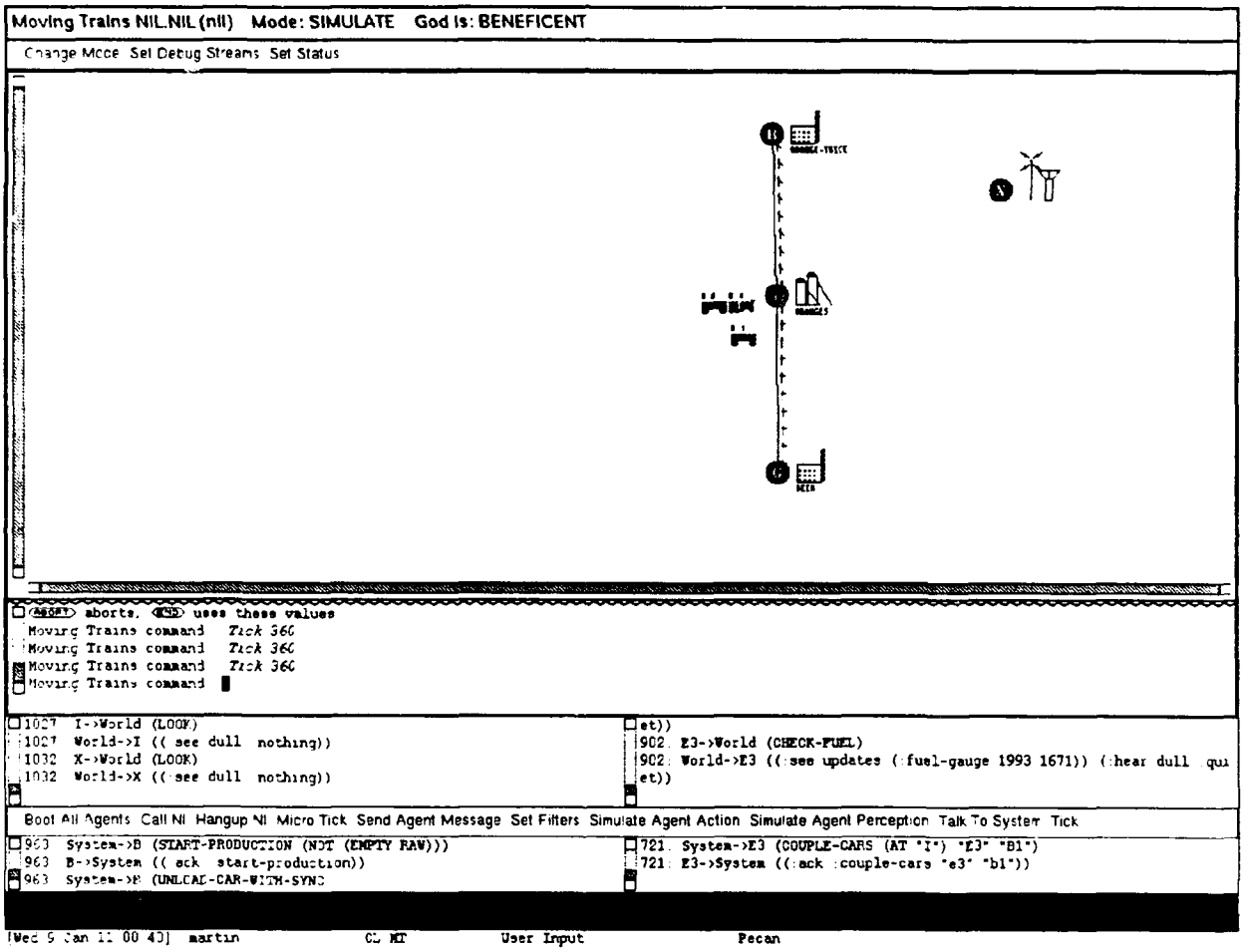

Figure 5: Appearance of layout when the train reaches City I

```
(SEND-MESSAGE "E3" 'COUPLE-CARS '((AT "I") "E3" "B1"))      (1)

(SEND-MESSAGE "B" 'UNLOAD-CAR-WITH-SYNC                    (2)
 '((AND (NOT (EMPTY "B1"))
        (HERE "B1"))
   "B1" ORANGES 1))

(SEND-MESSAGE "I" 'LOAD-CAR-WITH-SYNC '(ANYTIME "B1" 1 1))  (3)

(SEND-MESSAGE "E3" 'GOTO                                   (4)
 '((AND (NOT (EMPTY "B1"))
        (AT "I")
        (ON-TRAIN "E3" "B1"))
   "B"))

(SEND-MESSAGE "B" 'START-PRODUCTION '((NOT (EMPTY RAW))))   (5)
```

Each agent then executes the actions appropriate to these commands. Because each agent responds only to the commands directed to it, the behavior of the agents must be coordinated. The planner coordinates the agents through the conditions placed on the commands. The condition associated with these commands appear at the front of the list that appears at the fourth position in the form. For example, form (1) indicates that E3 should only start coupling cars when (AT I) becomes true. The order of the executor's messages are unimportant because these conditions insure that the actions will be executed in the correct order. In these particular instructions, the production center at City I first starts loading the oranges into B1 (3). When E3 gets to City I, it couples with B1 (1). Once the couple has completed and the oranges have been loaded, it leaves for city B (4). When the factory manager at city B notices that the train has arrived, it unloads the oranges (2). When it realizes that it has completed the unloading, it starts making orange juice (5).

All of the messages passed between the agents and the world and between the agents and the system are shown in the window just above the commands. The cities' conversations appear on the left and the factories' conversations appear on the right. An annotated trace of the example appears in Appendix A.

Interactions between the factory managers and the world are shown in the window directly to the left of the window showing interactions with the engineers. There are four factories in the scenario, X, a control center, B, an orange juice factory, I, an orange production center, and G, a beer factory. The interactions are labeled with the name of the factory.

When the commands are executed, the engine first goes to City I. When the engine has reached the city the display will appear as in figure 5. The manager of the production center at city I will have already started loading the oranges onto the empty car there. The engineer asks the city manager to couple the car that is being filled with oranges to its engine, then waits for the coupling to occur. The manager finishes loading the oranges then couples the car to the engine. Once the cars have been coupled the display will appear as in figure 6 and the engineer will head off to city B. When the train gets to city B, the engineer stops; it does nothing else. The manager of the factory at city B notices that the train has arrived, and orders its workers to unload the car. When the oranges have been unloaded, the manager recognizes that the conditions for making orange juice have been satisfies, so it starts the factory. When the oranges are processed, the manager stops the factory.

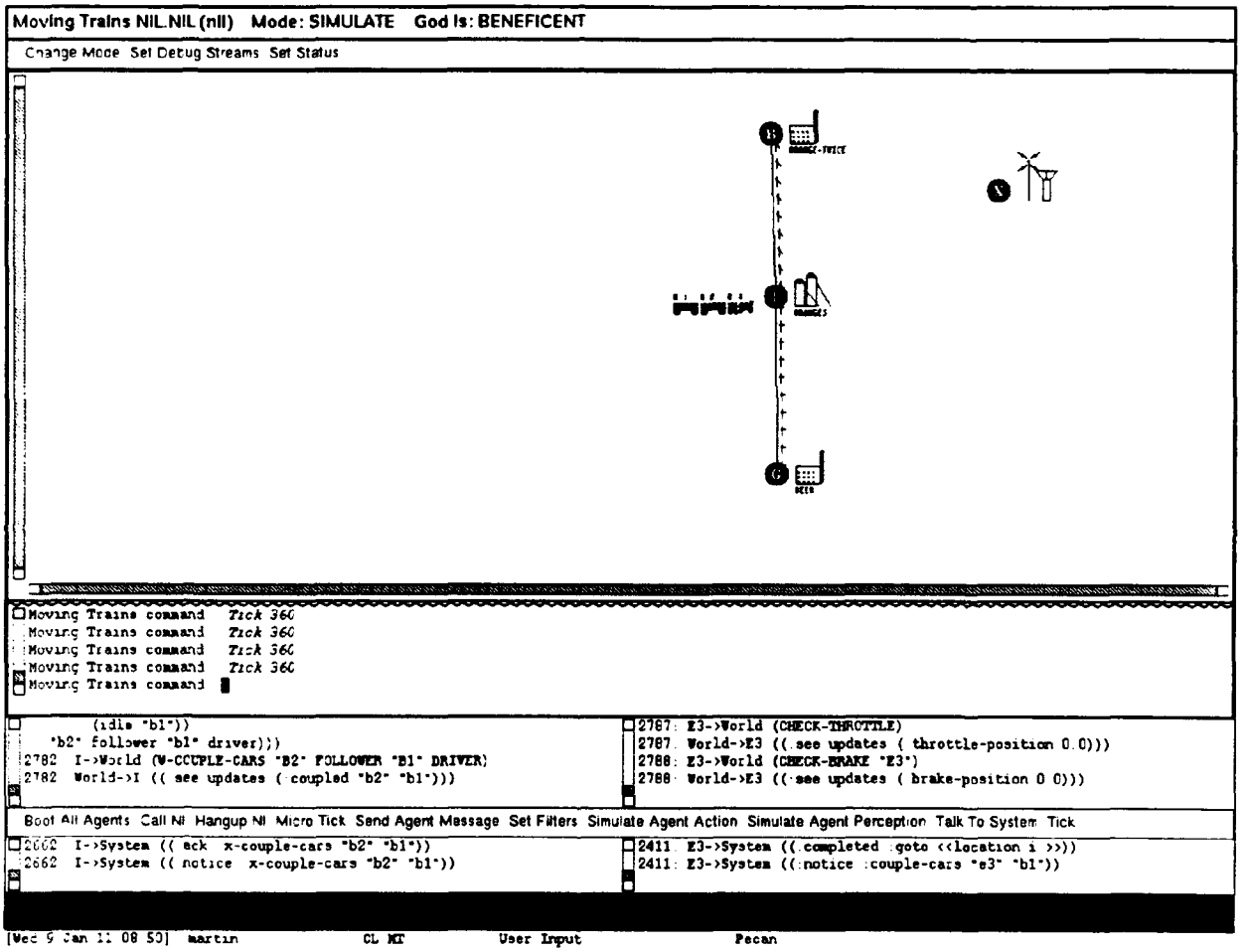

Figure 6: Appearance of layout when the boxcar B1 is coupled to the train.

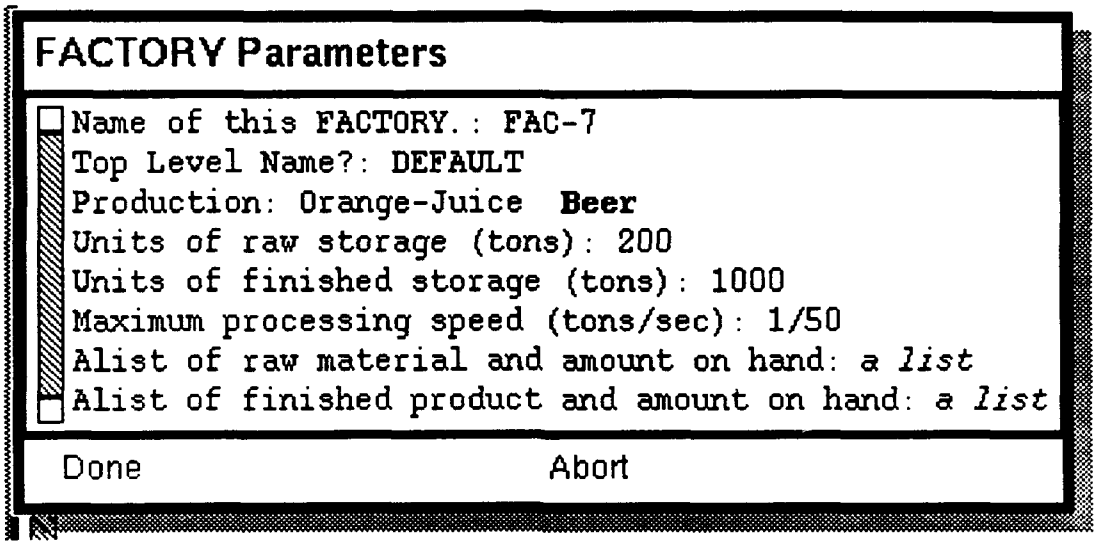

Figure 7: Menu for editing the parameters of a factory

3 Developing a Scenario

The simulator was designed to be a flexible tool for exploring the complexity of interacting with multiple agents in scenarios of varying complexity. To facilitate this flexibility, the simulator gives the user the ability to add and delete cities, train cars, routes and to specify their properties. One accesses *these editing functions* through the mouse. The edited scenario can then be saved by choosing the command "Save World" from the command menu.

As an example of editing a scenario, suppose the example scenario needs to be extended by adding a new city, A, and track running from City A to City G. Scenarios are developed from the edit screen. The edit window is called up by clicking left on "Change Mode" if the current mode is "Simulate".

