NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE STATE TRANSITION DIAGRAM WITH PATH PRIORITY AND ITS APPLICATIONS

by

Thomas Scholz

September 1993

Thesis Advisor: **Se-Hung Kwak**

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Master's Thesis

4. TITLE AND SUBTITLE
The State Transition Diagram With Path Priority and Its Applications (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Thomas Scholz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA. 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified/Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The overall software structure of the Naval Postgraduate School Autonomous Underwater Vehicle (NPS AUV) is the Rational Behavior Model (RBM), a tri-level, multilingual software architecture which is based on three levels of abstraction called the Strategic, Tactical, and Execution level. In this study, interests were focussed on the implementation of the Strategic level in CLIPS such that it exhibits the same behavior as the already existing implemenation written in Prolog.

As a tool for translating a backward chaining version of the Strategic level software (like Prolog) to a forward chaining one (like CLIPS), the State Transition Diagram With Path Priority (STDWP) was introduced in this study. Specifically, STDWP allows *graphical* translation between backward and forward chaining versions of the Strategic level.

This research shows empirically that the translation is always possible and that the two versions hold logical and behavioral equivalence. Thus, STDWP bridges two approaches in robot control which are based on forward and backward chaining.

14. SUBJECT TERMS	15. NUMBER OF PAGES
AND/OR Graph, Autonomous Vehicles, Intelligent Control, Mission Control, Rational Behavior Model, Robot Control, Rule Based Control, Rule Based Systems, Software Architecture, State Diagram, State Transition Diagram.	100
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

THE STATE TRANSITION DIAGRAM WITH PATH PRIORITY AND ITS APPLICATIONS

by

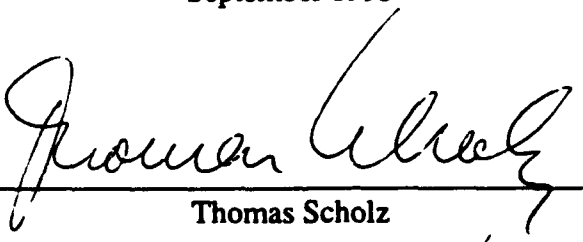
Thomas Scholz
Lieutenant Commander, German Navy
Diplom-Kaufmann, Universität der Bundeswehr Hamburg, 1983

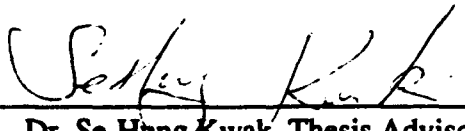
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

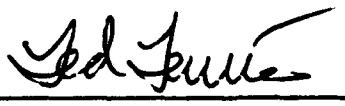
from the

NAVAL POSTGRADUATE SCHOOL
Monterey, California
September 1993

Author: 
Thomas Scholz

Approved By: 
Dr. Se-Hung Kwak, Thesis Advisor


Dr Yuh-Jeng Lee, Second Reader


Dr. Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The overall software structure of the Naval Postgraduate School Autonomous Underwater Vehicle (NPS AUV) is the Rational Behavior Model (RBM), a tri-level, multilingual software architecture which is based on three levels of abstraction called the Strategic, Tactical, and Execution level. In this study, interests were focussed on the implementation of the Strategic level in CLIPS such that it exhibits the same behavior as the already existing implemenation written in Prolog.

As a tool for translating a backward chaining version of the Strategic level software (like Prolog) to a forward chaining one (like CLIPS), the State Transition Diagram With Path Priority (STDWP) was introduced in this study. Specifically, STDWP allows *graphical* translation between backward and forward chaining versions of the Strategic level.

This research shows empirically that the translation is always possible and that the two versions hold logical and behavioral equivalence. Thus, STDWP bridges two approaches in robot control which are based on forward and backward chaining.

TABLE OF CONTENTS

A person with one watch knows what time it is;

a person with two watches is never sure.

anon.

