1998

An Authoring System for Intelligent Tutors for Procedural Skills

Rowe, Neil C.; Galvin, Thomas P.

Monterey, California. Naval Postgraduate School

A revised and improved version of this paper appeared in IEEE Intelligent Systems in 1998.
https://hdl.handle.net/10945/35982

NEIL C. ROWE AND THOMAS P. GALVIN

Computer Science, U.S. Naval Postgraduate School

Monterey CA USA 93943

Abstract -- Successful intelligent tutoring systems can now provide powerful learning environments, but have been hard to transfer to other instructional domains. Conventional authoring systems for tutors, on the other hand, can build a wide range of tutors, but generally do little to tailor instruction to the student. We describe MEBuilder and METUTOR, software tools with the advantages of both. MEBuilder is an authoring tool for procedural-skill tutors that uses object-oriented methods and automatic planning to prototype tutors quickly, and METUTOR is a run-time environment for its tutors. The tutors built offer simulations for "learning by doing" and use an extended form of means-ends analysis to provide intelligent focused tutoring. Using MEBuilder itself requires no programming, only answers to a few questions. It also features library management, inference of menus of relevant teacher options, designation of student exercises, and automatic error and consistency checking on teacher specifications. Experiments demonstrated that subjects could construct tutors faster with MEBuilder than with a conventional authoring system, despite little training. MEBuilder and METUTOR can be valuable tools for training in the specialized procedural skills of increasingly technological societies.

A revised and improved version of this paper appeared in *IEEE Intelligent Systems* in 1998.

II. - The METUTOR Tutoring Environment
A. - Specification of Actions
B. - Problem-Independent Intelligent Tutoring Methods
C. - Interrelated Graphics
D. - Assessment of METUTOR Alone
III. - The MEBuilder Tutor-Construction Shell
A. - Library
B. - Object Definition
C. - Task Graphs
D. - Exercises
E. - Exercise Compiler
F. - Assessment of MEBuilder
IV. - References
V. - Bibliography

I. Introduction

Procedural skills are skills for which the student must supply a sequence of actions to achieve some desirable result. They are important in many areas of traditional school curricula. Technical organizations like ours also require knowledge of wide range of reactive procedures for hypothetical situations and technical procedures for maintenance and repair of equipment.

Because procedural skills require some planning, students can find them difficult. They become especially challenging when there are many action choices and actions have context-dependent effects. Traditional rote learning does not work well then because students must know reasons for preferring one action to another, including in rare situations. Then "learning by doing" in computer simulations is often the best way to learn. A simulation can easily back up or redo actions, patiently test students on large numbers of subtly different problems, or teach skills expensive or dangerous to learn in the real world. Good tutoring of "learning by doing" requires intelligence, necessitating flexible knowledge representation and artificial-intelligence planning methods to analyze student plans and find alternatives. Our idea is to provide teachers with an easy-to-use general-purpose tool for authoring software for such intelligent tutoring of procedures, and let teachers particularize their application with narrowly focused knowledge.

Authoring systems building decision graphs that students navigate with answers to questions [1] have been around for twenty years. Such tools are good for simple procedural skills, but are hard to scale up: a task requiring n distinct actions may require a decision graph of n! nodes to cover all errors, Usually the courseware author just ignores most possible student errors, forcing the student to duplicate a predefined sequence.

Recently tools for authoring flexible and intelligent tutoring systems have appeared. A representative example is PIXIE [2], an expert-system development shell specialized to tutoring applications. PIXIE is rule-based, and permits a variety of domain-representation options, domain-inference rules, and domain-dependent tutoring rules. But PIXIE is for a professional programmer who thoroughly understands the rule-system metaphor, something difficult for college students. Other interesting tools include the tutor development system in [3], for domains whose skills can be modeled by production systems, but only part of the methodology is automated. [4] describes a tool that helps teachers define heuristic-search spaces for students to explore, but without "recommended" actions as in our approach. Both [3] and [4] are primarily for teaching mathematics. [5] describes a shell for "role-performance" skills using a library of "predefined general task structures" to help trainers develop tutors for vocational skills such as those of a cashier. While role-performance skills are our primary interest too, the latter's tutoring strategies sound quite ad hoc.

From the previous discussion and consideration of the needs of adult training, we emphasized equally the following design criteria in MEBuilder and METUTOR: (1) space efficiency; (2) believability of the simulations; (3) intelligent and automated task planning; (4) and ease of exercise construction and modification.

II. The METUTOR Tutoring Environment

METUTOR is the run-time environment for tutors constructed with MEBuilder. Both are both implemented in Quintus Prolog, run on Sun Sparcstations, and are available free. Teachers with programming experience can also fashion METUTOR tutors directly without MEBuilder, or augment an MEBuilder tutor.

METUTOR reflects our experience that procedural skills are often best specified and taught using means-ends analysis [6], a concept of broad application to human qualitative problem-solving, and a concept that even unsophisticated people can understand well. The idea is to recursively subdivide planning problems into three parts: achieving the preconditions of a recommended action, performing the recommended action, and achieving the original objectives after the recommended action. Even when not often used by people to solve a problem, a means-ends solution teaches a student about logical thinking. (Our system implementation, however, uses more powerful planning methods too.)

Fig. 1 shows the student's view of an example METUTOR tutor for leading a firefighting team on a naval ship, a skill important for naval officers to know thoroughly. The

student selects an action from the menu in the lower right and has just selected "wait". The lower left area shows the state of the simulation graphically; the upper right describes this state verbally plus holding messages to the student; and the upper left lists the actions taken so far. Fig. 1 shows a situation in which the fire is raging and giving off smoke, the fire team is equipped and at the fire, the power is turned off (as symbolized on the left), boundaries around the fire have been set, and the fire team has gotten close to the fire. Fig. 2 shows a different situation where the fire is verified to be extinguished but there is water on the floor, oxygen and explosive gases have been tested, and a casualty has just occurred from smoke inhalation because the student erred in calling in the reflash watch team before the area was desmoked.