3.1 Mouse Gestures

Mouse gestures provide the easiest way to generate or alter scenarios. By clicking appropriately on the mouse, objects can be added or deleted and information about the object can be obtained. Brief information about the results of possible mouse clicks is shown at the bottom of the screen.

To extend the example scenario by adding a city named City A and tracks from this city to City G, first click left on empty space where the new city should be. By answering "Y" to the question "Edit Defaults" a menu similar to that shown in figure 7 will appear. The name of the factory will appear as the label of the city at which the factory appears. That is, the name of the factory at City B is "B". To change the values of the defaults in this menu, click left on the value of the item. The item will disappear and a new value can be entered. The "Top Level Name" is the name of the type of agent which runs the factory, This should remain "DEFAULT" if the only purpose is altering the scenario. The alists of raw and finished products and amounts on hand appear as a list of dotted pairs, the first element of which is the item, the second element of which is the amount. For example, if this factory has 20 tons of oranges on hand the alist of raw materials would be "((oranges . 20))". The other items are self explanatory. When the values for the city are set appropriately, click left on "Done". The menu will disappear and the city will appear.

To add the track, move the mouse to City A and click middle. Notice that the mouse action window at the bottom of the screen indicates that this action will "Start a segment from here". As indicated, the action will begin a segment of track starting at the City A. Moving the mouse to City G will change the value for the middle button to "End a segment from here". Clicking middle will generate a segment of track starting at City A and continuing to City G. If you answer "Y" to the question "Edit Defaults?" a menu will appear. The most important option will be the control center that controls the segment. The control center sets the switches and signals on this segment of track allowing the engineer to move faster. To make use of the control center, the engineer has to ask for permission to use the track. The control center then monitors trains on the track and signals the engineer when the track is safe. The engineer can then safely drive faster than it could if it had to stop within its line of sight. Details of control center operation are presented below.

3.2 Windows

There are six commands on the edit menu towards the middle of the screen. These commands are described in more detail below. The three most useful menu commands are: "Clear World", "Load World", and "Save World". It is easier to add segments using mouse gestures, and usually when one wants to delete all tracks or trains one really wants a clean slate. "Clear World" restarts the editing process this way. Loading and saving worlds are useful commands for saving state between sessions.

Once changes have been made, they can be saved by clicking left on "Save world". A prompt asking for the name of a file to save the information will appear. Any fully qualified file name can be entered here. The default filename will overwrite the current version of the scenario, but the Symbolics usually saves versions of files it overwrites, so old versions can be recovered. Once a simulation has begun, the state of the simulation can no longer be saved in this manner as the states of the agents controlling the trains and the cities are not saved. If a scenario in which an agent is active is saved, the restarted agents will have no idea of what they were doing when the scenario was saved.

To make more extensive changes, one might want to completely clear the screen. This is done by clicking left on "Clear World". A new scenario may called up then by clicking left on "Load World". This is a good way to restore a scenario if serious mistakes are made in editing an existing scenario.

4 Simulator Overview

This section provides a user's level overview of the simulator. It describes how the simulator can be used to generate scenarios, the different types of simulations that may take place using a particular scenario, and how to interface planners with the simulator. The first subsection describes the objects currently in the system. It details the parameters the user can set for each object. The second subsection describes the appearance of the window system and the operations available from it. The third subsection shows how to use the mouse to manipulate these objects. Defining new types of objects is described in the next section. At any time when using the simulator, pressing the "help" key will give a menu of operations appropriate to the current situation.

4.1 Objects

There are three broad categories of objects in the TRAINS-90 simulator: cars, tracks, and cities. Cars are objects that move along the tracks between cities, possibly carrying cargo. Tracks are the routes along which the cars travel. Cities are the destinations of the cars. Along a different axis, the objects may be separated into two categories: animate and inanimate. Animate objects have agents

Cars			
Name	*Cargo*	*Cooled*	*Powered*
Engine	none	No	Yes
Box Car	solid	No	No
Dump Car	solid	No	No
Reefer	solid	Yes	No
Tanker	liquid	No	No
Refrigerated Tanker	liquid	Yes	No

Figure 8: Table of types of cars

associated with them; inanimate objects have no agents and are incapable of independent action[1]. Agents can be associated with cars or cities. Animate cars are engines; animate cities are factories, production centers, or control centers.

Most objects in the system have names. Names are more important for those objects associated with agents as they have to be addressed by name, but the other objects also have names. Names are important in commands and conditions for commands. For example, even though a fork does not have an agent associated with it, specifying a route may require commanding an agent to go to that fork. Also, control centers send and receive radio messages from switched tracks by name.

Cars

There is only one type of animate car, an engine. Besides its name, the engine has two attributes that can be specified: the maximum horsepower, and the amount of fuel initially in the engine. The maximum horsepower will determine how quickly the engine can accelerate given the weight of the cars it is pulling.

Engines are the most complex cars as they are controlled by an agent. Each engine is equipped with a throttle and a brake that the engineer can set. If the engineer sets the throttle the engine will exert force on the track through the drive wheels unless it is wrecked or out of fuel. The engine slows due to track and air friction, so the throttle needs to be applied continuously to keep the train moving. The faster the train moves, the greater the air friction, so the engine uses more fuel as it moves faster. In the scenario described above, the train moves at 600 miles per hour, and uses almost 10,000 gallons of fuel for the fifteen mile trip. If the force is sufficient to move the engine and the train of cars attached to the engine it will move forward. To stop, the engineer can either put on the brake or run the motor in reverse or both. The amount of braking either from the engine or from the brakes is measured in horsepower. If sufficient horsepower is applied, the train stops. Breakaway is not currently simulated so the train can apply an arbitrary amount of force to the tracks. Future versions of the simulator will limit the maximum amount of force that can be applied by the wheels. Future versions will also take brake efficiency into account–brakes become less effective as they get hot.

There are five other types of cars: reefers, boxcars, dumpcars, tankers, and refrigerated tankers. Besides their names, these cars have five attributes that can be set. These attributes are: capacity, cargo, maximum loading speed, maximum unloading speed, and maximum braking power. The capacity determines the tonnage the car can hold. The cargo indicates the amount and variety of cargo the car holds initially. This item is an alist in which the key represents the type of the cargo and the value represents the number of tons of cargo of that type the car is carrying. For example, a

[1] This is an oversimplification as cars might roll down hill. Only agents are capable of engaging in purposeful action.

Track Types			
Name	*Connections*	*Switched*	*Type*
Straight	2	No	Straight
Signal	2	Yes	Straight
Marker	2	No	Straight
Fork	3	Yes	Connector
Triple	4	Yes	Connector
Cross	4	No	Connector
Clover	4	Yes	Connector
Decoupler	2	No	Decoupler

Figure 9: Table of types of track

car carrying 20 tons of oranges would have "(('oranges . 20))" as its cargo attribute. The maximum loading speed and maximum unloading speed of the cars represents the maximum speed at which the cars cargo can be loaded or unloaded. The intention is that it is faster to load or unload dump cars than it is to load or unload box cars or reefers. The default values for these attributes reflect this intention. The maximum braking power is the largest braking force in horsepower the car can exert.

Other than loading and unloading speed the attributes which distinguish the cargo carrying cars are speed of decay and liquid carrying capability. Perishable cargo spoils slower in the refrigerated cars–reefers and refrigerated tankers–than in the other types of cars. Tanker and refrigerated tankers can carry only liquids whereas the the other types of cars can carry only solids.

Box-cars, reefers, and dump cars are designed to carry solid materials. Box cars are good for carrying non-perishable items, but perishable items, like oranges, will begin to decay after a while. Oranges can be transported farther if they are kept in a refrigerated car. Both refrigerated cars and boxcars take about twenty minutes to unload. If the items being shipped are not fragile, they can be sent in a dump car. Dump cars are usually used for carrying coal. They have an open top so they can be loaded from hoppers, and they have hatches on the bottom that can be opened to allow their contents to be dumped. Care should be taken when using a dump car to make sure that they are dumped in appropriate locations. If they are dumped in a place where there is no unloading facility their contents will block the track until it is removed.

Tankers and refrigerated tankers are designed to carry liquids like orange juice. Like the reefers, the orange juice will keep longer in the refrigerated tanker than it will in the regular tanker.

Track

There are three categories of track: straight track, connections, and decouplers. There are three types of straight track: an ordinary straight pieces, markers and signals. Each piece of straight track is 70 feet long. Each pixel represents 10 track units, so each pixel is 700 feet across. A marker is a section of straight track that has a sign-post associated with it. An engineer can tell how far it is to the next city each time it passes a marker. In addition, each time a car crosses it, the marker notifies the control center that has responsibility for that segment of track. A signal is a piece of straight track through which the control center can communicate with the engineer. If the signal is up, the way is blocked; if it is down the way is clear.

There are four types of connections: forks, triples, crosses and clovers. A fork connects three segments with a switch. Triples connect four segments with a switch. Clovers connect fours segments of track in such a way that, having entered on any segment, it can leave on any of the others. Crosses

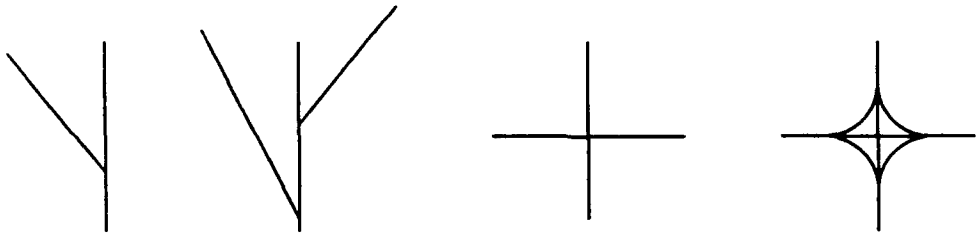

Figure 10: Types of connectors

are places where two segments of track cross each other, but do not allow the engine to move from one segment to another. The switches may be controlled by a control center which is entered as one of the attributes of the connectors. Alternatively, they may be set manually by the engineers. Figure 10 shows the different types of connectors.

Pieces of straight track are organized into segments of arbitrary length terminated by either a connector or a city. In addition to their name, straight segments also have five attributes that can be set when they are initialized: from, to, marker-frequency, signal-frequency, and the control center that controls the segment. The from and to attributes determine what the segments connects. The marker-frequency specifies the frequency of markers on the segment in turn determining the accuracy with which an engineer can calculate its position. The signal-frequency determines how often the engineer sees a signal that the track is clear. This will affect the engineers maximum safe speed. The control center sets the signals.

Cities

There are three types of cities: factories, production centers, and control centers. All cities have maximum amounts of raw and finished storage, amounts of raw and finished materials on hand, and a "top level name". The maximum amounts of raw and finished materials determines the cities storage capacity. The raw and finished amounts on hand are alists, the keys of which are the type of raw or finished materials, the values of which are the current amount of these materials. The "top level name" is the name of the program the agent that controls the city runs to do its reasoning. Cities differ according to type. A factory consumes raw materials and produces finished materials. A production center produces raw material, and a control center sets the switches establishing a routing protocol.

In the example described above, City G is a beer factory, City I is a production center producing oranges, City B is an orange juice factory, and City X is a control center. The manager at City B checks continuously to see that is silos are full, and, if they are not, fills them by buying oranges from the farmers in the neighborhood. City I can make orange juice given that it has oranges on hand, and a place to put the finished product. The manager at City X receives requests from the engines to use a segment of track, and, once received, grants these requests if the track is not otherwise in use. Once permission has been given, it sets the switches for the engine, and sets the signals to let the engineer know that the track ahead is clear.

4.2 Display

There are two modes of operation for the simulator: the edit mode and the simulate mode. The edit mode allows one to enter new objects into the scenario; the simulate mode runs the simulation. The simulator starts in edit mode.

Edit Mode Window

The simulator is in edit mode in figure 3. There are six main segments of the edit mode window. From top to bottom, these segments are: the mode display, the mode menu, the scenario display, the edit listener, the edit menu, and the mouse display.

The mode display indicates the state of the system. The information displayed is: the name of the program (Moving Trains), which mode it is in (Edit), and the amount of randomness in the simulation (God is:). The mode menu provides three menu options for changing the state of the simulation. The first option (Change Mode) changes the mode. Clicking left on this option changes to simulate mode if the simulator is in edit mode and changes to edit mode if the simulator is in simulate mode. Clicking right allows one to specify whether to go to simulate or edit mode. The second option (Set Debug Streams) allows one to set up files to capture messages among the human, the system, the agents, and the world as they scroll across the screen. Clicking left on this option generates two files "*ENGINE-1*" and "*FACTORY-1*". These files will be put in emacs buffers automatically. Clicking right allows one to select the names of the files. This option is helpful for debugging scenarios as it allows one to examine the operations of each of the agents after the fact. The windows in which the information is displayed can be scrolled, but they are small, and much information passes through them quickly. The third option (Set Status) allows one to control the amount on randomness in the system. Currently there is only one level available.