I. INTRODUCTION

A. BACKGROUND

The link between intelligent behavior and mission planning and control of autonomous vehicles is one of the major issues in research efforts of artificial intelligence and robotics. It is apparent that the creation and incorporation of "intelligence" for truly autonomous vehicles that are able to conduct a mission fully independently requires serious considerations about the software architecture. One solution is the Rational Behavior Model (RBM), a multi-paradigm, tri-level architecture for the control of autonomous vehicles [Ref. 1]. The attribute "tri-level" is based on the three levels of abstraction, namely the strategic (top), tactical (middle), and execution (bottom) level. "Multi-paradigm", on the other hand, describes the utilization of different programming languages in order to exploit their strengths and to avoid their weaknesses wherever it is appropriate.

The RBM architecture has been implemented for the Naval Postgraduate School Autonomous Underwater Vehicle Model II (NPS AUV II) and has been successfully run in simulation using one workstation for each level and being connected via ethernet.

Initially, the three levels of RBM have been implemented in Prolog (strategic level), Classic-Ada (tactical level), and C (execution level). The hardware onboard the NPS AUV currently consists of two Gespac computers with an 80386/MS-DOS and a 68030/OS-9 processor. Unfortunately, there does not exist a PC-version for Prolog that provides an appropriate interface to Classic-Ada. The study presented here provides one solution for this problem. This is, as a major part of this study, the translation from Prolog to CLIPS has been accomplished for the strategic level. The translated CLIPS implementation of the strategic level now works with a tactical level which has been completed in Ada in another study [Ref. 2].

B. OBJECTIVES

In this thesis, interests were focused on the implementation of the strategic level by rule-based programming languages and on the investigation of a translation between two programming languages whose production systems are based on backward and forward chaining, respectively, will exhibit the same behavior on mission control and execution of robust vehicles, such as the NPS AUV II. In other words, is it possible to achieve both logical and behavioral equivalence of two different chaining implementations for the strategic level?

C. SCOPE

This study was specifically focussed on three major tasks:

(1) Discussing the graphical representations of rule based programming languages; in particular, AND/OR goal trees for a language with a backward chaining inference engine and State Transition Diagrams With Path Priority (STDWP) for a language with a forward chaining inference engine. The main emphasis is put on the discussion of STDWP since it was introduced newly in [Ref. 3].

(2) Applying both graphical representations as tools for the implementation in the corresponding programming language.

(3) Investigating the logical and behavioral equivalence of the example programs and the CLIPS implementation for the RBM's strategic level for the NPS AUV.

D. ORGANIZATION

This study contains two major parts:

The first part covering Chapters II and III, discusses state diagrams and state tables in general and the background of STDWP's. The STDWP is introduced and presented as a tool for graphical representation of a forward chaining rule-based system. Its terminology and definitions are explained as well in this context.

The second part covering Chapter IV, applies the STDWP as a means of translating backward and forward chaining implementations from basic to more complicated

examples. Finally, the translation of the RBM's strategic level for the NPS AUV from a backward to a forward chaining implementation is presented.

The summary and conclusions are made in Chapter V, where also an outlook on the STDWP's general applicability is ventured and where challenges on future research are discussed.

II. STATE TABLES AND STATE DIAGRAMS IN GENERAL

State tables and state diagrams have been mostly used in switching theory and for logic design to represent sequential systems, such as logic circuits. Therefore, a brief overview of logic circuits and of available tools for designing such circuits are presented in the following sections. State tables and state diagrams will be discussed in particular since they constitute the background for "State Transition Diagrams with Path Priority" as will be shown in Chapter III.

A. OVERVIEW

Logic circuits are basically classified into two groups, namely combinational and sequential circuits. [Ref. 2] [Ref. 3]

A combinatorial circuit is defined as a circuit whose output at any time is a function of its input signals without regard to previous inputs; i.e., a domestic lighting circuit controlled by an ordinary tumbler switch turns the light "on" if the switch is up, it turns it "off" if the switch is down.[1]

A sequential circuit is defined as a circuit whose output at any given time is determined by the order in which the input signals are applied; i.e., a lighting circuit controlled by a cord-pull. The effect of pulling the cord depends on the state the circuit is currently in; if the light is "on" then pulling the cord will turn it "off", if the light is "off", the light will be turned "on".