A. Specification of Actions

Means-ends analysis and METUTOR require recommendation conditions, preconditions, and postconditions for actions. For instance to "extinguish" in our firefighting tutor, the definitions in Prolog form are (extending the program in chapter 11 of [7]):

recommended([out(fire)],extinguish).

precondition(extinguish,

[location(fire),raging(fire),equipped(team),set(boundaries),confronted(fire)]).

The first line says that extinguishing is recommended whenever a goal is to get the fire out. The next two lines say that to extinguish a fire, your location must be the fire, the fire must be raging, the fire team must be equipped, the boundaries of the fire must be set, and you must be facing the fire (conditions true in Fig. 1, for instance). These are Prolog facts, but Prolog rules can be used instead to put calculation and procedural features into METUTOR specifications. Postconditions of actions are defined separately as deleted facts, added facts, and random changes to facts:

deletepostcondition(extinguish, [raging(fire), set(boundaries), confronted(fire),

verified(out(fire)), watched(reflashing), debriefed(team),

tested(gases), tested(oxygen) safe(gases), safe(oxygen), unsafe(gases), unsafe(oxygen)]).

addpostcondition(extinguish, [out(fire),watery,smokey]).

addpostcondition(extinguish, [not(deenergized(fire,area))],

[present(casualty),dead(casualty),present(crater),raging(fire)], `There is a big explosion!').

randchange(extinguish,[],out(fire),raging(fire),0.3,'Fire is still raging.').

The first three lines say that extinguishing a fire removes (but does not deny) any facts about the fire raging, the boundaries being set, the fire being approached, the fire being verified out, anyone watching for reflashing, the team being debriefed, the gases and oxygen being tested, and the gases and oxygen being either safe or unsafe. The next line says the default facts added when a fire is extinguished are that fire is out, things are watery, and things are smokey. The next three lines give context-dependent postconditions: If a student tries to extinguish the fire when the fire area is not deenergized, then instead a dead casualty is present, a crater is in the floor, the fire is still raging, and the student is told "There is a big explosion." The last line says that 30% of the time when a student tries to extinguish a fire, even if they have achieved all the preconditions, the fire will continue raging to add realism.

Setting up an METUTOR tutor is thus easier than with the more general tools of [2], [3], and [4], and just requires writing the above definitions for every relevant action, plus a starting state and goal conditions like these for firefighting:

start_state([location(repair,locker), raging(fire), smoky]).

goal([verified(out(fire)), safe(gases), safe(oxygen), not(equipped(team)), not(smokey),

not(watery), not(watched(reflashing)), not(present(casualty)), not(unreplaced(casualty)),

not(treated(casualty)), not(dead(casualty)), debriefed(team), deenergized(fire,area)]).

Though not used in the above example, variables can be used to generalize specifications; Fig. 3 shows variables (the "?"s) in action names in a tutor for teaching system administrators the basics of response to malicious intrusion on Unix systems.

B. Problem-Independent Intelligent Tutoring Methods

METUTOR's automatic planning extends means-ends reasoning to include caching of solutions to subproblems, caching and automatic backtracking on failure to solve a subproblem, recognition and avoidance of unachievable preconditions and infinite loops, time limits on subproblems, and a variety of debugging information. With this reasoning, the student's action at any point can be compared with the best action then, and an incorrect student action can be analyzed to find how the student chose it. Domain-independent tutoring rules (not mutually exclusive) include:

* If the student's action does not change anything, say so.

* If the original goal is now unsolvable, say so and stop.

* If the student has five times avoided a certain best action, give a hint (as in the "Say, why not" message in the upper right of Fig. 1).

* If the student's action's name is similar to the best action (e.g. "desmoke" instead of "deenergize"), tell them.

* If the student's action only makes sense by misreading the current state (like "safe oxygen" for "unsafe oxygen"), or misinterpreting the graphics, tell them (as in the "Have you confused" in the upper right of Fig. 3).

* If the student's action is unnecessary to solve the problem (like testing gases while a fire is still raging), tell them.

* If the student has returned from a digression, point out where and how they digressed. (A digression is started by an action that, while perhaps eventually needed, does not help when it occurs.) For example, if the student goes from the repair locker to the fire scene, deenergizes the area, then goes back to the repair locker to get their equipment, the first action was a digression from getting equipped. Fig. 2 gives another example with the "Do you see" remark citing the student's unncessary trip to the repair locker.

* If the student picks a recommended action, but one not of the highest priority, compare the justifications for it and the best action in their precondition chains. Point out how the purpose of the best action is better than the student's apparent comparable purpose. Fig. 2 gives an example in the "OK, but a hint" message: The student should make the fire area safe (by desmoking and dewatering) before setting the reflash watch, normally the last action before returning to the repair locker to get debriefed. Fig. 1 gives another example.

* If the student does something that will lead to an obvious failure, don't tutor but let them proceed to that failure, for they will remember that better. A big advantage of simulations is that you can permit dangerous consequences. For instance, there can be an explosion if the student forgets to turn the power off before extinguishing a fire.

These rules permit a wide range of tutoring with domain-independent prescriptions working on the domain-dependent problem definition written by the teacher. They are more precise than the "phenotypes of erroneous actions" of [8]. (Additional rules in METUTOR also check teacher errors like unachievable preconditions.)

C. Interrelated Graphics

Many researchers like [2] and [6] have observed the value of multiple representations in learning, especially visual ones recently. For procedural skills, the simulated state is important, and METUTOR gives an English description at every step. But METUTOR also allows the teacher to establish many-to-many mappings between fact sets in states and bitmap (or perhaps text) sets shown in a display area. For instance, our firefighting tutor has:

bmap(smokey, [location(repair, locker)], smokey1, 8, 1, yellow).

bmap(smokey, [location(fire)], smokey2, 380, 6, yellow).

These refer to two bitmap files, a small picture of smoke billows called "smokey1" and a big picture called "smokey2". (We provide our own utilities to enable teachers to make line drawings, manipulate them with transformations, clip them, shade them, and turn them into bitmaps.) The first line says that the visual representation of "things are smokey", when the team is at the repair locker, is the "smokey1" bitmap drawn in yellow with upper left corner at (8,1). The second line says that if the team is at the fire, the visual representation should be the bigger "smokey2" (to appear to approach the fire) with upper left corner at (380,6).