The scenario display appears directly under the mode menu. Here the state of the scenario is displayed. In this window mouse activity changes the state of the world. The effect of mouse clicks are determined by the surface over which the mouse is moving. The mouse is sensitive to four different types of surface: empty space, cities, railroad cars, and track segments. The mouse will act somewhat differently depending on the type of city, car, or track. For example, the mouse will act differently when it is over a city with a production center than it will over a city with a factory.

The fourth window displays a history of the commands sent to the simulator. This window can be scrolled so previous commands can be seen. When further information is required by a command, that information is entered here. For example, when the "Load World" command is selected, the prompt for the name of the file to load appears here.

The fifth window from the top, the edit menu, provides a selection of options that can be selected from the edit screen. Clicking left on the options from this menu will have the following effects:

Add Segment Adds a track segment. It prompts for a place at which the segment starts, then for a place at which the segment ends. These places may be entered by clicking right on an existing position in the layout.

Clear World Deletes everything in the world and reinitializes the simulation.

Delete all Tracks Deletes all tracks and all information relating to tracks.

Delete all Trains Deletes all railroad cars.

Load World Loads a file containing a world into the simulation. Prompts for the name of a file containing the world.

Save World Saves a world into a file. Prompts for the name of a file in which to save the world.

The mouse display is the sixth window from the top. It is a black box across the bottom of figure 3. This window gives brief information about effects of the mouse gestures in the current context. Keeping an eye on this window gives information about the operation of the program. Holding down the shift keys (shift, control, etc.) in any combination changes this line appropriately.

Simulate Mode Window

Figure 4 shows the simulate mode screen. This screen is divided into eight segments: the mode display, the mode menu, the scenario display, the simulate listener, the simulate menu, the world/agent communication display, the system/agent communication display and the mouse display. The mode display, the mode menu and the scenario display are the same as in the edit screen. The scenario display is somewhat smaller as there are more windows in the simulator screen. The hidden parts of the screen can be displayed by using the scroll bars.

The simulate listener acts like the edit listener, except, of course, the commands that are entered here are simulate commands rather than edit commands. Some of these commands appear in the next window down, the simulate menu. The effect of clicking left on these commands is as follows.

Boot All Agents restarts the agents. This is usually a bad idea if a simulation is running. The effect is that an engineer wakes up after brain surgery in an engine running at 300 MPH. Effects are predictable, but usually undesirable.

Call Nl establishes connection with the natural language/ planning system.

Hangup Nl closes a connection to the natural language planning system. Actually this command is of limited use as repeated "Call Nl" commands simply set up new planner processes, and the old ones eventually die.

Micro Tick simulates the passing of a single second. Usually this is a finer grain than is desirable.

Send Agent Message sends a message to an agent. A more efficient way of sending messages to agents is clicking left on the agent's icon.

Set Filters queries for type of commands and perceptions to filter. The output from these commands and perceptions will not be displayed. Setting filters speeds up the simulator.

Simulate Agent Action sends a command to the world as if an agent. This command prompts for the agent from whom this request is to be. The mouse gesture, shift-left, over the agent icon will also have this effect.

Simulate Agent Perception sends a perception to an agent. The command will prompt for the elements necessary. The mouse gesture, control-left, over the agent icon will also have this effect

Talk to System sends a stream of characters terminated by a carriage return to the Natural language processing/planning process.

Tick simulate the passing of a length of time. Clicking left simulates six minutes. Clicking right allows the entry of an arbitrary number of seconds. The number entered will become the default for subsequent left clicks.

The world/agent display window is divided into two halves. The left half shows the messages between the managers and the world; the right half shows the messages between the world and the engineers. Both requests for actions and perceptions are shown in this window. Perceptions and action reports are triples. The first item of the triple is the type of perception or action, the second

Object Type	*Mouse Gesture*	*Pointing To*	Mode
Mouse gestures generating cars			
Engine	Control-Left	Place	Edit
Reefer	Control-Right	Place	Edit
Boxcar	Control-Middle	Place	Edit
Dumpcar	Control-Shift-Left	Place	Edit
Tanker	Control-Shift-Middle	Place	Edit
Ref. Tanker	Control-Shift-Right	Place	Edit
Mouse gestures generating track			
Segment	Left	Place	Edit
Fork	Control-Left	Space	Edit
Triple	Control-Middle	Space	Edit
Cross	Control-Right	Space	Edit
Clover	Control-Shift-Left	Space	Edit
Decoupler	Control-Shift-Middle	Space	Edit
Mouse gestures generating cities			
Factory	Left	Space	Edit
Production-Center	Shift-Left	Space	Edit
Control-Center	Meta-Shift-Left	Space	Edit

Figure 11: Mouse gestures in edit mode

item is the filter value, and the third is the value of the perception or action. By setting filters on the filter values, display of reports of this type can be inhibited. For example, whenever the engine is running, the engineer hears engine noise. The filter type of this perception is "dull". By setting the filters for "dull", display of these perceptions will be inhibited.

The agent/system display window is also divided like the world/agent window. Communication between the system and the agents is displayed in these windows. Like the world/agent window, the left half shows communication between the system and the factories whereas the right half shows the communication between the system and the engineers. If the TCP connection to the system is being used, the communication between the human and the system will be displayed in the window dedicated to factories.

The seventh window shows the effects of mouse gestures in simulate mode. It acts like the mouse display in edit mode.

4.3 Mouse Gestures

The effect of mouse gestures depends on two parameters: the mode of the program and the position of the mouse. In edit mode, the user can add, delete, modify or get information about objects. Simulate mode allows the user to interact with the simulation. Here, one can send messages to agents by clicking on the agents icon.

Edit Mode

In edit mode, the mouse offers the ability to add, delete, or display information about the object to which the mouse is pointing. If the mouse is pointing to empty area, it will offer the option of putting a city at that position and associating with that city a factory, production center, or control

center. When the mouse is over an object to which tracks connect (eg. forks, cities), it will offer to start a segment if no segment is currently being defined, or to end a segment if a segment is being defined. In general, clicking right on the mouse will give a menu of options to perform. Usually, clicking left on an object will delete that object, and clicking middle will start or end a track segment from that object.

The mouse gestures for adding objects work only in edit mode. There are two types of objects to which the mouse can point: empty space or a non-empty place. If the mouse is over empty space, one can add a city or one of the track types that connect segments: forks, triples, crosses, or clovers. Cars can be added over places. In this case other cars act as places though they act differently in other aspects of the simulator. If a car is added at another car it is is added connected to that car. If a car is added at a city at which another car is currently situated, the cars are not connected. Segments can also be started over places. A table of the mouse gestures for adding objects in edit mode are shown in figure 11.

Information about the state of objects can be displayed from either edit or simulate mode. From edit mode, the command to display the information must be accessed through the menu pulled up by clicking right on the object. In edit mode, clicking middle will display the information about a place or a car.

Simulate Mode

In simulate mode there are two categories of object: those with associated agents and those without associated agents. If the mouse is over an inanimate object–one with no associated agent–it will offer information about that object. If it is over an animate object, it will offer the ability to send messages to the agent, to simulate perceptions of actions of that agent, to display information about the agent, or to display information about the object. As in the edit menu, clicking right gives a menu of options. In simulate mode, little can be done over empty space. The user of the simulation can communicate with agents by clicking left on the agent's icon.

4.4 Communication with Agents

One can also communicate with the agents by using the "Send Agent Message" from the simulate menu. The system communicates with agents by accessing the mt-eval server. To use the mt-eval server use "(mt-eval form)" where form is an s-expression that is to be evaluated in the simulator. Most commonly, the functions to be evaluated will be SEND-MESSAGE and RECORD-SYSTEM-UTTERANCE. These functions are used directly when the system needs to communicate with the simulation.

The function SEND-MESSAGE takes three arguments. The first argument is the name of the agent to which the message is to be sent. The second argument is the name of a command the agent is to execute. The third argument is the list of parameters to this command. The first of this list of parameters is a form which represents the conditions under which the command is to be executed. In the example described above, the system sent the following message:

```
(SEND-MESSAGE "E3" 'COUPLE-CARS '((AT "I") "E3" "B1")).
```

This message is addressed to "E3", the engine in the scenario, and requests this engine to do a couple action. This action is to occur when the engine finds itself "(AT "I")". The two parameters this particular command takes are the named of the cars to be coupled. In this case the parameters are "E3" itself and the car names "B1". The commands that can be sent to agents are described in appendix E; the conditions the agents can recognize are described in appendix D.

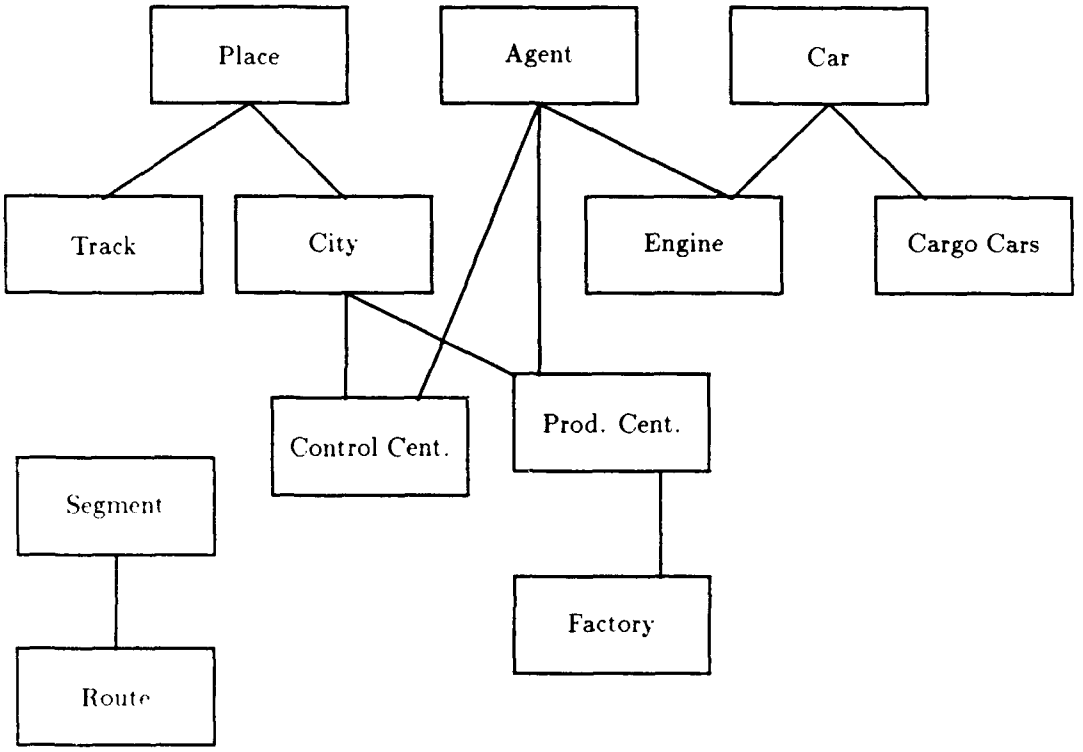

Figure 12: Sketch of the object structure of the simulator

The function RECORD-SYSTEM-UTTERANCE takes a string as a parameter. It simply prints that string on the system interaction window of the simulation screen. This function is useful when the user is communicating with the system through the simulation.

5 Program Overview

The simulator is written in Common Lisp using CLOS. It was designed and built as an object oriented program to make modification of the program easier. New objects based on the old objects can be added to the simulation by specializing the objects already in the system. Macros have been written to make this operation even easier. The macro DEF-MT-OBJECT generates all of the simulator's standard functions. As it defines the object, this macro also defines functions for adding the object, editing the object, adding mouse gestures for adding and editing the object, and loading and saving the object.

The simulator contains two types of programs. The simulation of the world, and the simulation of the agents. The world and each of the agents occupies a separate process. The system communicates with the agents by sending them condition command pairs. The agents add these pairs to a list and, whenever they have nothing else to do, search the list for the first one one whose conditions are met. The agent then attempts to fulfill this command. The conditions the agents can recognize are specified in appendix D. The commands the agents respond to are specified in appendix E. The agents respond to commands and recognize situations by executing a sequence of actions that effect the real world. These actions are specified in appendix F.

To perform actions, the agents send messages to the simulator. Each of the actions takes a specified amount of time during which the agent is put to sleep. The amount of time require for

Functions on "object"			
Name	*Effect*	*Info. Loc.*	*Num. Objs.*
COM-ADD-object	Add	Listener	Single
COM-EDIT-object	Edit	Listener	Single
ADD-object-HERE	Add	Mouse	Single
EDIT-THIS-object	Edit	Mouse	Single
COM-LOAD-object	Load	File	Multiple
COM-WRITE-object	Save	File	Multiple

Figure 13: Functions common to all objects

each action is described in appendix G. When the time the action takes has passed, the state of the simulated world is changed to represent the result of the agent's action and the agent is wakened. If the action the agent performed was a perception the information the agent requested is communicated to it.