Sequential circuits are further classified into

(1) event-driven,

(2) clock-driven, and

(3) pulse-driven circuits,

---

1. Combinational circuits are not further discussed in this study.

each of which can be determined in more detail as a cyclic or an non-cyclic circuit depending on the fact whether the circuit returns to its initial state or not.[2]

Several methods are available to design and to describe sequential circuits:

(1) Verbally, by the means of *word statements*,

(2) Diagrammatically, by the means of *state diagrams*,

(3) Tabularily, by the means of *state tables*, and

(4) Algebraically, by the means of Boolean statements, loosely referred to as *sequential equations*. [Ref. 3]

Verbal statements are subject to misinterpretation and that is why they tendentiously cause ambiguity. So verbal statements are not a recommendable tool. A state diagram, on the other hand, is free of ambiguities and can be easily understood because of its clear representation. State tables are in general utilized to depict the sequence (also the time sequence) of a circuit's input and output. They can also be helpful to reduce the size of a circuit when such a reduction is possible. Sequential equations are mostly used for engineering purposes before the circuit is actually implemented. [3]

B. STATE TABLES

The form and features of state tables vary throughout the relevant literature. State tables are usually applied and modified to certain requirements. However, according to [Ref. 3], [Ref. 4], and [Ref. 5] most state tables have the following features in common.

A state table has as many rows as states and as many columns as possible input signals (or states). In each square of the table, the next state is entered as result of the column's input applied to the state of this row.[4] If there is no next state the intersection in the table is left blank.

---

2. For more details it is referred to [Ref. 3], [Ref. 4], and [Ref. 5].
3. Tool (1) and (4) will not be discussed in more detail since they are not useful for the objectives of this study.
4. Therefore, state tables are sometimes called "next-state tables" [Ref. 2]

A simple example of a state table is given in TABLE 1.

TABLE 1: AN EXAMPLE STATE TABLE

	Input			
States	X_1	X_2	X_3	X_4
A		B	D	
B	D			B
C	C			A
D	D			C

The circuit depicted by the state table in TABLE 1 issues four states, namely "A", "B", "C", and "D". Possible inputs are "X_1", "X_2", "X_3", and "X_4". For example, state "A" transitions either to state "B" if input "X_2" is existent, or to state "D" if input "X_3" is existent. No transitions are made from "A" if the input "X_1" or "X_4" occurs, so there are no next-states in these cases.

Additional data, such as an initial state, a final or accepting state, or a certain output after a transition, may be built in a state table as well. For this study, however, the discussed features are satisfactory.

C. STATE DIAGRAMS

The information available in a state table is graphically represented in a state diagram. A state diagram in general consists of nodes, directed lines[5], and input / conditions. The nodes represent states. The arrows link the nodes and represent the interstate transitions. An arrow connecting a node with itself indicates that no change of state occurs. The input / conditions are usually inserted above or below the arrow representing the corresponding transition.

5. In the following, a directed line is referred to as an "arrow".

The example given in TABLE 1 is graphically displayed as a state diagram in Figure 1.