The lower left of Figs. 1-3 holds bitmaps, including smokey2 at top center in Figs. 1 and 2. In Fig. 3 the graphic objects are intelligently-intended text of Unix directory listings with a few added markings; variables in actions can be instantiated from a menu (as in the upper middle). Note that the teacher does not have to design the rest of the interface, as the lists of possible actions, previous actions, and verbal tutoring are supplied automatically. The teacher can also designate overlay priority for bitmaps, so flames go in front of the background, but people go in front of flames. And the teacher can attach actions to mouse clicks in designated regions of the screen, as for example in Fig. 3 for designating file arguments.

D. Assessment of METUTOR Alone

METUTOR by itself has been used in ten M.S. theses and about 50 class projects in advanced courses. Some representative tutors are summarized in Fig. 4. All but the first were developed by M.S. students. Development time is approximate since significant digressions were required for debugging and improving METUTOR itself.

As for the design criteria in Section I, since METUTOR can exploit the full power of first-order predicate calculus (from Prolog), its logical capabilities are broad. METUTOR also does reasonably well on intelligent and automated task planning because it can use an extended form of means-ends analysis. METUTOR has minimal facilities for defining and managing exercises. But it does greatly simplify skill specification. The firefighting tutor, for instance, has 23 actions and 13,433 possible states. Each of those would require a least a minute to specify text and interconnections with traditional authoring software like that of [1], for a total of 224 man-hours, if the tutor functionality were to be the same as the METUTOR tutor. In general, METUTOR specifications are O(n) in size, n the number of actions, whereas a traditional authoring system of the same functionality would require O(n!) or worse than that if actions must be done more than once. So even if METUTOR specifications are harder to write, METUTOR will much easier than traditional authoring for large n.

METUTOR does well on space efficiency because logical specification of actions gives principles rather than a complete state graph. 13,433 states need at least four bytes each for two pointers, plus at least a byte of pointer to text, for a total of 671,650 bytes, whereas the METUTOR firefighting specifications require 5,499 bytes including the symbol table.

III. The MEBuilder Tutor-Construction Shell

MEBuilder helps teachers construct METUTOR tutors. It automatically converts simpler information obtained from the teacher into the METUTOR action specifications of Section II.A. (It does not affect the tutoring rules of II.B or graphics of II.C, which are straightforward for teachers to exploit.) It functions much like an expert-system shell specialized for procedural expertise. Teachers do predominantly bottom-up design by first defining property sets, then objects, then actions, then task graphs, and then exercises. Fig. 5 shows MEBuilder's organization.

A. Library

The library helps further achieve the first design criterion, space efficiency. Each object, action, task graph, exercise, and compiled exercise gets its own file in a library subdirectory. When the teacher loads, saves, or modifies any MEBuilder entity, it automatically loads, saves, or marks for modification the related entities. For instance, if a new property is added to the oxygen tester, actions involving the tester must be marked for reexamination before their next save, to set appropriate values in conditions of the action. Libraries may be coalesced to build general-purpose libraries of objects and task graphs, to save much development time.

B. Object Definition

MEBuilder's object modeling helps further achieve the second design criterion, believability, by organizing the simulation world realistically. MEBuilder's object modeling shows benefits of object-oriented design described in [9]:

* Generalization Hierarchies. Object properties inherit downward to subtypes. There are two top objects, "prop" and "character". Inheritances may be overridden

* Component/Possession Relationships. Prop objects may have parts, like a wrench having a handle, and character objects may have possessions, like a mechanic owning a wrench. Parts and their subparts automatically map to parts and subparts of subtypes

* State Relationships. Sets of mutually exclusive properties can be associated with the object. For instance, in cooking procedures, the cooking status property of food can be "raw", "warm", "cooked", or "burned". Particular properties can be hidden from the student (like when oxygen is unsafe but the student must test it). Summary facts can simplify state descriptions by representing the co-occurrence of several other facts, like "flashlight is working" meaning "flashlight's case is closed, its top is assembled, its batteries are ok, and its bulb is ok".

* Stimulus/Response Relationships. Actions are also inheritable objects, and each action has an actor and primary prop (like firefighting has a fire team leader and a fire), plus additional props and actors. Action definition requires preconditions, intended effects, and side effects (like extinguishing a fire has a precondition of an equipped fire team, its intended effect is that the fire is out, and a side effect is that the area is flooded with water). Once objects for an action are selected, those conditions must be drawn from only the objects' property values, which permits selection by often-short menus. Students choose actions, but the teacher can also define "daemons" that automatically do actions; daemons either progress through a property set (like with cooking status of a food when you leave it in a hot oven too long), loop (like streetlight colors), or are invoked by an action.

Teachers are led through a structured series of questions from which data structures are created. Fig. 6 shows an example interaction that defines the procedure of frying a hamburger. The teacher defines four objects, then two property sets, then two operations, and finally the task for the student. This example starts with an empty library for clarity, but task definition in general is easier since it can reuse basic definitions.

C. Task Graphs

To further address the third design criterion of intelligent task planning, a valuable alternative representation of a procedural skill is a graph of the sequence of actions, a "task graph" or procedural net [10]. Originally we let teachers draw the graph themselves, but they made many subtle errors in deciding which pairs of actions could be left unordered. So we now automatically infer a preliminary task graph, and let teachers edit it in restricted ways.

So whenever a teacher has finished defining actions, and wants to check if they can be strung together correctly to solve student exercises, they call on the task graph module. For each graph, MEBuilder requests a principal actor, starting state, and list of goal conditions. It then uses the extended means-ends analysis routine of METUTOR to try to find an action sequence from the start to the goal, with the recommendation conditions for the actions assumed to be their intended effects. The resulting sequence is shown to the teacher as visual feedback about the consequences of the action definitions. (If no action sequence can be found, the teacher has a bug; trace information is then provided, summarizing in English paraphrase the key information from what a Prolog trace would show. Unachievable subproblems are specially flagged since these are usually where the bug lies.) The teacher can then edit the object definitions, action definitions, or also the task sequence by reordering or introducing parallelism (making it a graph) where appropriate. Fig. 7 gives an example output task graph found by MEBUILDER, from teacher definition of preflight procedures of an aircraft (due to John Kisor).