5.1 Objects and Display

An outline of the structure of the existing objects is shown in figure 12. There are three basic classes in this hierarchy: places, agents, and cars. Places are specialized into production centers which are in turn specialized into factories. Agents are specialized in two ways. If an agent is associated with a car, that car is an engine. If an agent is associated with a place, that place is a factory, production center, or control center. Cars are also specialized in two ways: engines, and cars that carry cargo. Cargo cars are specialized into the five types of cargo carriers: boxcars, reefers, dump cars, tankers, and refrigerated tankers. Tracks are a kind of place also. These are further specialized into straight track and the different types of connectors: forks, triples, crosses and clovers. Sequences of straight track are technically another basic class. Usually instead of pieces of straight track the program deals with segments. Routes are segments that are specialized to run between cities.

All objects have six functions defined with them as shown in figure 13. The function, COM-ADD-object, generates a new object of the desired type. When this function is called, it first asks if the user wants to edit the default values associated with the object. If so, a menu of option to be changed is presented. COM-EDIT-object, displays a menu of the current slot values and lets the user change these values. ADD-object-HERE and EDIT-THIS-object, are functions associated with mouse gestures. ADD-object-HERE reads the position of the mouse and calls COM-ADD-object with these locations. EDIT-THIS-object called COM-EDIT-object on the object the mouse is over. COM-LOAD-object is similar to COM-ADD-object except that all of the object parameters ordered. This function is used to add an object to a scenario when that scenario is read from a file. COM-WRITE-object writes a COM-LOAD-object function with parameters that describe the object to a file. When such a file is loaded, the simulator executes the sequence of COM-LOAD-objects that were written by COM-WRITE-object, reloading the scenario.

The code to control the display uses the dynamic windows package from Genera. These functions will soon be replaced by functions based on CLIM. Changing to CLIM will make the code more portable. Since the design is in flux, this part of the program will be passed over in silence.

5.2 Simulated Physics

Physics is simulated at different levels of detail for engines and factories. For engines physics involves friction and momentum, whereas for factories, physics involves the actions of the factory workers.

The result of this different level of description of physics for factories and engines is the complexity of the actions the engineers and factory managers can perform. Engineers can do things like setting the throttle of their engine and checking the speed at which they are traveling; factory managers can do things like coupling cars.

Most of the simulated physics involves setting slots in objects to certain values. For example, the result of a factory manager coupling cars is that the coupler slots of the two cars to be coupled are filled with appropriate values. One notable difference to this generalization is the movement of the trains. When an engineer sets the throttle of an engine, the value of the throttle slot is changed in the object representing that engine in the simulation. At each time step, the simulation calculates the acceleration that would result from the power generated by the throttle setting and the current weight of the train, and updates the train's position using that value and the train's current speed. The corresponding situation for factories is production.

Time in the simulation is discreet: it moves forward in one second quanta. Time is incremented by the generic function, "TICK", which handles what happens in one time quanta.

Every action has a cost associated with it. These costs appear in appendix G.

Track

A piece of track is a place that can have a single occupant. More than one occupant is a train wreck. Next-track contains the next track that a car moving across the track in the "normal" direction will come to. A marker is a special kind of straight track, that can tell an engineer how far it is to the end of the segment. A signal is a special kind of straight track, that can tell an engineer that the track ahead of it is clear. A decoupler is piece of straight track on which an engineer can decouple one car from another. Presumably there is at least one decoupler in every city since factory managers can decouple cars at will. A fork has an entrance and two terminals. A train entering from the entrance will exit from the terminal to which the switch is set. Entering from the one of the terminals will place a train at the entrance if the switch is engaged correctly, otherwise it causes a derailment. A fork is a switched track with one entrance and two terminals. A triple is a switched track that allows a train to go from the entrance to three terminals. A cross is an unswitched track with four connectors (ie. the tracks simply cross, a train cannot move from one track to another). A clover is a switched track with four connectors that allows the train to move from any one of the four connectors to any other. (From any entrance, one can get to any exit.) A segment is a series of tracks with two ends. An end can be any place object. Except at their ends segments contain only straight track sections.

Track has two purposes in the physics. It determines when two trains wreck and, if it is a switched track, it determines the terminal on which the train will exit according to the state of the switch. Eventually, altitude will also be associated with track.

Trains

A car has a connector in front of it and behind it so it can be connected to other cars. Cars that carry things are called cargo cars. The cargo cars are: boxcars, reefers, dumpcars, tankers and refrigerated tankers. An engine is a car and an agent. In addition it has an orientation, a maximum HP and a throttle.

All movement of the train is relative to the engine. At each time step, the position of the engine is updated, then the position of each car attached to it is updated according to the direction it is moving and the connector to which the car is attached. The generic function that implements the movement of cars is called "ONE-STEP". The function "ONE-STEP" moves the train forward one

track segment. The segment onto which the train will move as determined by the connectivity of the track and the state of the switch is implemented by the generic function "NEXT-STEP".

For trains, "TICK" updates the position of the train. Most of the work of this method is performed by the function "UPDATE-TRAIN-POSITION", which is called only on the engine. In TRAINS-90, the engine is always at the front of the train. After the position of the train has been updated, the engine's remaining fuel is calculated. For cargo cars, the status of any loading or unloading operations is also updated. Generally loading and unloading operations can take place only while the car is in a city, but some cars may leak, particularly after a wreck or derailment.

Cities

The world of managers is simulated on a different level of detail from that of the engineers. For example, a factory consists of a factory manger and a number of factory workers; the manager's actions involve giving commands to the workers. A manager's actions have more direct results than an engineer's. For example, a factory manager can couple two cars by making a world request. The intuition is that a factory manager is simply giving this command to the engineer of a switch engine associated with the factory. An engineer would back into first car, then back the first car into the second car, then find a decoupler to decouple the first car.

A city can hold more than one car at a time and can terminate an arbitrary number of segments. There are three types of animate cities: production centers, control centers, and factories. A production center is a city that generates raw material. A control center is a city that advises other agents and controls switches. A factory can change raw materials into finished materials at a certain rate. At any one time a factory has a some amount of raw and finished materials on hand.

Different things change each tick according to the type of plant associated with the city. For factories, the amount of raw materials, and the amount of finished materials changes according the the setting of the speed slot on the factory object. If the factory is out of raw materials the factory manager is notified. For production centers, the amount of raw materials is updated at each tick. If the silos are full, the production center manager is notified. Nothing is updated for control centers, but some objects send them messages indicating the positions of the trains. (For example, a train moving by a switch or a signal sends the control center associated with that switch or signal a message.)

5.3 Simulated Agents

The agents described in this section are default agents. As of this writing they are the only implemented agents. The simulation was designed to allow the addition of agents of different types.

Commands to agents are condition command pairs. The agent waits until the condition is true, then puts the list of actions to be executed in a list associated with the variable CURRENT-ACTION. The agent then executes the actions in order.

Agents are each in their own process. This is desirable, because it give the agents responsibility for gathering information. It is necessary because, if the agent's beliefs were the same as the state of the world, changes in the state of the world would change the agents beliefs. If the agents were reasoning about the state of the world when the world changed, they could get into an inconsistent state. Putting each agent in a separate process does introduce some complexity in communicating with the world and each other, but this complexity is encapsulated in a few functions. Indeed, the encapsulation required by inter-process communication increases the safety of the system by decreasing the degree to which agents can interfere with the world and with each other.

The function WORLD-REQUEST sends a message to the simulator. It returns a list of observations indicative of the success of the act. Though most actions and perceptions take time, those

perceptions that are automatic in humans, have no such cost. For example, hearing a train wreck costs nothing as such a perception would presumably be instantaneous. If a request is empty, then all actions and perceptions with no cost are done. WORLD-REQUEST simulates the temporal cost of an agents action by putting that agent to sleep for the number of quanta necessary to do the request. The number of quanta that the agent sleeps for each action is described in appendix G.

It is expected that:

- Eventually agents will have limited processing time, and they will automatically lose quanta without doing any actions if none have been requested.
- Eventually agents will be able to process simultaneously with the simulator effecting actions, that is, multi-quanta actions will not cause the agent to sleep until they are finished.
- Multi-quanta actions will be interruptible (but at some cost to the agent in time).

The function AGENCY-REPORT sends messages from the agents to the system. This function returns nothing.

AGENT-BUILDER is a function of one argument, the agent structure the window system builds. It is called from the process of the window system, and will build an agency process to handle this agent. The agent may copy anything it wants from the agent structure, but should not keep a pointer to anything in the window process. In particular, it should not keep any part of the agent structure that is not copied. It is free to reuse the type definition, if it wants to copy the whole structure so it has some idea of itself. It can then update it by generating the right messages to the simulator process using WORLD-REQUEST.

The constants the world uses for its simulation should not be available to the agents. In agents currently in the system, approximations to the world's constants are displayed in appendix H.

Agents determine when they need to act based on anxiety. Determining when to act is particularly important for perceptions as the agent's knowledge gradually goes out of date. Anxiety is computed as the base raised to the power of the number of ticks since it was last updated, so the larger the base, the larger the anxiety. A base of one remains one no matter how long it is, and will never be updated, if other bases are larger. The agent's processing is considered an anxiety of five: the process executes an agent's command, then update variables until the anxiety of five forces it to go back to analyzing the current situation, setting throttles, etc. This means that if the agent itself has an anxiety level above five, it will never notice that new commands have arrived.

The actions the agents can perform are documented in appendix F.

Engineers

Since the engineer does not have privileged access to the world, it must be able to calculate its speed, location, throttle settings, and all of the other details of the world. Because perceptions take time, such calculations are less time consuming that looking whenever information is needed.

To move down the track an engineer must set the throttle to generate sufficient force to move the train it is pulling, but insufficient force to derail any of the cars. It must continue this acceleration until it reaches the velocity at which it can stop within its line of sight without derailing the cars it is pulling. Once it has reached this cruising velocity, it must decrease the throttle setting so it only overcomes the friction of the train it is pulling. When the train comes to a switch, the engineer must stop the train before it crosses the switch, get out of the train, set the switch, get back into the engine, and start moving again.

The engineer's task is simplified by control centers. Control centers monitor track segments allowing only one train at a time on that segment. The engineer can therefore move much faster

as it can be sure that it will not need to stop until it gets to the end of the segment. The control center also sets the switches for the engine and controls the signals along the track to reassure the engineer that no other train is on the track. The engineer can tell when it is nearing the end of the segment for which it has permission by watching the markers along the track. Watching the markers is necessary so the engineer can start braking before it gets to the end of the track. Since the switches and signals send messages to the control center, the control center can give early warning of an errant train on an unassigned track.

Unlike acceleration there are two ways to decelerate a train. Each car, including the engine, has a brake (though it costs a lot of time to use them, since, in the current model the engineer must visit any car other than the engine), and the engine can be run in reverse. Deceleration is particularly tricky when the train nears its destination. Because the engineer does not have completely accurate (within feet) knowledge of its position, it must make sure that it continues to move until it has reached the destination and not stop just a few feet short of the loading dock.

Managers

The agents associated with cities are simpler than the agents associated with trains as they do not need to deal with the complexities of the moving trains. The activities of the agents associated with cities differs a little according to the type of plant associated with the city. For example, factory managers can start the production of finished materials from raw materials, but control center managers and production center managers cannot. On the other hand control center managers can control switches but neither of the other two types of manager can.

All of the managers can load, unload, couple, and uncouple cars. They can also receive messages from the engineers telling them where the trains are. Unless the managers are working on a particular problem, they check the cars currently in their city and the amount of raw and finished goods on hand. Depending on the type of manager they are, they deal with this information differently.

The managers of production centers keep track of the state of their silos and start filling them if they determine that they are low. The silos empty when the manager loads cars from them. Factory managers are able to start the factories. Other than this they act much like production center managers.

Managers of control centers have a different job from the managers of production centers or factories. These managers need to keep track of the occupants of segments of track and give permission to use these segments to requesting engineers if the segments are free. They do this by setting the switches entering the segment appropriately and setting the signals on the segment. It then gives permission to the engineer to enter the segment.

Communication between Agents

Agents communicate through radio messages. Radio requests are generated by making a world request for a radio message to the desired agent. The radio message is generated by the simulation and is sent to the agent named. The agent to whom the message is addressed receives a perception that it has received a radio message at the next tick. The time it takes to send or receive a message depends on the number of words in that message.

5.4 Example

As an example of how the agents and the simulated world work together, consider an engineer coupling two cars. The current engineers are not smart enough to couple the cars themselves, so they must move to the city where the two cars are and ask the factory manager associated with that

city to couple the cars. (One of the cars could be the engine the engineer is driving, so the engineer could bring one of the cars to be coupled with it.)

When an engineer gets the a condition command pair of which the command is (COUPLE CAR1 CAR2), it first tests if the condition holds. If the condition does hold, the engineer then looks for the two cars to be coupled. If it finds them it radios a request to the manager to couple the cars, and waits until the manager has finished. The radio message to the factory manager is a condition command pair exactly like those the engineer performs, so the engineer must wait until the manager gets around to its request. It does this by sending itself messages to cooperate with the city. Once the cars are coupled, the manager notifies the agent that it has finished coupling the cars through another radio message.