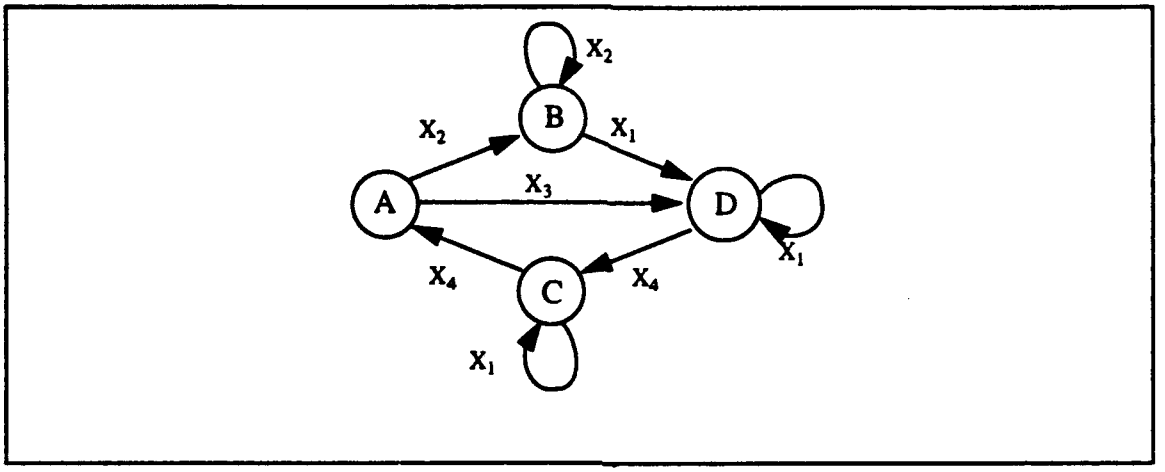


Figure 1: State diagram derived from the state table in TABLE 1

There is no difference between a state table and a state diagram, except for the different representation. A state diagram follows directly from a state table and gives a pictorial view of the state transitions and is very suitable for human interpretation. By the same token, state diagrams are often used as initial design specification of sequential circuits, machines, or systems [Ref. 6].

III. THE STATE TRANSITION DIAGRAM WITH PATH PRIORITY (STDWP)

A. INTRODUCTION

A state transition diagram (STD) is a modeling tool for describing a time dependant behavior of a system. It is represented as a graph that consists of nodes, each representing one of the possible states of the system, and of arrows which represent the valid state transition among the states. A state transition occurs only in case of a detected specific condition. A state is defined as a set of attributes which characterizes the system at a given time. No knowledge of past inputs is needed at this moment to determine the further behavior of the system.

The ordinary STD is extensively used as a tool of switching theory and logical design. In this chapter, a new extended STD is introduced which is called a *State Transition Diagram With Path Priority* (STDWP)

B. DEFINITIONS AND EXAMPLES

The notation used for conventional STD's requires that responses, in the form of state transitions, are identified for all possible conditions. This may lead to states which have two or more successor states. To avoid nondeterminism, the conditions for each transition path must be mutually exclusive. For example, if a set of condition α is needed to trigger a transition from state "A" to state "B", and another set of condition β causes a transition from state "A" to state "C", then it must be true that $\alpha \cap \beta = \varnothing$.

Unfortunately, the approach adopted by conventional STD's, in order to avoid nondeterminism, limits its applicability to real world problems. In [Ref. 7], Kwak et alteri presented a new approach to relieve this limitation. It is called a *State Transition Diagram With Path Priority* (STDWP). A STDWP has all the characteristics of a conventional STD.

However, unlike a STD, a STDWP allows a state to have two or more successor states with non-mutually exclusive state transition conditions. For example, if a set of condition α is needed to trigger a transition from state "A" to state "B", and another set of condition

β causes a transition from state "A" to state "C", then either $\alpha \cap \beta = \varnothing$ or $\alpha \cap \beta \neq \varnothing$ is allowed. The latter case is an unique characteristic of a STDWP. Additionally, in a STDWP, nondeterminism caused by $\alpha \cap \beta \neq \varnothing$, is taken care of by priorities among possible non-unique state transition paths. That is, if a specific condition makes multiple state transition paths eligible, then only one state transition path with the highest path priority will be selected and the corresponding state transition will be made.

Definiton (1): A *State Transition Diagram With Path Priority* (STDWP) has the characteristics of a conventional STD and allows for each state two or more successor states with non-mutually exclusive state transition conditions. Non-mutually exclusive state transitions issue path priorities, such that only one transition will actually be made.

A portion of a STDWP is shown in Figure 2. There are four states, "A", "B", "C", and "D". Suppose, "A" is the current state, then "B", "C", and "D" are successor states. The state transition conditions are shown as combinations of "X", "Y", and "Z". The path priority for the appropriate state transition conditions is also shown with numerical values for "p", namely "p=1", "p=2", and "p=3" where a bigger number means a higher priority. For example, the state transition conditions from "A" to "D" are "X" or "Y" with path priority "p=2". Therefore, there are three state transition paths which have non-mutually exclusive state transition conditions. For example, if condition "X" is currently met, all three state transition paths become eligible. However, only one state transition path is actually made from "A" to "C", that is the path with the highest path priority, in this case

the transition from "A" to "C". This fact is documented by a *competing arc* which connects the relevant state transition arrows.

Definiton (2): If a state in a STDWP has multiple successor states with non-mutually exclusive state transition conditions, then a *competing arc* indicates the competing relationship among these successor states.