Task graphs are more than visual aids, however, because additional constraints on a task can be inferred from them:

* Variables. These provide for greater generality, as with METUTOR alone. For example, a firefighting task can be instantiated for any team leader and any fire.

* Context-Dependent Side Effects. Special postconditions of actions can be designated for particular places in task graphs.

* Random Side Effects. Postconditions of actions can be designated as random with specified probabilities.

* Unordered Actions. Actions that have much flexibility in their place of occurrence (like dewatering before returning to the repair locker) can be tagged as only needing to occur before a particular point.

Teachers can do a range of manipulations on task graphs to better model tasks and all the ways to do them. This includes: (1) finding permutable actions or subprocedures (successive actions whose preconditions and postconditions do not interact, using the ideas of [10]); (2) permuting actions; (3) putting sequential actions in parallel; (4) putting parallel actions in sequence; (5) adding the special features listed above. But teachers are prevented from violating "step dependencies", for instance that if the postconditions of action X are included in the preconditions of action Y, then X must precede Y in the task graph. Currently, disjunctive options in tasks must have separate task graphs.

D. Exercises

To further address the fourth design criterion of ease of exercise construction, teachers can define exercises which differ from task graphs in two important ways:

* Variables in objects, starting states, and goal conditions must be instantiated, to make the exercise concrete for the student.

* An exercise can require more than one task graph. Every object in the exercise must appear in at least one of its task graphs.

Example exercises for novice pilots include:

* Subtasks. One exercise could be the preflight checks and startup, another the taxi procedures and tower communications.

* Increasing Difficulty. One exercise could be the entire task with no emergencies. Another could contain minor failures of equipment. Yet another could contain numerous overlapping random failures to really test the pilot.

* Different Props. Each exercise could use a different type of aircraft with different procedures.

Teachers can also set some special options in exercises, such as blocking all random events in the task graph, overriding some of its probabilities, and setting "agent speed" or how often a character other than the student gets a turn per each student turn (like malicious agents working to undermine the student).

E. Exercise Compiler

Compilation has two phases:

* All objects, actions, task graphs, and exercises are checked for consistency. This is necessary since the user can edit them in any order.

* The MEBuilder database (including constraints induced by the task graph) is translated into METUTOR facts in the form described in Section II; the rest of this section elaborates. Since METUTOR is not object-oriented, full inheritances of specifications must be done. Conditions become expressions of the form <property-value>(<object>), and actions become <action>(<object1>, <object2>, ...).

The recommended facts in METUTOR recommend an action to achieve some conditions, the "intended effects" from action definition. The facts, however, must be sorted for efficiency since METUTOR always tries the first recommended action first. We obtained good performance with sorting to satisfy these heuristics:

* The fact recommending action X precedes that for action Y if all occurrences of X precede all occurrences of Y in every task graph for the exercise.

* Facts for actions not used in any task graph for the exercise go at the bottom.

Precondition facts for METUTOR come from four sources:

* Explicit preconditions in the action's definition, like "fire is confronted" for "extinguish".

* The opposite of the intended effects of an action, like "fire is not out" for "extinguish" (for otherwise the action would not affect the state).

* Explicit preconditions from editing of a task graph (expected to be rare).

* Intended effects in the task graph just previous to this action (for otherwise the teacher should have put the two in parallel during editing of the task graph). An example is when "extinguish" occurs just after "set boundaries" in the teacher's task graph, from which we can infer that "boundaries are set" is a precondition of "extinguish". Unordered actions provide similar preconditions.

Preconditions may be inferred to be context-dependent to fully account for a task graph, especially when an action occurs more than once. Among the precondition facts for an action, the most restrictive ones go first.

Addpostcondition facts come from:

* Positive intended effects and side effects of an action, like "fire is out" and "area is watery" for the "extinguish" action.

* Positive "added side effects" in the task graph, like "there is a crater in the floor" when the user forgets to turn the power off.

Deletepostcondition facts come from:

* Negative intended effects and side effects, like "it is no longer true that boundaries are set" for the "extinguish" action.

* Negative "added side effects".

Postconditions may also need to be context-dependent. They are sorted like precondition facts.

Some randomness can be specified with the initial state of an exercise, like whether there is a medic present for firefighting (students must learn both cases). Some comes from probabilistic side effects specified in the task graph, like when extinguishing does not succeed at first. Other randomness comes from daemons with probabilities. All these can be mapped directly to randchange facts.

F. Assessment of MEBuilder

In one experiment, students in our M.S. program were given a task to implement with MEBuilder with no instructions besides the manual. The task was procedures for obtaining a Ph.D. degree, and required definition of eight props, one character, six property sets, and six actions. 6 subjects who had not used MEBuilder previously required 12.5 hours mean total time to do the assignment, and 4 subjects with about 10 hours experience required 4.7 hours. All tutors were correct. So METUTOR is not hard to learn.

Another experiment compared MEBuilder with a simple courseware authoring system for procedural skills that we wrote. Six subjects received detailed instruction about tutoring systems and details of MEBuilder (none had used it before). Subjects worked individually to write traditional courseware and an MEBuilder tutor for the same exercise. Four had an exercise of preparing to scuba dive (three of which did the courseware first), and two had an exercise of replacing a gasket on a car water pump (one of which did the courseware first). Both exercises were carefully developed to have 15 steps and 36 possible solutions so that measurements would be comparable. But students were given partially-constructed tutors.

The data structures the subjects produced with MEBuilder were identical to those of the authors. Those produced with the courseware for the dive problem were also identical to the authors (although missing potentially useful error transitions), but those for the gasket exercise missed a few states. Only one tutor had bugs.

The subjects averaged 1.8 hours using MEBuilder and 2.3 hours using the courseware. The major difference was the number of commands issued: 26 on the average with MEBuilder and 119 with the courseware. As teachers get more experienced with MEBuilder, we expect its commands will need less time, and the gap in development time should widen. And a good library of objects could help MEBuilder, because of the generality of its reasoning, more than a good library of state transitions could help courseware. And these were just simple exercises; with the 13,433 states of the firefighting exercise, the superiority of MEBuilder should be dramatic.