The following shows a pseudocode trace of an engineer coupling two cars. The executor executes the command "SEND-MESSAGE (E3 'COUPLE-CARS '((AT "I") "E3" "B1"))". Because the engineer is unable to couple cars itself, it will comply with the request by forwarding it to the manager of a city which will in turn send it to a switch engine in the freight yard. The engineers inability to couple cars itself makes the command an interesting one as it require communication between the engineer and the appropriate city manager as well as control of its own actions.

By making such a request the executor causes E3 to get a radio message which E3 recognizes a "COUPLE-CARS" command and executes the COUPLE-CARS-COMMAND-HANDLER.

```
COUPLE-CARS-COMMAND-HANDLER (ENGINE CONDITION CAR1 CAR2)
BEGIN
   AGENCY-REPORT (:ACK :COUPLE-CARS CAR1 CAR2)
   PLAN ← PLAN ∪ (CONDITION (E-COUPLE-CARS CAR1 CAR2))
END
```

The COUPLE-CARS-COMMAND-HANDLER replies to the executor that it has received the message "(:ACK :COUPLE-CARS CAR1 CAR2)" and puts "((AT "I") (E-COUPLE-CARS CAR1 CAR2))" on its plan list. Whenever the engineer finds that it has nothing else to do, it looks on the plan list for the first item whose condition is true. If it finds that it is at city I, the engineer will execute "(E-COUPLE-CARS CAR1 CAR2)".

```
E-COUPLE-CARS (CAR1 CAR2)
BEGIN
   REPORT (:NOTICE :COUPLE-CARS CAR1 CAR2)
   CUR-PLAN ← ((DO-E-COUPLE CAR1 CAR2))
END
```

As can be seen, this function reports to the executor that the engineer is beginning the process of coupling cars, and it puts the list "((DO-E-COUPLE CAR1 CAR2))" on the variable CUR-PLAN. This variable holds a list of actions to do. In this case the action consists on the single action DO-E-COUPLE. The engineer gets the first element of the list on CUR-PLAN and executes it.

```
DO-E-COUPLE (CAR1 CAR2)
BEGIN
   IF (Not-in-city)
      ERROR (:NOT-IN-CITY :COUPLE-CARS CAR1 CAR2)
   Slot1 ← empty-slot (car1)
   Slot2 ← empty-slot (car2)
   IF (Not (slot1 and slot2))
      ERROR (:NOT-AGENT-OF-BOTH-CARS :COUPLE CAR1 CAR2)
   SEND-MESSAGE (CURRENT-LOC 'X-COUPLE-CARS
                        ('ANYTIME CAR1 SLOT1 CAR2 SLOT2))
   SEND-MESSAGE (SELF 'COOPERATE-WITH-CITY
                        ('ANYTIME CAR1 C1-SLOT CAR2 C2-SLOT))
END
```

The execution of this function sends a message to the manager of the factory at the engineer's current location and a message to the engineer. The message to the manager requests it to couple the cars; the message to the engineer requests it to wait until the manager gives notice that the cars have been coupled.

The manager handles the command sent by the engineer by executing the function, X-COUPLE-CARS.

```
X-COUPLE-CARS (PRODUCTION-CENTER CONDITION CAR1 SLOT1 CAR2 SLOT2)
   PLAN ← PLAN ∪ (CONDITION (DO-COUPLE-CARS CAR1 SLOT1 CAR2 C2-SLOT))
```

As did the engineer, the manager puts the command and the conditions on the list of condition command pairs. In this case the manager puts "((ANYTIME DO-COUPLE-CARS CAR1 SLOT1 CAR2 C2-SLOT))" on its plan list. The next time the manager has nothing to do, and no plans with "ANYTIME" conditions precede this entry in the list, the manager will execute DO-COUPLE-CARS. The DO-COUPLE-CARS function reports that it has begun the coupling and calls the function DO-COUPLE-CARS-1.

```
DO-COUPLE-CARS (CAR1 SLOT1 CAR2 C2-SLOT)
BEGIN
   REPORT (:NOTICE :X-COUPLE-CARS CAR1 CAR2)
   CUR-PLAN ← ((DO-COUPLE-CARS-1 CAR1 SLOT1 CAR2 SLOT2))
END
```

The DO-COUPLE-CARS-1 functions calls the function W-COUPLE-CARS which is an command to the simulator to couple the cars.

```
DO-COUPLE-CARS-1 (CAR1 SLOT1 CAR2 SLOT2)
   W-COUPLE-CARS (CAR1 SLOT1 CAR2 SLOT2)
```

The function W-COUPLE-CARS sets the values of slot1 to car2 and the value of slot2 to car1. To move a car, the simulator looks in the follower slot of the lead car if the cars are moving forward and the driver slot of the lead car if the car is moving backwards and updates the position of the car it finds there. This causes the cars move down the track together as if they were coupled.

6 Adding New Capabilities

The simulator was designed to allow a maximum amount of flexibility. This flexibility allows both additions to the world and additions to the agents. New objects can be added, new actions involving existing objects can be added, new commands to existing agents can be added, and new types of agents can be added.

The object oriented design makes it easy to add new types of objects to the simulator by specializing existing objects. The flexibility of the object oriented design has been extended by providing a macro, DEF-MT-OBJECT, that allow one to define all of the basic functions on objects with one macro call. A macro, DEF-AGENT-ACTION is provided for adding new actions to the simulator. This macro sets the variables so the agent and the world can cooperate to generate the result of the action.

A single macro for adding new commands is not provided, but macros which make the additions of commands relatively easy are provided. Adding new types of agents is more difficult, because agents are the most complicated entities in the simulation. Still, a standard way of communicating with the simulated world and a standard entry point for the agent's code are provided. Thus the primary complexity of adding new agents to the world is the complexity of generating the code for the agents.

This section is sketchy; it is meant to be a guide to the code. It is hoped that it will provide enough guidance that a reasonably skilled programmer could add to the system without having to figure out what every function does.

6.1 Adding Features to the World

To add a new feature to the world, two things need to be defined: the object itself, and the means by which the simulator will display the object. Display of the object is currently implemented using the dynamic windows package in Genera. Plans are underway to change this representation to CLIM as soon as CLIM is added to Genera. Changing to CLIM will make the code much more portable. As it stands now, the simulator will only run on Symbolics machines running Genera 8.0.1. By changing to CLIM, the code will also run under Allegro Common Lisp, Lucid Common Lisp and, other systems as the standard is taken up by other common lisps running CLOS. Due to the state of flux in the presentation, this aspect will not be expanded on further.

To add a new object, the macro DEFCLASS-X should be used. DEFCLASS-X also creates a TYPE-P function and MAKE-TYPE function, like DEFSTUCT does. Any of the classes already defined can be used as superclasses.

The macro DEF-MT-OBJECT allows one to generate all of the functions shown in figure 13 with one macro call. This macro takes seven arguments: TYPENAME, VARNAME, WHERETYPE, INDIRECT-LOCATION, PARENT, ADD-GEST, and SLOTS. The first argument, TYPENAME, is the name of the object being generated. This will be the same name that was given to the DEFCLASS-X macro. The second parameter, VARNAME, is the name of the variable that contains a list of all objects of that type in the current scenario. The third parameter, WHERETYPE, determines the manner in which the object will be placed. If the object is stationary (eg. track of cities), this parameter should have the value "DW:NO-TYPE". If the object being defined is a car, it will have the value "(OR CAR PLACE)". The value of this parameter determines the type of the object at which the new object can be located. A city can be located anywhere, but a car must be located either at a place, or connected to another car. The third parameter, INDIRECT-LOCATION, tells whether the car should be added at a place or connected to a car. If the value of this parameter is T, the car will be added connected to another car, if it is nil, it will be placed at the location entered with the function being defined is called. The fifth parameter, PARENT, allows

one to enter the name of a macro calling this macro for debugging purposes. The sixth parameter, ADD-GEST, is the mouse gesture that adds an object of this type of the scenario. Finally, the seventh parameter, SLOTS, indicates the slots of the object to be filled if the user wants to edit the defaults of the object when adding one to the scenario.

The following is a macro that was written to define stationary objects such as cities.

```
(DEFMACRO DEF-STATIONARY-OBJECT (TYPENAME VARNAME GESTURE &REST SLOTS)
  '(DEF-MT-OBJECT ,TYPENAME
(,VARNAME DW:NO-TYPE NIL DEF-STATIONARY-OBJECT ,GESTURE
          ,@(IF (SUBTYPEP TYPENAME 'AGENT)
                '((TOP-LEVEL-NAME '((MEMBER ,*VALID-TOP-LEVEL-NAMES*))
                   :PROMPT "Top Level Name?"
                   :DEFAULT 'DEFAULT)))
                             ,@SLOTS)))
```

This macro calls DEF-MT-OBJECT with the parameters TYPENAME, VARNAME, GESTURE and SLOTS passed to DEF-STATIONARY-OBJECT. This macro would in turn be called as follows.

```
(DEF-STATIONARY-OBJECT CONTROL-CENTER AGENT-LIST :META-SHIFT-LEFT
  (RAW-STORAGE 'NUMBER :PROMPT "Units of raw storage (tons)"
               :DEFAULT 0)
  (FINISHED-STORAGE 'NUMBER :PROMPT "Units of finished storage (tons)"
                    :DEFAULT 0)
  (RAW-ON-HAND 'LIST
               :PROMPT "Alist of raw material and amount on hand"
               :DEFAULT NIL)
  (FINISHED-ON-HAND 'LIST
        :PROMPT "Alist of (stored) finished products and amount on hand"
        :DEFAULT NIL))
```

This macro defines a control center. It will be kept on a list with all of the other agents, and will be generated by a <meta>-<shift>-<left> mouse gesture. The slots to be filled are RAW-STORAGE, FINISHED-STORAGE, RAW-ON-HAND and FINISHED-ON-HAND all of which default to 0.

The macro DEF-AGENT-ACTION makes easier adding actions the agents can perform on either new or existing objects. DEF-AGENT-ACTION takes seven arguments: ACTION-NAME, HANDLER-ARGLIST, AGENT-TYPE, ACTION-TYPE, FILTER-TYPE, COST, and HANDLER. The first argument, ACTION-NAME, is the name of the new action to be added. This will be the symbol the agent sends to the simulator when it wants to do the new action. The second parameter, HANDLER-ARGLIST is a list of pairs consisting of the parameters this new action will take and the type of the parameter. The next four arguments, AGENT-TYPE, ACTION-TYPE, FILTER-TYPE, and COST, are formed into a list. AGENT-TYPE is the type of the agent that will use this action. ACTION-TYPE is the type of action: perception or action. The value of FILTER-TYPE will be associated with any display the action causes. By setting the filters, agents, and the display, can ignore all inputs from the system of a particular type. The last element of this list of arguments, COST, contains the cost in ticks of the action. The cost may be a function that is computed when the action is attempted. A cost function is supplied when the cost depends on some parameter of the action. For example, it is more expensive to inventory five cars than it is to inventory one. After the parameters have been specified, the body of the action is specified. The body is a function that is called in the simulator that may change the state of the simulator to effect the action.

The following is the call to DEF-AGENT-ACTION that defines the action COUPLE-CARS for a city.

```
(DEF-AGENT-ACTION W-COUPLE-CARS
  ((CAR1-NAME STRING) (C1-SLOT (MEMBER 'DRIVER 'FOLLOWER))
                      (CAR2-NAME STRING) (C2-SLOT (MEMBER 'DRIVER 'FOLLOWER)))
                     (CITY ACTION NIL *CITY-COUPLE-COST*) ...)
```

In this instance the action name is W-COUPLE-CARS. The parameters to the action are two car-names, which are strings, and two slots, whose names are either driver or follower. Any city can attempt the action; it is an action rather than a perception; the filter type is nil, and the cost is a constant *CITY-COUPLE-COST*.

The effect of a call to this macro is to update four global variables, and to define a method on the type of agent able to attempt this type of action that performs the action. The macro updates: *AGENT-ACTION-QUANTA*, an alist of agent actions and numbers of quanta to process them; *ACTION-HANDLER-ALIST*, an alist of agent actions and functions to handle them; *ACTION-HANDLER-TYPELIST*, an alist of agent actions and types for the arguments of the function(s) for handling them; and *ACTION-TYPE-ALIST*, an alist of actions and types of action (actions are done before the tick, perception after).

6.2 Adding Features to the Agents

The generic function CHECK-CONDITION-P implements the types of conditions the agents can check for. Each type of agent has a different method for implementing their perceptions but most of the conditions the agents can check for are implemented by the method for any agent. This more general set of conditions is checked if none of the more specific methods works.

The macro DEF-AGENCY-COMMAND makes adding commands for agents easier. This macro takes three arguments and the body of the command to be added. The first argument is the name of the command to be added. The second argument is the argument list for the function which will be called when the agent handles the command. The third argument is the type of agency for which this is a new command. For example, an engineer's COUPLE-CARS command is defined as follows.

```
(DEF-AGENCY-COMMAND COUPLE-CARS
                    ((CONDITION PE-CONDITION)
                     (CAR1-NAME STRING)
                     (CAR2-NAME STRING))
                    ENGINE-AGENCY ...)
```

The first item in the parameter list is the name of the command. This particular command takes three parameters, the condition under which it is to be executed, and the two cars to be coupled. The command pushes the condition and the command onto the plan list after acknowledging receipt of the command.