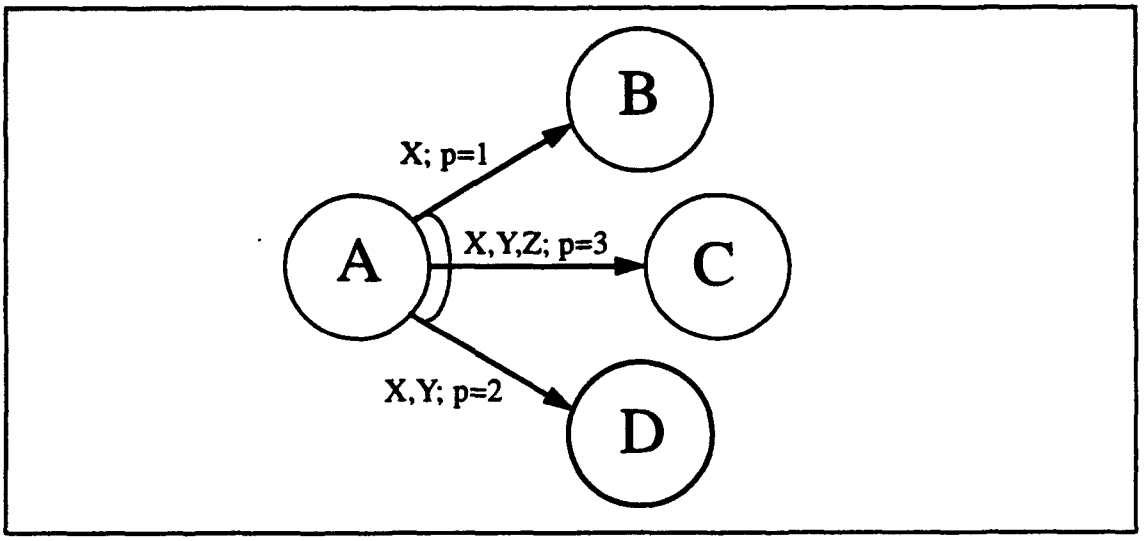


Figure 2: A portion of a State Transition Diagram With Path Priority

Therefore, a state transition in a STDWP is composed of two stages of operation:

First, *activate* possible state transition paths. In the previous example, all three state transition paths were activated with condition "X".

Second, *execute* the state transition which has the highest path priority among activated state transition paths. This sequence of two stages is called the *activation stage* and *execution stage*, respectively.

In Table 2, for all possible combinations of "X", "Y", and "Z" conditions, the corresponding state transitions are tabulated with the two state operations. The numerical entries in the left column indicate if the corresponding condition(s) exist(s) or not. "0" means absence of the corresponding condition, and "1" means its existence. The alphabetic entries in the other columns show either activated state transitions (see center column) or

executed state transitions (see right column); i.e., "AB" in the right column means an actual state transition from "A" to "B" if the condition combinations "X" or "XZ" are existent.

TABLE 2: Transition Table for STDWP in Figure 2

Condition XYZ	Activated Transition	Executed Transition
000	-	-
001	-	-
010	-	-
011	-	-
100	AB	AB
101	AB	AB
110	AB, AD	AD
111	AB,AC,AD	AC

Another major difference to conventional STD's is an implicit backtracking feature built in a STDWP. In other words, if the available input does not satisfy any of the explicit state transition conditions from the current state, then backtracking will be initiated. That is, the current state is changed to that previous state which has multiple successor states and is closest to the current state. This discovery leads to the following definitions.

Definiton (3): A *branching state* is a state with multiple successor states. If a branching state is the first state that can be arrived by traversing against the normal state transition direction, then it is called a *closest previous branching state*. However, a branching state itself cannot be a closest previous branching state.

Definiton (4): A branching state may have a competing arc and/or path priorities if there is a non-disjoint state transition condition.

Definiton (5): A state with multiple previous states is called a *merging state* and has no closest previous branching state. Successor states of a merging state also have no closest previous branching state unless one of the successor states becomes a branching state. A starting state is treated as a merging state.