IV. References

[1] E. Steinberg, Teaching Computers To Teach, second edition. Lawrence Erlbaum, Hillsdale, NJ, 1991.

[2] D. Sleeman, "PIXIE: A Shell for Developing Intelligent Tutoring Systems." In Artificial Intelligence in Education, Volume 1, pp. 239-265, Ablex, Norwood, NJ, 1987.

[3] J. Anderson, A. Corbett, K. Koedinger, and R. Pelletier, "Cognitive Tutors: Lessons Learned." The Journal of the Learning Sciences, 4 (2), 1995, 162-207.

[4] G. Kearsley, "Authoring Systems for Intelligent Tutoring Systems on Personal Computers". In Instructional Designs for Microcomputer Courseware, ed. D. Jonassen, Lawrence Erlbaum, Hillsdale, NJ, 1988, 381-396.

[5] D. Guralnik and A. Kass, "An Authoring System for Creating Computer-Based Role-Performance Trainers." World Conference on Educational Multimedia and Hypermedia, Vancouver, Canada, June 1994, pp. 235-240.

[6] A. Newell and H. Simon, Human Problem Solving. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[7] N. Rowe, Artificial Intelligence Through Prolog. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[8] E. Hollnagel, "The Phenotype of Erroneous Actions". International Journal of Man-Machine Studies, 39 (1993), pp. 1-32.

[9] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[10] E. Sacerdoti, A Structure for Plans and Behavior. Elsevier, New York, 1977.

V. Bibliography

[11] J. Anderson and R. Pelletier, "A Development System for Model-Tracing Tutors". Proceedings of the International Conference of the Learning Sciences, Evanston, IL, 1991, 1-8.

[12] R. Burton, "Diagnosing Bugs in a Simple Procedural Skill." In Intelligent Tutoring Systems ed. D. Sleeman and J. Brown, pp. 157-183. Academic Press, London, 1982.

[13] A. Collins and J. Brown, "The Computer as a Tool for Learning through Reflection." In Learning Issues for Intelligent Tutoring Systems, ed. H. Mandl and A. Lesgold, pp. 1-18. Springer-Verlag, New York, 1988.

[14] T. Curran, and S. Keele, "Attentional and Nonattentional Forms of Sequence Learning." Journal of Experimental Psychology: Learning Memory, and Cognition, 19, 1 (1993), pp. 189-202.

[15] W. Johnson, "Developing Expert System Knowledge Bases in Technical Training Environments." In Intelligent Tutoring Systems: Lessons Learned, pp. 21-33. Lawrence Erlbaum, Hillsdale, NJ, 1987.

[16] M. Jones, A. Gibbons, and D. Varnier, "A Re-usable Algorithm for Teaching Procedural Skills." World Conference on Educational Multimedia and Hypermedia, Vancouver, Canada, June 1994, pp. 299-304.

[17] G. McCalla., J. Greer, and the SCENT Research Team, "SCENT-3: An Architecture for Intelligent Advising in Problem Solving Domains." In Intelligent Tutoring Systems: At the Crossroad of Artificial Intelligence and Education, Ablex Publishing, Norwood, NJ, 1990.

[18] M. Miller and S. Lucado, "Integrating Intelligent Tutoring, Computer-Based Training, and Interactive Video in a Prototype Maintenance Trainer." In Intelligent Instruction by Computer: Theory and Practice, ed. M. Farr and J. Psotka, Taylor and Francis, Washington DC, 1992, 127-150.

[19] T. Murray and B. Woolf, "Tools for Teacher Participation in ITS Design." Proceedings of Second International Conference on Tutoring Systems, Montreal, Canada, June 1992, pp. 593-600. Springer-Verlag, New York, 1992.

[20] D. Peachy and G. McCalla, "Using Planning Techniques in Intelligent Tutoring Systems." International Journal of Man-Machine Studies, 24 (1986), pp. 77-98.

[21] J. Psotka, L. Massey, and S. Mutter, Intelligent Tutoring Systems: Lessons Learned. Lawrence Erlbaum, Hillsdale, NJ, 1987.

[22] N. Rowe and F. Suwono, "Aiding Teachers in Constructing Virtual-Reality Tutors." Fourth Annual Conference on Artificial Intelligence, Simulation, and Planning in High Autonomy Systems, Tuscon, AZ, 1993, pp. 317-323.

[23] B. Woolf, D. Blegen, J. Jansen, and A. Verloop, "Teaching a Complex Industrial Process." In Artificial Intelligence in Education, Volume I. Ablex, Norwood NJ, 1987, pp. 413-427.