The new commands and conditions added to the agent will be processed according to the planning paradigm outlined above. That is, when the agent gets a command, it will place that command on a list of things to do. When it has nothing to do it will check this list for the first item whose conditions are met. It will then place the actions that facilitate the command on the CUR-PLAN variable and begin executing then one by one. The next subsection outlines how to modify an agent's planning paradigm.

6.3 Adding a New Planning Paradigm

New agents can be added at different levels of detail. If a new agent is to be added to the scenario using the paradigm described here, all of the functions and methods defined above can be used. In

general, however, it is expected that different planning paradigms will be used. In this case the agents will have to be restructured. Because the interaction between the simulator and the agents is through world request and notifications, adding radically different agents should not be difficult.

Each agent runs in its own process requiring a certain amount of complexity to the generation of new agent. Each new agent must be able to handle requests from the simulator for new actions to perform, and must be able to handle perceptions the simulator will send it. The system takes care of the inter-process communication, however, so the user need not deal with this. Moreover, the functions that add objects with agents associated with them automatically associate the appropriate agent with the appropriate object.

To add a new reasoning module, first add a new top level name in the list of valid top level names which is associated with the parameter *VALID-TOP-LEVEL-NAMES*, then write a new top level loop for that top level name. This loop must initialize everything in the agent's knowledge base then call TOP-LEVEL-LOOP. The function TOP-LEVEL-LOOP gets a goal from the list of goals the agent is to perform, then calls HANDLE-GOAL with that item. The agents currently implemented do all of their reasoning in HANDLE-GOAL: a different reasoner can be added by changing this function.

When a new object that has an agent associated with it is added to a simulation, the functions generated by the macro DEF-MT-OBJECT automatically associated an agent with any object that has an agent as a subtype. It does this by pushing:

```
(SETF (AGENT-PROCESS (NAME-TO-AGENT <agent name>)))
(AGENT-BUILDER (NAME-TO-AGENT <agent name>))
```

onto the list of things to initialize whenever the simulator process restarts. This list is stored in the state variable "INITS" associated with the program frame for the simulator. Each time the simulator is restarted, the forms in the initialization list are evaluated, so the appropriate agent type will be inserted into the AGENT-PROCESS slot on the AGENT object. Once this is done the AGENT-BUILDER function is called on this object. This function builds a process appropriate to the agent. The function "AGENT-BUILDER" also associates the symbol <top level type>-AGENT-TOP-LEVEL with the INITIAL-FUNCTION slot on the process. Therefore, every time the simulator calls the process in which the agent resides, this function will be called. This name is the one added to *VALID-TOP-LEVEL-NAMES* earlier.

Though any function could be used as the agent's top level function, the user will probably want to rely on the functions already available in the system. To do this, the top level function should call COMMON-AGENT-LOOP as its last function. The COMMON-AGENT-LOOP is defined as,

```
(DEFUN COMMON-AGENT-LOOP ()
  (WITH-SLOTS (GOAL CUR-PLAN) *AGENCY*
    (LOOP (COND
            (CUR-PLAN
             (HANDLE-GOAL (SETF GOAL (POP CUR-PLAN))))
            (T
             (CHOOSE-EXECUTABLE-PLAN)
             (HANDLE-GOAL GOAL))))))
```

This function looks in CUR-PLAN to see if the agent is already working on something. If it is, it continues working on the list of activities it has started. If it is not working on something, it chooses an executable plan by looking in the list of commands it has seen to see if the conditions for any of the plans has been met. If it has, it sets CUR-PLAN to be the list of actions associated with the command, and begins working on the first one.

The agents' reasoning is implemented by the function HANDLE-GOAL. The current agents reasoning consists of a lisp function for each of the goals the agent can handle so this function is defined to be:

```
(DEFUN HANDLE-GOAL (GOAL)
  (APPLY (CAR GOAL) (CDR GOAL)))
```

That is, a lisp function with the same name as the head of the goal is applied to the goal's parameters.

To add a different reasoner, one need only modify HANDLE-GOAL. For example, a more sophisticated reasoner might use its knowledge of the world to reason about ways in which the goal could be achieved. Suppose such a reasoner was called PLAN. Such a planner might take the goal, apply its knowledge to the goal, and return a list of actions the agent effect directly by making a world request. To add such a planner HANDLE-GOAL might appear as follows.

```
(DEFUN HANDLE-GOAL (GOAL)
  (DOLIST
    (ACTION (PLAN GOAL))
    (WORLD-REQUEST ACTION)))
```

Of course, writing an effective "PLAN" function will be difficult.

7 Conclusion

The TRAINS-90 simulation provides a basis for experimentation with a wide range of issues in planning and natural language. The scenario is rich enough to allow conversations with complex structure but is constrained enough to allow these complex conversations without encoding vast amounts of world knowledge. Moreover, the strict separation of the simulation from the system, the TRAINS-90 simulation provides a grounding for the conversations that is unavailable in purely symbolic manipulation.

The separation of the simulation from the system also provides challenges to planners. This separation means that the planner must face the problem of incorrect and insufficient knowledge and must deal with the trade-off between gathering information through perception and inferring information based on a possibly incorrect basis. Moreover, because the simulator runs in its own time frame, issues of temporal reasoning arise naturally. All reasoning takes place under real-time pressure as the simulation changes continuously. The agents must also be able to deal with unexpected events, accuracy failures, etc. and must decide when to report their failures to the planner.

Finally, the flexibility of the design of the system provides a basis for further experimentation in other domains. The ability to add new types of agents allows experimentation in communication between agents of different types. The ability to change the simulated world provides a basis for extending the world to allow experimentation in planning for system not possible in a world of only trains and factories.

A Details of the Simulation of the First Example

Only an overview of the operation of the simulator was presented in the first section. The following shows in greater detail the operation of the simulator as the system's commands are executed by the simulated agents, and the agents' commands are executed by the simulated world. The first subsection shows the operation of the engineer, and the second subsection shows the operation of the managers.

A selection of the running trace of the agents' outputs is shown in the text. Each element of this trace contains three fields:

```
<time> : <connection> <message>.
```

The **<time>** field represents the number of seconds that have elapsed since the start of the simulation. The **<connection>** field shows the communicating parties, an arrow separating the sender from the receiver. Messages from the system to an agent will be of the form **System-><agent>**; whereas messages from the agents to the world will be of the form **<agent>->World**. The third field is the message that was sent. Messages from the system to an agent will contain commands and conditions. Messages from the agent to the world will contain requests for action. Messages from the world to the agents will be a list of the agents' current perceptions. Messages from the agents back to the system represent the agents' reports.

A.1 Engineer's Actions

When engine E3 first starts, it sends a message to the system that a it is restarting. It does this so that if it is rebooted in the middle of executing one of the system's plans, the system will know that the engine has suddenly forgotten everything it ever knew. The following notice will appear in the appropriate window:

```
0: E3->System ((:notice :restart-in-progress))
```

In the simulated first second after notifying the system that it has just woken up, it reads the map to find out the location of the fixed objects in world such as factories and cities, listens, and checks the radio. If it has received instructions from the executor, those messages will come over the radio. If it does get these messages, it acknowledges their receipt. In this example, The map information the engineer gets tells it that there are no cities without factories, there is one production center called "I", two factories called "G" and "B", and one control center called "X". It does not hear anything outside, but there are messages on the radio. The messages on the radio are the plans the system has sent it.

```
1: E3->World (READ-LOCATION-LIST)
1: World->E3
 ((:see map-info
  (:location-map
    city nil
    production-center (("i" 628 188))
    factory (("g" 629 340) ("b" 624 49))
    control-center (("x" 816 97))
    fork nil
    triple nil
    cross nil
```

```
    clover nil
    decoupler nil))
 (:hear dull :quiet)
 (:radio radio
  (goto
   (and (not (empty "b1"))
        (at "i")
        (on-train "e3" "b1")
        (at "i"))
   "b"))
 (:radio radio (couple-cars (at "i") "e3" "b1")))
1: System->E3 (GOTO
               (AND (NOT (EMPTY "B1"))
                    (AT "I")
                    (ON-TRAIN "E3" "B1")
                    (AT "I"))
               "B")
1: E3->System ((:ack :goto <<location b >>))
1: System->E3 (COUPLE-CARS (AT "I") "E3" "B1")
1: E3->System ((:ack :couple-cars "e3" "b1"))
```

In the second second, it reads the connections between the fixed objects, and finds that there is track between "I" and "B" and between "G" and "I" and that "X" controls both of them.

```
2: E3->World (READ-CONNECTION-LIST)
2: World->E
 ((:see map-info
  (:connection-map ("seg-2" "i" "b" 1381 "x" 151)
                   ("seg-1" "g" "i" 1511 "x" 151)))
 (:hear dull :quiet))
```

In the third second it checks to see which control center has control of the switches and finds that since there are no switches no one has control of them. Again it hears nothing outside the engine.

```
3: E3->World (READ-CONTROL-ASSIGNMENTS)
3: World->E3
 ((:see map-info
  (:control-assignments fork nil triple nil clover nil))
 (:hear dull :quiet))
```

After gathering all of this information, the engineer checks the cars attached to it engine. This operation takes a while; it is not until the 141st second that the operation is complete. When the operation is complete, both the operation requested and the results are displayed.

The engineer continues to collect information until the next interesting sequence of events occurs around second 375. At this point the engineer gets a command to go to City I immediately. The engineer acknowledges the command and continues with its work.

```
375: System->E3 (GOTO NIL <<LOCATION I >>)
375: E3->System ((:ack :goto <<location i >>))
```

At second 381, the engineer decides that everything is in order for the trip, and notifies the system that it is going to start.

```
381: E3->System ((:notice :goto <<location i >>))
```

The first thing the engineer does is call the control center for the segment of track it will need to use.

```
386: E3->World
 (RADIO-REQUEST "X" (REQUEST-PERMISSION "E3" ("Seg-1") "G" "I"))
```

The engineer waits until it gets permission at second 453.

```
453: System->E3 (PERMISSION-GRANTED "X" ("Seg-1") "G" "I")
```

Then starts the train towards city "I" by seeing the throttle to 37/100 of its maximum.

```
454: E3->World (SET-THROTTLE 37/100)
```

As the engineer moves down the track it continually updates its information about its position and speed, and occasionally checks its fuel. The engineer controls its speed by changing the throttle until it gets near the city. When it gets near the city it first reduces the throttle causing backwards acceleration.

```
804: E3->World (SET-THROTTLE 1/50)
804: World->E3
 ((:hear dull :engine-noise)
  (:feel dull :backward-acceleration)
  (:feel dull :throttle-set))
```

As it begins to slow it also sets the brake.

```
806: E3->World (SET-BRAKE 1/50 "E3")
806: World->E3
 ((:hear dull :engine-noise)
  (:feel dull :backward-acceleration)
  (:feel dull :brake-set))
```

Finally, it begins to run the engine in reverse.

```
808: E3->World (SET-THROTTLE -3/100)
808: World->E3
 ((:hear dull :engine-noise)
  (:feel dull :backward-acceleration)
  (:feel dull :throttle-set))
```

After about fifteen minutes, the train is in the station.

```
886: E3->World (LOOK)
886: World->E3
 ((:see map-info ("i" :type city :at 0))
  (:hear dull :quiet)
  (:feel dull :backward-acceleration))
```

At this point, the display will appear as in figure 5. The engineer checks its fuel, and, because it has been traveling at a high speed, discovers that its fuel is low. It does nothing during the considerable time it takes to load fuel. Once the fuel has been loaded, the engineer notices that it is now in City I and should therefore try to couple with boxcar "B1". To do so it sends a radio request to the manager of City I. The manager arranges for a switch engine to do the actual coupling.

```
2420: E3->World (RADIO-REQUEST "I"
                 (X-COUPLE-CARS
                  (AND (IDLE "B2")
                       (IDLE "B1"))
                  "B2" FOLLOWER "B1" DRIVER))
```

To make sure that it does not try to pull out of the station until the cars are coupled, it sends messages to itself to cooperate with the city until the cars are coupled.

```
2423: E3->World (COOPERATE-WITH-CITY (W-COUPLE-CARS "B2" FOLLOWER "B1" DRIVER))
2783: E3->World (CONTINUE-COOPERATION)
```

The engineer notices when the cars are coupled, notifies the system that it is going to start executing its second command, requests permission to use the next section of track, and sets the throttle again. Once the cars are coupled, the display appears as in figure 6.

```
3478: E3->World (CHECK-CARGO "B1")
3478: World->E3
 ((:see updates
  (:cargo "b1" ((oranges . 1))))
  (:hear dull :quiet))
3485: E3->System ((:notice :goto <<location b >>))
3490: E3->World
 (RADIO-REQUEST "X" (REQUEST-PERMISSION "E3" ("Seg-2") "I" "B"))
3520: System->E3 (PERMISSION-GRANTED "X" ("Seg-2") "I" "B")
3521: E3->World (SET-THROTTLE 43/100)
3521: World->E3
 ((:hear dull :engine-noise)
  (:feel dull :forward-acceleration)
  (:feel dull :throttle-set))
```

About six minutes later, the engineer reaches City B. It again loads fuel. This time it takes more than thirty minutes before the fuel is loaded. The first thing it notices after the fuel is loaded is a request from the factory manager at City B to unload the car. The engineer then unloads the car for a little more than fifteen minutes.

```
6075: E3->World (LOAD-FUEL)
6075: World->E3
 ((:hear updates :gurgle)
  (:radio radio (unload-car nil "b1" oranges 1)) (:hear dull :quiet))
6080: System->E3 (UNLOAD-CAR NIL "B1" ORANGES 1)
6080: E3->System ((:ack :unload-car "b1" oranges 1))
6080: E3->System ((:completed :goto <<location b >>))
6080: E3->System ((:notice :unload-car "b1" oranges 1))
6146: E3->World (UNLOAD-CAR "B1" ORANGES 1)
7192: E3->World (FINISH-UNLOAD-CAR "B1" ORANGES 1)
```

At this point the engineer has completed all of the tasks assigned to it.