Therefore, if any condition of all possible state transitions cannot be met, then the state changes to the closest previous branching state. This transition is called backtracking

For example, Figure 3 shows a STDWP with six different states. Note, that the STDWP in this figure has disjoint state transition conditions. Thus, there are no competing arc and no path priorities. "A" is the closest previous branching state for state "B", "C", "D", and "E". "F", however, does not have a closest previous branching state since it is a

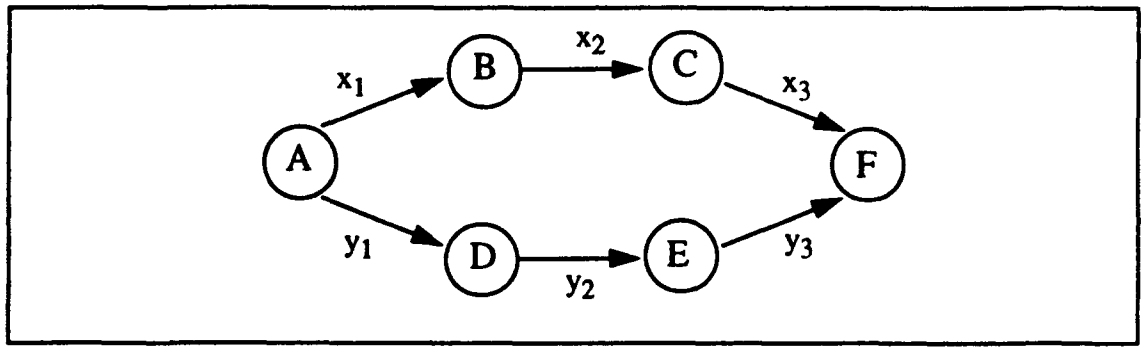


Figure 3: A STDWP with six states

merging state. The backtracking paths for each state may be explicitly drawn in a STDWP as shown in Figure 4, but are preferably not included to keep the drawing simple.

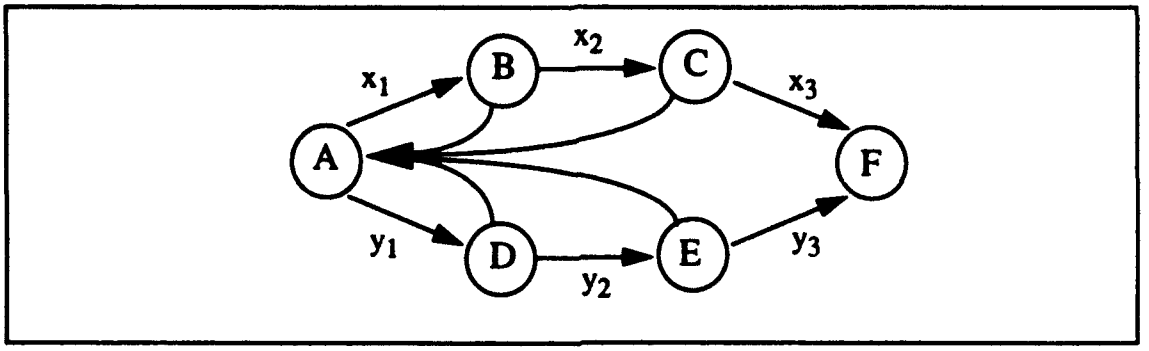


Figure 4: A STDWP showing implicit backtracking paths

One possible scenario of state transitions is shown in Table 3. Since the required state transition condition is available at the right time a sequence of state transitions from "A" to "F" via "B" and "C" can be conducted without any backtracking.

TABLE 3: Scenario without backtracking

Time	Current State	Input	Next State
t_1	A	x_1	B
t_2	B	x_2	C
t_3	C	x_3	F

Another scenario is shown in Table 4. After the state transition from "A" to "B" and from "B" to "C" are carried out, input x_3 is not available at time t_3. At this moment, rather than stopping and waiting for an external input indefinitely, backtracking is started to the closest previous branching state, which is state "A". Fortunately, the external input y_1 is available. Thus, the state transition to "D" is possible and the path eventually arrives at "F".

However, if the external input y_1 were not available, then no further state transition would be done and state "F" would never be reached.