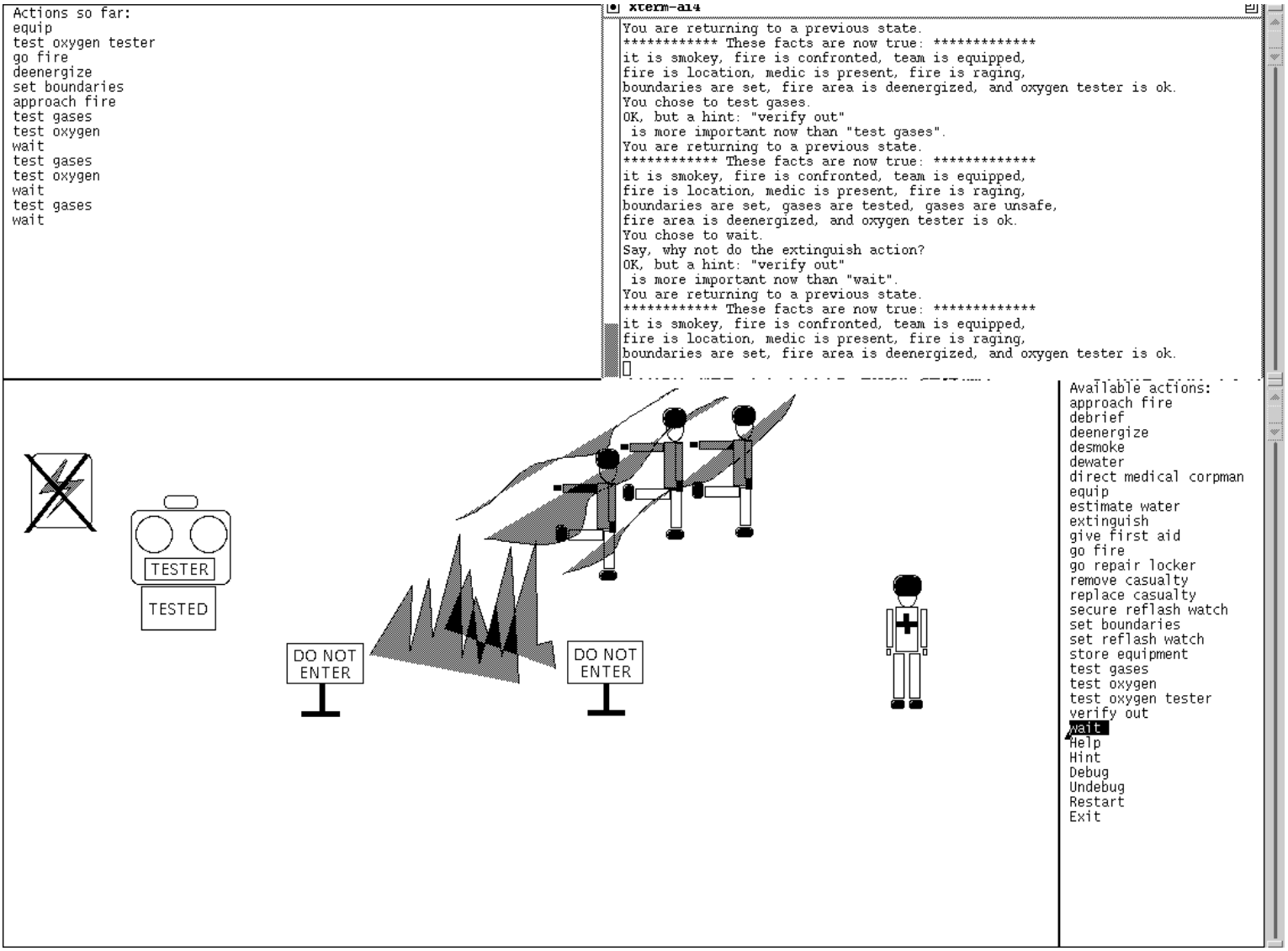

Figure 1: An example of a METUTOR interface for a firefighting tutor

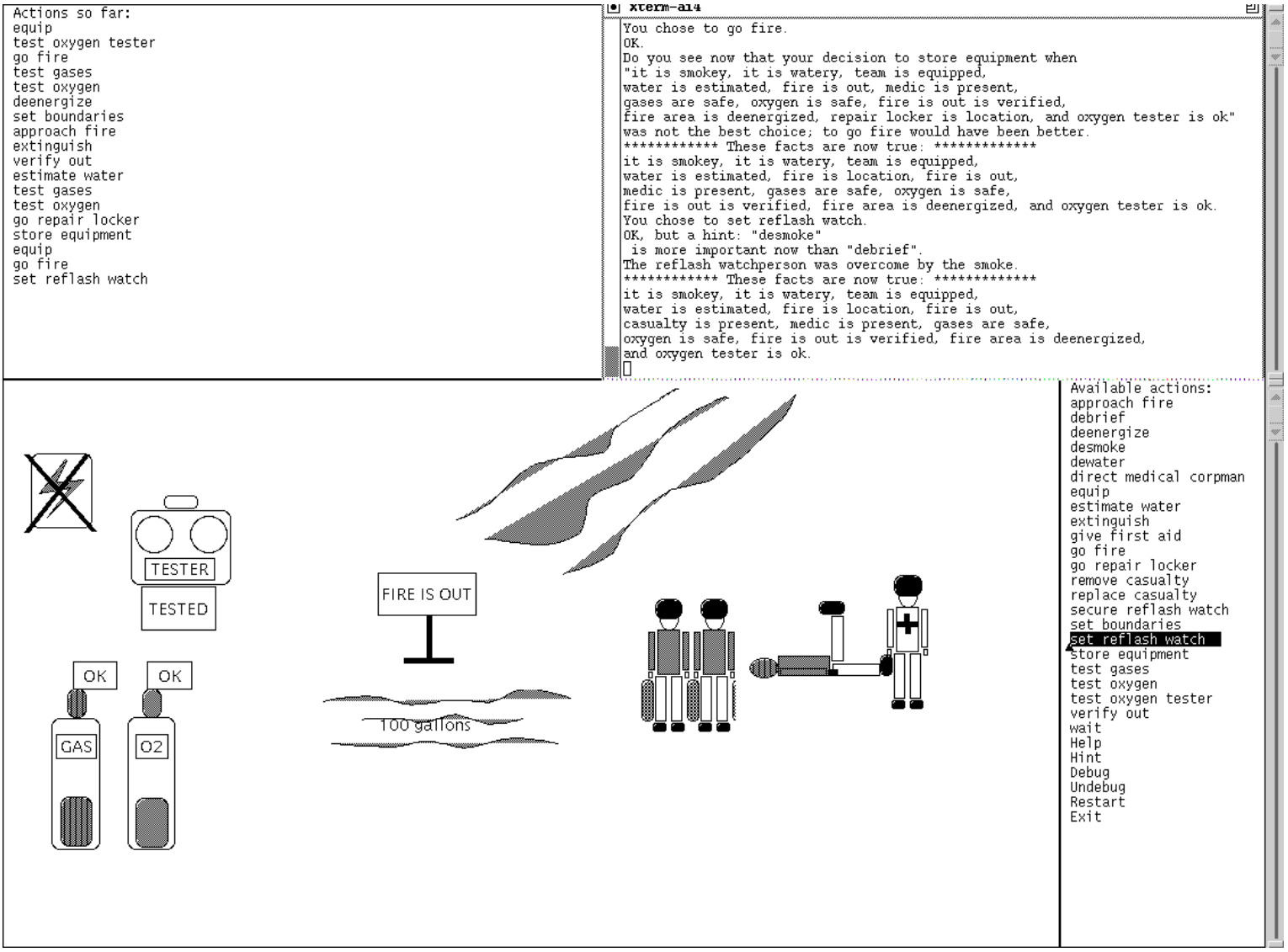

Figure 2: Another example from the firefighting tutor

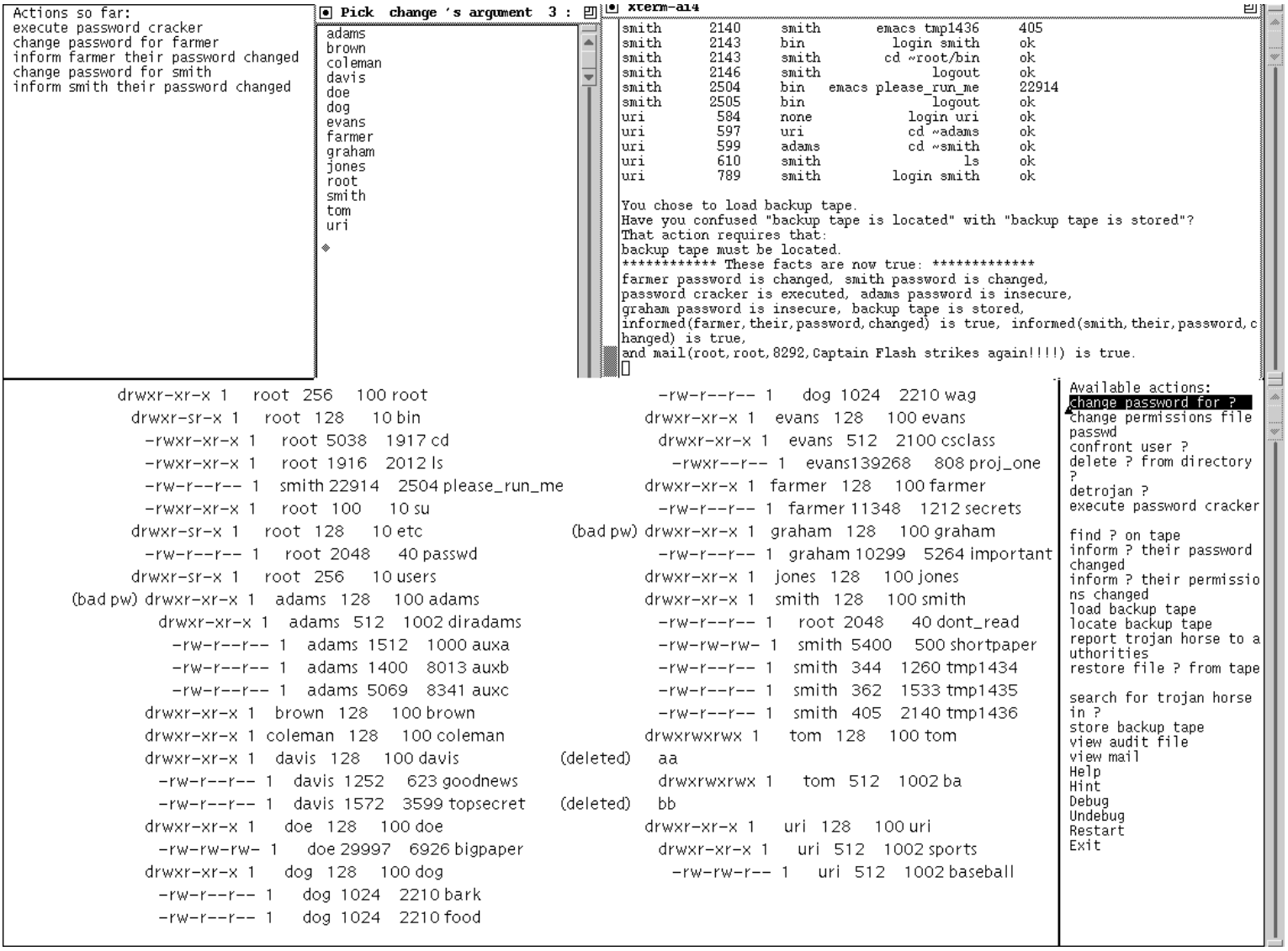

Figure 3: METUTOR interface for an intrusion-detection tutor

Tutor domain	Number of atom instances in task definition	Number of distinct atoms in task definition	Number of operators	Number of possible actions	Approximate time to build (hours)	Year completed	Notes
Firefighting on Navy ships	3978	224	23	23	50	1986-1995	Test example
Cardio-pulmonary resuscitation	2980	200	22	22	75	1988	Early project, some of its code now unnecessary