A.2 Managers' Activities

Though there are more cities in the scenario than engines, the tasks the managers of these cities perform are simpler than the engineer's. The commands to the factories, production centers and control centers, are considered to be commands to a manager who in turn has workers simulated by the world simulator do the actual actions. Therefore, the factory managers are able to accomplish more complex tasks with a single world request.

As does the engineer, the factories first tasks are notifying the system that they are restarting and reading information about the current scenario. In addition, the production center at City I gets the command to start loading boxcar "B1" with oranges immediately. The factory at City B gets commands to unload the same car when it arrives, and to start producing oranges juice when it has the raw materials to do so. Since B1 will contain oranges when it arrives at City B, the factory will start production as soon as B1 gets there. As does the engine, the factories read the connections and control assignments for the routes between cities.

```
0: X->System ((:notice :restart-in-progress))
0: B->System ((:notice :restart-in-progress))
0: G->System ((:notice :restart-in-progress))
0: I->System ((:notice :restart-in-progress))
1: I->World (READ-LOCATION-LIST)
1: World->I
 ((:see map-info
  (:location-map city nil production-center
  (("i" 628 188)) factory
  (("g" 629 340) ("b" 624 49)) control-center
  (("x" 816 97))
  fork nil triple nil cross nil clover nil decoupler nil))
 (:radio radio (load-car-with-sync nil "b1" oranges 1)))
1: System->I (LOAD-CAR-WITH-SYNC NIL "B1" ORANGES 1)
1: I->System ((:ack :load-car-with-sync "b1" oranges 1))
1: G->World (READ-LOCATION-LIST)
...
1: B->World (READ-LOCATION-LIST)
...
1: System->B (START-PRODUCTION (NOT (EMPTY RAW)))
1: X->World (READ-LOCATION-LIST)
...
1: B->System ((:ack :start-production))
1: System->B (UNLOAD-CAR-WITH-SYNC
              (AND (NOT (EMPTY "B1"))
                   (HERE "B1"))
              "B1" ORANGES 1)
1: B->System ((:ack :unload-car-with-sync "b1" oranges 1))
```

City X, the control center, also reads a list of signals on each of the segments. The manager of City X has the ability to control the state of the signals letting trains know that the track ahead is clear. At step 427, City X gets a request from engine E3 to use segment-1. To satisfy this request, the manager of the control center checks that the track requested is clear, sets the signals to indicate the track is clear to the engineer, then gives the engineer permission to use the track. As cars pass the signals, the control center manager gets messages from the signals. These messages first appear in step 512.

```
427: World->X
((:see dull :nothing)
 (:radio radio (request-permission "e3" ("seg-1") "g" "i")))
427: System->X (REQUEST-PERMISSION "E3" ("Seg-1") "G" "I")
429: X->World (SET-SIGNAL "SIG1954" :UP)
429: World->X ((:see dull (:signal-ack-set "sig1954" :up)))
431: X->World (SET-SIGNAL "SIG1953" :UP)
431: World->X ((:see dull (:signal-ack-set "sig1953" :up)))
...
452: X->World (RADIO-REQUEST "E3" (PERMISSION-GRANTED "X" ("Seg-1") "G" "I"))
...
512: World->X
((:see dull :nothing)
 (:radio nil (here-i-am "b2" ("sig1954" 0)))
 (:radio nil (here-i-am "e3" ("sig1954" 0))))
```

City I checks its inventory, checks the cars in its freight yard, then, at step 907, notices that it can start loading boxcar B1 with the oranges as requested by the system. At step 967, it starts loading them. At step 2227, the loading is complete; it has taken about twenty minutes for the manager's workers to load the boxcar.

```
907: I->System ((:notice :load-car-with-sync "b1" oranges 1))
...
967: I->World (LOAD-CAR "B1" ORANGES 1)
...
2227: I->World (CHECK-CARGO "B1")
2227: World->I
((:see updates (:cargo "b1" ((oranges . 1))))
 (:see updates (:load-completed "b1" oranges)))
```

In the mean time, engine E3 has arrived at the city and requests the manager to couple the boxcar to the train it is pulling. City I gets the radio message, acknowledges it, and orders the switch engines to couple the cars. Two minutes later, the car is coupled to the train.

```
2662: World->I ((:see updates (:raw-inventory (oranges . 199)))
 (:radio radio
  (x-couple-cars
   (and (idle "b2")
        (idle "b1"))
   "b2" follower "b1" driver)))
2662: System->I (X-COUPLE-CARS
                 (AND (IDLE "B2")
                      (IDLE "B1"))
                 "B2" FOLLOWER "B1" DRIVER)
2662: I->System ((:ack :x-couple-cars "b2" "b1"))
2662: I->System ((:notice :x-couple-cars "b2" "b1"))
...
2782: I->World (W-COUPLE-CARS "B2" FOLLOWER "B1" DRIVER)
2782: World->I ((:see updates (:coupled "b2" "b1")))
```

Engine E3 requests permission to use the next segment of track as before, permission is granted, and the train leaves for the orange juice factory at City B.

City B notices that engine E3 has arrives at step 4038. The manager at City B checks the cargo of both cars, then, having gotten these reports, it recognizes that the conditions for unloading the car have been met, so it notifies E3 that it is going to unload the oranges. A few minutes later the oranges are unloaded,

```
4038: B->World (CHECK-ON-HAND RAW)
4038: World->B
((:see updates (:raw-inventory))
 (:radio radio (here-i-am "e3" ("b" 0))))
4038: System->B (HERE-I-AM "E3" ("B" 0))
...
4593: B->World (CHECK-CARGO "B1")
4593: World->B ((:see updates (:cargo "b1" ((oranges . 1)))))
...
4893: B->World (CHECK-CARGO "B2")
4893: World->B ((:see updates (:cargo "b2" nil)))
4893: B->System ((:notice :unload-car-with-sync "b1" oranges 1))
4898: B->World (RADIO-REQUEST "E3" (UNLOAD-CAR NIL "B1" ORANGES 1))
...
5018: B->World (UNLOAD-CAR "B1" ORANGES 1)
```

It takes the manager's workers a while to tell the manager that the oranges are unloaded, so it is more than forty minutes before the manager realizes that the conditions are right to start making the orange juice. Once it realizes that it has the oranges, it starts production. Finally, about two and a quarter hours after the system sent the commands to the various agents, the orange juice is made.

```
7493: B->World (CHECK-ON-HAND RAW)
7493: World->B ((:see updates (:raw-inventory (oranges . 1))))
7493: B->System ((:notice :start-production))
...
7553: B->World (SET-SPEED 1/100)
7553: World->B
((:hear dull :processing)
 (:see updates (:speed 1/100)))
...
7793: B->World (SET-SPEED 0)
...
7913: B->System ((:notice :production-finished))
```

B Running the Simulator on the Symbolics 36xx

B.1 Starting the simulator

To use the TRAINS-90 simulator the simulator system must be loaded. The names of the TRAINS-90 simulator is Moving Trains, or MT for short. To load the system simply type "Load System MT" at the prompt on the console. Once the system has been loaded, the simulator can be accessed by typing <Select> <Symbol>-= (ie. Hit the <Select> key once, then hold down the <Symbol> key while hitting the "=" key). The simulator screen will appear.

B.2 Loading the Scenario

To load a scenario click left on the "Load World" option on the edit mode menu. A prompt for the world to load will appear. At this point, the name of the file which contains the saved world should be entered. The world shown in figure 3 is saved in the file "mt:mt;small-layout.lisp". When <Return> is hit, the world will be loaded. Because the system is implemented in CLOS, loading the scenario the first time may take a while as the system has to compile all of the CLOS objects in the scenario.

To change to simulate mode click left on "Change mode simulate" in the upper left hand corner of the screen. The screen will change to appear as in figure 4.

To start the example, Engine E3 will have to be given instruction to go to City I. To do this click right on the icon for Engine E3. A menu of command that can be sent to the engine will appear in the "Moving trains command" window. Select the command goto from those offered by clicking left on that option. The program then asks for a condition and a location. No condition need be entered as it defaults to 'ANYTIME. Enter a location by clicking left on "*a location*" then clicking left on the icon for City I. This will enter the correct location parameter. One the location parameter has been entered either click left on "<end> uses these values" or hit the <end> key. Once the engine is moving towards City I the conversation can begin.

B.3 Running the Example

There are two ways of running the system: calling the system expressly from a lisp listener, or calling the system via TCP through the nl-server. To call the system directly from the lisp listener, simply type "top-loop-test". If the system is called from a lisp listener, the user will be prompted for input by "==>". To see the plan being executed, change to the simulator by typing <Select> <Symbol>-=.

To call the system from the simulator, first change to simulate mode. Then click right on the command "Call NL" in the list of commands towards the middle of the screen. This calls the appropriate server and sets up the TCP connection between the simulator and the system. This connection need only be set up once. Once the simulator and the system are connected, sentences can be sent to the system by clicking right on the "Talk to System" command in the command menu. The simulator is currently set to provide the example conversation as defaults. To send the defaults simply hit return. To sent anything else, type the alternative sentence and hit return.

If the system is run from the simulator, it runs in a background process set up by the server process. When the system produces output, it will generate a default background typeout stream and notify the user that it wants to type out by sending a notification to the console of the machine on which it is running. Hitting <Function>-0-S will expose a window in which the system can present its output.

Once the conversation has completed, the plan has been generated, and the agents have received their instructions, the simulator can be started by clicking left on the Tick option from the menu towards the middle of the simulator screen. Clicking left on this option will simulate the passing of six minutes. To save time, the simulator display is updated only when the commands are completed. Generally, sex minute intervals are a good compromise between the cost of redisplay and the benefit of seeing the scenario change. The passing of different amounts of time can be simulated by clicking right on Tick. The simulator can be moved forward in one second increments by clicking left on micro-tick. The example takes about two hour in simulated time, and about fifteen minutes real time on a 36xx.

B.4 Location of Files

The files containing the code for the simulation are listed in appendix C. In general, those files with "mt" in the title, deal with the display of the simulator; those files with "agency" in the title deal with agents; and those files with "sim" in the title deal with physics. Defsystem.lisp defines the way the files are compiled and loaded.[2]

C TRAINS-90 Simulator Files

agency-constants.lisp contains the data structure for generic agents, and parameters affecting their reasoning.

agency.lisp contains the functions from generic agents. The method for CHECK-CONDITION-P for all agents is in this file. HANDLE-GOAL, AGENCY-REPORT, AGENCY-BUILDER, and WORLD-REQUEST are also in this file.

agent-action.lisp contains the primitive actions any agent can perform. This file contains a list of DEF-AGENT-ACTION macro calls.

city-agency.lisp contains the functions and methods for managers that are not specific to factories, production centers or control centers.

control-agency.lisp contains the functions and methods specific to control centers.

defsystem.lisp defines the mt system (like a make file).

dispatcher.lisp contains the functions for communication between agents and between the system and agents.

engine-agency.lisp contains the functions for engineers.

factory-agency.lisp contains the functions for factory managers.

layout.lisp contains a large layout.

mt-defs.lisp contains the data structure definitions for common to the entire system. In particular, this file contains the CLOS classes for the main object in the simulator.

mt-images.lisp contains the icons and images for the simulator.

mt-objects.lisp contains the macros and functions for defining new objects. That is, this file contains macros which define all of the instances and connections between instances to generate a simulator object. This file contains the definition for DEF-MT-OBJECT and the macros that call it.

mt-windows.lisp contains the functions that generate the windows.

productionc-agency.lisp contains the functions for the production centers.

sim-constants.lisp contains the constants for the simulated physics.

sim.lisp contains the functions for the simulated physics. In particular, this file contains the definition of TICK, ONE-STEP, NEXT-STEP, and UPDATE-TRAIN-POSITION.