Ship-to-ship transfers at sea	3466	180	35	35	50	1989	Two-student tutor, each student with a workstation and own operators
F-4 aircraft fuel system malfunctions	5152	286	8	49	75	1990	Inefficient coding
Police procedures in hostage situations	2403	101	27	27	25	1993	Class project
Scuba diving procedures	2634	125	20	26	25	1993	Class project
Amphibious craft beaching procedures	2686	382	19	19	25	1994	Class project
Intrusion detection in Unix systems	2567	141	16	302	50	1995	Coordinates with an intelligent simulator of intrusive behavior

Figure 4: Statistics on some representative METUTOR tutors

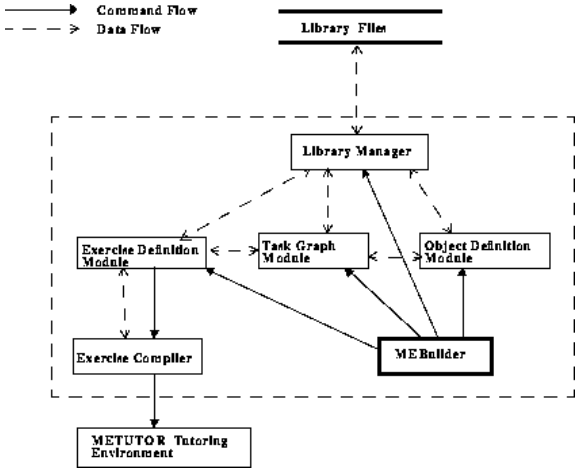

Figure 5: The architecture of MEBUILDER

MEBUILD> create object

What is the new object's name? ingredient

The following are the available parent objects.

[1] prop

[2] character

Choose ONLY one of the above> 1

Object "ingredient" saved.

MEBUILD> create object

What is the new object's name? frying pan

The following are the available parent objects.

[1] prop

[2] character

[3] ingredient

Choose ONLY one of the above> 1

Object "frying pan" saved.

MEBUILD> create object

What is the new object's name? hamburger

The following are the available parent objects.

[1] prop

[2] character

[3] frying pan

[4] ingredient

Choose ONLY one of the above> 4

Object "hamburger" saved.

MEBUILD> create object

What is the new object's name? cook

The following are the available parent objects.

[1] prop

[2] character

[3] frying pan

[4] hamburger

[5] ingredient

Choose ONLY one of the above> 2

Object "cook" saved.

MEBUILD> create property set

For which object? ingredient

What is the name of the property set? cookedness

New Property: ingredient is -- raw

New Property: ingredient is -- cooked

New Property: ingredient is -- burned

New Property: ingredient is -- end

Does this set correspond to information that could be

hidden from the student? Answer yes or no.>> no

Object "ingredient" saved.

MEBUILD> create property set

For which object? ingredient

What is the name of the property set? location

New Property: ingredient is -- in refrigerator

New Property: ingredient is -- in pantry

New Property: ingredient is -- in frying pan

New Property: ingredient is -- on counter

New Property: ingredient is -- end

Does this set correspond to information that could be

hidden from the student? Answer yes or no.>> no

Object "ingredient" saved.

MEBUILD> create operation

For which object? ingredient

Now specify other objects needed to perform

the operation.

[1] cook

[2] frying pan

[3] hamburger

[4] another ingredient

Choose one or more of the above or "none"> 2

Name the operation: fry ingredient

The following are possible intended effects:

[1] ingredient is raw

[2] ingredient is cooked

[3] ingredient is burned

[4] ingredient is in refrigerator

[5] ingredient is in pantry

[6] ingredient is in frying pan

[7] ingredient is on counter

Choose ONLY one of the above> 2

An assumed precondition is "ingredient is anything but cooked"

The following are possible preconditio

for "ingredient"

[1] ingredient is in refrigerator

[2] ingredient is in pantry

[3] ingredient is in frying pan

[4] ingredient is on counter

Choose one or more of the above or "none"> 3

The following are the possible normal side effects for "ingredient"

[1] ingredient is in refrigerator

[2] ingredient is in pantry

[3] ingredient is on counter

Choose one or more of the above or "none"> none

Object "ingredient" saved.

MEBUILD> create operation

For which object? ingredient

Now specify other objects needed to perform

the operation.

[1] cook

[2] frying pan

[3] hamburger

[4] another ingredient

Choose one or more of the above or "none"> 2

Name the operation: put ingredient in frying pan

The following are possible intended effects:

[1] ingredient is raw

[2] ingredient is cooked

[3] ingredient is burned

[4] ingredient is in refrigerator

[5] ingredient is in pantry

[6] ingredient is in frying pan

[7] ingredient is on counter

Choose ONLY one of the above> 6

An assumed precondition is

"ingredient is anything but in frying pan"

The following are possible preconditions

for "ingredient"

[1] ingredient is raw

[2] ingredient is cooked

[3] ingredient is burned

Choose one or more of the above or "none"> none

The following are the possible normal side effects

for "ingredient"

[1] ingredient is raw

[2] ingredient is cooked

[3] ingredient is burned

Choose one or more of the above or "none"> none

Object "ingredient" saved.

MEBUILD> create task

What is the new task's name? make hamburger

The primary actor is:

[1] character

[2] cook

Choose ONLY one of the above> 2

Now specify other objects needed for the task.

[1] another cook

[2] frying pan

[3] hamburger

[4] ingredient

Select one or more of the above or "none">> 2 3

The following are the current initial conditions for "hamburger".

Indicate which ones you want changed:

[1] hamburger is raw

[2] hamburger is in refrigerator

Choose one or more of the above or "none"> none

For this task, what is the effect on the "hamburger"?

Indicate which properties you want changed.

[1] hamburger is burned

[2] hamburger is on counter

Choose one or more of the above or "none"> 1

Choose the appropriate new objective:

[1] hamburger is raw

[2] hamburger is cooked

[3] hamburger is burned

[4] hamburger's cookedness are immaterial

Choose ONLY one of the above> 2

[1] hamburger is cooked

[2] hamburger is on counter

Choose one or more of the above or "none"> 2

Choose the appropriate new objective:

[1] hamburger is in refrigerator

[2] hamburger is in pantry

[3] hamburger is in frying pan

[4] hamburger is on counter

[5] hamburger's location is immaterial

Choose ONLY one of the above> 5

Indicate which properties you want changed.

[1] hamburger is cooked

[2] hamburger's location is immaterial

Choose one or more of the above or "none"> none

The following is my first attempt at solving the task.

[1] put hamburger in frying pan

[2] fry hamburger

Figure 6: MEBUILDER example

[1] Conduct preflight inspection on aircraft.

[2] Engage aircraft's external power.

[3] Engage huffer.

[4] Start aircraft's engine.

[5] Start aircraft's APU.

[6] Engage aircraft's APU's generator.

[7] Engage aircraft's APU's bleed air.

[8] All of these:

[8a] Subprocedure:

[8a1] Have pilot request flight clearance.

[8a2] Have pilot request taxi clearance.

[8b] Subprocedure:

[8b1] Disengage huffer.

[8b2] Disengage aircraft's external power.

[8b3] Check aircraft's NWS.

[8b4] Check aircraft's brakes.

[9] Taxi aircraft.

[10] All of these:

[10a] Subprocedure:

[10a1] Check wind sock.

[10a2] Check aircraft's trim

[10b] Subprocedure:

[10b1] Have pilot request takeoff clearance.

[11] Shut off aircraft's APU.

[12] Max aircraft's throttles.

[13] Fly the aircraft.

Figure 7: Task graph found by MEBUILDER for aircraft preflight actions