[2] Locally, all of these files are found in the directory /s5/mt. On the lisp machines, if the translations for moving trains have been loaded, the files also reside in mt:mt;.

small-layout-saved-route.lisp contains the route list in a form the agents can use. This file is generated by the simulator the first time it is started.

small-layout.lisp contains the scenario for the example.

D Conditions

'anytime Do the command as soon as possible.

(at location) Do the command when location is reached. (Applies only to engines.)

(here object) Do the command when a object arrives. (Applies only to cities.)

(coupled car1 car2) Do the command when car1 is connected to car2.

(on-train car) Do the command when car is attached to the train being pulled. (applies only to engines.)

(empty car) Do the command the car is empty. (The car must be in the city to which the command is addressed or on the train to which the command is addressed.)

(and ...) Do the command when the conjunction of conditions is true.

(or ...) Do the command when the disjunction of conditions is true.

(not perception) Do the command when the perception is not true.

(empty) Do the command when there are no raw materials.

(needs-filled) Do the command when the silos are empty. (Applies to production centers.)

(idle tool) Do the command when the tool is not being used.

E Commands

E.1 General Commands

Restart Restart processing.

load-car Load goods into a car.

unload-car Unload goods from a car.

E.2 City Commands

Keep-silos-full Produce more raw materials.

load-car-with-sync Contact the engineer who is in control of the car and make sure it does not leave until the car is loaded.

unload-car-with-sync Contact the engineer who is in control of the car and make sure it does not leave until the car is unloaded.

x-couple-cars Couple the cars for the engineer who made the request.

x-uncouple-cars Uncouple the cars for the engineer who made the request.

here-i-am Sent by an engine or signal to tell city where the engine is.

do-start-production Start producing finished product (Applies only to factories.)

E.3 Engine Commands

goto Take the most direct route to the location mentioned.

couple-cars Couple the two cars mentioned.

uncouple-cars Uncouple the two cars mentioned.

where-are-you Reply to the agent sending the request with the best estimate of current location.

permission-granted Note that the agent has sole access to the current track segment. This command allows the engineer to travel much more quickly than would otherwise be reasonable.

E.4 Control Center Commands

request-permission Sent by an engine to request permission to use a controlled route. Returns a list of the permitted access, the engine will have to re-request anything not (yet) given permission for.

request-resumption Sent by an engine when stopped waiting on a signal. This is "in case" we forgot about it.

F Actions

F.1 Perception Actions

Look causes the agent to make visual scan of the area. For example, an factory might look down the track to see what is coming. This perception is limited by line of sight.

Smell causes the agent to test the air. For example, an engineer whose brakes are failing might smell the burning brakes. This perception is even more local than sight.

Listen causes the agent to make a auditory scan. For example the engineer hears the engine running as long as it has fuel. This command is less local than sight. Agents hear radio messages. message has come in.

F.2 Initialization Actions

Read-location-list reads in a list of the location of the cities and track connections.

Read-equipment-list reads in a list of the equipment at hand at the agents location.

Read-connection-list reads in a list of the segments connecting the cities.

F.3 City Actions

Set-speed sets the speed of production of the items the production center makes. If the production center is a factory this effects the rate at which finished materials are produced and raw materials are consumed.

Check-speed checks the speed of production.

Check-cargo checks the cargo in a car.

Check-on-hand checks the amount of the item mentioned available.

W-couple-cars actually couples cars.

W-uncouple-cars actually uncouples cars.

Load-car puts goods in a car.

Unload-car removes goods from a cargo car.

Inventory-cars checks the cars in the city.

Radio-request causes the agent to check the radio to see if a

F.4 Engine Actions

Cooperate-with-city waits until the action requested of the city is complete.

Continue-cooperation continues waiting until the action requested of the city is complete.

Set-throttle sets the force of the engine. This command indirectly sets the speed of the engine.

Check-throttle checks the amount of power the engine is producing.

Check-velocity checks the speed of the engine.

Check-fuel checks the amount of fuel available.

Set-brake sets the status of the brake.

Check-brake checks the status of the brake.

Check-cargo checks the cargo on a particular car.

Choose-city-direction decides which of the possible exits form the city to take.

Set-switch sets the state of a switch.

Load-fuel loads fuel into the engine.

Load-car sends a request to the city to load a car.

Unload-car sends a request to the city to unload a car.

Inventory-cars sends the engineer out to check what is in each of the cars attached to its engine.

Radio-request causes the agent to check the radio to see if a

G Action Cost Parameters

CAR-TRAVERSAL-COST (ROUND (/ +FEET-PER-TRACK-UNIT+ 3)) Number of ticks it takes for engineer to move to another car.

TRACK-TRAVERSAL-COST (ROUND (/ +FEET-PER-TRACK-UNIT+ 5)) number of ticks it takes for engineer to walk one track unit.

SET-SWITCH-CONSTANT 60 Number of ticks for engineer to disembark, set a switch, and reembark.

LISTEN-COST 0 Cost to do a listen.

SMELL-COST 0 Cost to do a smell.

CITY-DIRECTION-COST 5 Cost to enter and exit a city on an arbitrary track segment (exponential in number of segments).

BRAKE-CHECK-LOCAL 1 Cost to check the brake in the engine.

BRAKE-CHECK-REMOTE 1 Cost to check the brake in a car (assume in the car).

SET-BRAKE-LOCAL 1 Cost to set the brake in the engine.

SET-BRAKE-REMOTE 2 Cost to set the brake in a car (assume in the car).

CHECK-VELOCITY-COST 1 Cost to read the speedometer.

CHECK-THROTTLE-COST 1 Cost to read the throttle.

SET-THROTTLE-COST 1 Cost to set the throttle.

RADIO-WORD-COST .5 Cost to radio a word.

RADIO-START-COST 3 Cost to start a radio message to a recipient

CITY-INVENTORY-COST 120 Cost for a city to find out what cars are there

CARGO-CHECK-COST 180 Cost to check out the contents of a car

PC-LOOK-COST 60 Cost for any pc to see what's coming

ENGINE-LOOK-COST 1 Cost for an engine to look ahead (non-zero to force engineer in cab)

LOAD-FUEL-RATE 2 Gallons/sec for loading fuel.

CHECK-FUEL-COST 1 Assume it's just a dial, unlike our icon

CITY-GOODS-INVENTORY-COST 360 Cost for a city to inventory what's in storage (sep for raw and finished

CHECK-PC-SPEED-COST 15 Cost to check how fast the pc or factory is producing stuff

CITY-START-LOAD-COST 60 Cost for a city to start loading a car (actual time to finish load will be determined by load-rate of car, etc.

CITY-START-UNLOAD-COST 60 Cost for a city to start unloading a car (actual time to finish load will be determined by unload-rate of car, etc.

ENGINE-START-LOAD-COST 20 Cost for an engine to start loading a car (actual time to finish load will be determined by load-rate of car, etc.

ENGINE-START-UNLOAD-COST 20 Cost for an engine to start unloading a car (actual time to finish load will be determined by unload-rate of car, etc.

CITY-COUPLE-COST 120 Cost for a city to effect a car coupling with help from engineer

CITY-UNCOUPLE-COST 300 Cost for a city to effect a car decoupling with help from engineer

PC-SET-SPEED-COST 60 Cost to set the speed of a PC or factory

CC-SET-SIGNAL-COST 2 Cost to update a signal with a new position

CC-SET-SWITCH-COST 5 Cost to update a switch with a new position

H Agency Constants

The following shows the constants the current agents use.

H.1 Approximations used by Agents

Agents do not have access to the parameters used by the simulation. Their information is estimated based on the following parameters.

WALKABLE-DISTANCE (/ 200 +FEET-PER-TRACK-UNIT+) Number of track quanta an engineer feels like walking.

MAX-AGENCY-ACCEL (MPH-TO-TRACKS-PER-TICK 4) Don't try to accelerate faster than this.

MAX-AGENCY-DEACCEL (MPH-TO-TRACKS-PER-TICK 4) Don't try to slow down faster than this.

MIN-AGENCY-DEACCEL (MPH-TO-TRACKS-PER-TICK .5) Don't bother slowing down by this much, unless we are close.

ASSUMED-CONTROL-RESOLUTION 1/100 How good I think my use of the controls are.

ASSUMED-LOOK-DELAY 3 Maximum Number of ticks likely to pass before we look out the window.

ASSUMED-ACTION-DELAY 3 Maximum Number of ticks likely to pass before we can do something.

ASSUMED-PC-ACTION-DELAY 360 Max Number of ticks likely to pass before a PC can do something.

SPEEDING-MAX 1.1 factor of max velocity before we consider ourselves to be seriously 'speeding' and slow down faster.

H.2 Anxiety Levels

Agents need to check the condition of the world at regular intervals, because their access to information about the world is limited. The frequency with which they check is controlled by their anxiety level. Anxiety is calculated as the exponential b^t where b is the parameter described below, and t is the number of seconds (ticks) that have passed since the agent last checked the condition. Agents always check the condition for which they have the highest anxiety level.

ENGINE-CARS-ANXIETY 1.2 Base for the anxiety for engine checking cars.

ENGINE-CARGO-ANXIETY 1.15 Base for anxiety for engine checking cargo

ENGINE-VELOCITY-ANXIETY 2.4 Base for anxiety for engine checking speed.

ENGINE-LOCATION-ANXIETY 3.0 Base for anxiety for engine checking position.

ENGINE-BRAKE-ANXIETY 1.05 Base for anxiety for engine checking brake.

ENGINE-THROTTLE-ANXIETY 1.05 Base for anxiety for engine checking throttle.

ENGINE-FUEL-ANXIETY 1.6 Base for anxiety for engine checking fuel.

PC-CARS-ANXIETY 1.6 Base for anxiety for production center checking cars.

PC-CARGO-ANXIETY 1.5 Base for anxiety for production center checking cargo.

PC-ARRIVALS-ANXIETY 3 Base for anxiety for production center checking on the arrival of cars.

PC-RAW-ON-HAND-ANXIETY 1.45 Base production center's anxiety for checking amount of raw material in silos.

PC-FINISHED-ON-HAND-ANXIETY 1.1 Base for anxiety for production center checking finished materials on hand.

PC-SPEED-ANXIETY 1.05 Base for anxiety for engine production center speed of production.

FACTORY-CARS-ANXIETY 1.6 Base for anxiety for factory checking cargo.

FACTORY-CARGO-ANXIETY 1.5 Base for anxiety for factory checking cargo.

FACTORY-ARRIVALS-ANXIETY 3 Base for anxiety for factory checking the arrival of cars.

FACTORY-RAW-ON-HAND-ANXIETY 1.4 Base for anxiety for factory checking amount of raw material on hand.

FACTORY-FINISHED-ON-HAND-ANXIETY 1.45 Base for factory for engine checking finished materials on hand.

FACTORY-SPEED-ANXIETY 1.05 Base for anxiety for factory checking speed of production.

CONTROL-CARS-ANXIETY 1.1 Base for anxiety for control-center checking cars.

CONTROL-CARGO-ANXIETY 1.05 Base for anxiety for control-center checking cargo.

CONTROL-ARRIVALS-ANXIETY 1.1 Base for anxiety for control center checking on the arrival of cars.

CONTROL-RAW-ON-HAND-ANXIETY 1.1 Base for anxiety for control center checking amount of raw material on hand.

CONTROL-FINISHED-ON-HAND-ANXIETY 1.1 Base for control center's anxiety for checking finished materials on hand.

References

[Allen and Schubert, 1991] James F. Allen and Lenhart K. Schubert, "The TRAINS Project," Computer Science 91-1, University of Rochester, 1991, TRAINS Technical Note.

[Brown *et al.*, 1988] Christopher Brown, Dana H. Ballard, Timothy G. Becker, Roger F. Gans, Nathaniel G. Martin Thomas J. Olson, Robert D. Potter, Raymond D. Rimey, David G. Tilley, and Steven D. Whitehead, "The Rochester Robot," Computer Science 257, University of Rochester, 1988.

[Ferguson, 1991] George Ferguson, "Domain Plan Reasoning in TRAINS-90," Computer Science 91-2, University of Rochester, 1991, TRAINS Technical Note.

[Light, 1991] Marc Light, "Semantic Interpretation in TRAINS-90," Computer Science 91-3, University of Rochester, 1991, TRAINS Technical Note.

[Martin *et al.*, 1989] N. Martin, J. Allen, and C. Brown, "ARMTRAK: A Domain for the Unified Study of Natural Language, Planning, and Active Vision," Technical Report 324, University of Rochester, 1989.

[Schubert, ming] Lenhart K. Schubert, "Language Processing in the TRAINS Project," Computer Science 91-1, University of Rochester, forthcoming, TRAINS Technical Note.

[Traum, 1991] David Traum, "The Discourse Reasoner in TRAINS-90," Computer Science 91-5, University of Rochester, 1991, TRAINS Technical Note.