TABLE 4: Scenario with backtracking

Time	Current State	Input	Next State
t_1	A	x_1	B
t_2	B	x_2	C
t_3	C	y_1	A
t_4	A	y_1	D
t_5	D	y_2	E
t_6	E	y_3	F

In the above discussion, only one state transition condition was utilized for each state, and all state transition conditions were unique. This setup does not imply a limitation of STDWP; also was t_i introduced just for demonstration purposes. Like common STD's STDWP is capable of representing both, synchronous and asynchronous states.

A STDWP can also be organized as a multi-level STD [Ref. 8] where an individual state of a higher-level diagram can encapsulate a lower-level diagram. Then the final state(s) in the lower-level diagram is(are) identical to the exit conditions of the higher-level state. This decomposition may be recursively applied to give order and understandability to an otherwise complex diagram, the decomposition represents a standard approach to the design and analysis of complex, state-based systems.

C. SIGNIFICANCE AND APPLICABILITY

The introduction of STDWP's facilitates the graphical representation of a new class of problems. In [Ref. 9] and [Ref. 10] STDWP's were applied to represent and explain high-level control of the NPS AUV-II graphically. Another application of a STDWP is the graphical representation of a forward chaining rule-based system. Such a system operates in two steps, rule activation and rule-firing [Ref. 11]. This exactly corresponds to the

STDWP's activation and execution stages. Thus, competing behaviors among rules in the rule agenda of a forward chaining rule-based system can be graphically depicted, and rule-firing can be graphically modeled, too.

As a consequence, STDWP's build a bridge between a forward chaining rule-based system as domain of Artificial Intelligence and a control based system represented by a conventional STD. Practical engineers with background of STD's and with a little knowledge of STDWP's can easily extend this idea to a high-level control of complex autonomous robots without learning an entirely new forward changing rule-based system. This advantage turns out to be very beneficial for the control of autonomous robots, because rule-based control of robot systems is very promising for design and coding of mission logic, as it was reported in the Rational Behavior Model software control architecture research of [Ref. 12] and [Ref. 9]. The impact of STDWP's is just unfolding, their application yet to be discovered for the future.

The following sections of this study show the application of STDWP's in general and for RBM software control architecture in particular. The RBM is mainly designed for control of a real-time autonomous agent, and will be briefly described later.

IV. APPLICATIONS OF THE STDWP

A. INTRODUCTION

In this chapter, the very basic structures, "AND-Relationship" and "OR-Relationship", that may be found in any portion of a Prolog program are discussed and depicted by STDWP structures. Implementations in CLIPS are then derived from these STDWP structures. In order to show how this translation pattern can be applied to more complex structures, two small Prolog programs are introduced later. They are first displayed by an AND/OR goal tree (whose definition will follow in the next section) and then translated in a STDWP. Finally, the STDWP is implemented in CLIPS code.

Either program follows a different approach of implementation. As it will be shown, the second implementation predominates the previous one by its assets, although both programs are equivalent to the initial Prolog code, in particular in regard to logic and behavior.

The final section of this chapter briefly introduces the Rational Behavior Model (RBM) and its strategic level, one of three main components of the RBM. The strategic level is represented graphically by an AND/OR goal tree and by a STDWP. For either representation the corresponding implementation is explained and presented.

B. BASIC STRUCTURES

1. AND-Relationship

A simple example of a Prolog rule and its notation as an AND/OR goal tree is shown on the left hand side of Figure 5. An AND/OR goal tree is defined as follows:

"An *AND/OR goal tree* is a graphical representation of AND/OR goal decomposition, with the root node representing the root goal, the leaf nodes representing the primitive goals, all other nodes representing intermediate goals subject to further decomposition, and the connecting arcs representing the logical relationship between subgoals and the goal from which they were decomposed." [Ref. 9]

In the given example, "A" is the goal, "B" and "C" are subgoals. Both subgoals have to be satisfied to satisfy the goal "A". This relationship is graphically shown in an AND/OR goal tree by introducing a parent node "A" and two children nodes, "B" and "C". The "AND-Relationship" between "B" and "C" is depicted with an arc connecting the adjacent branches. The AND/OR goal tree which is equivalent to the Prolog rule is now translated in a STDWP. The result is shown on the right hand side of Figure 5.

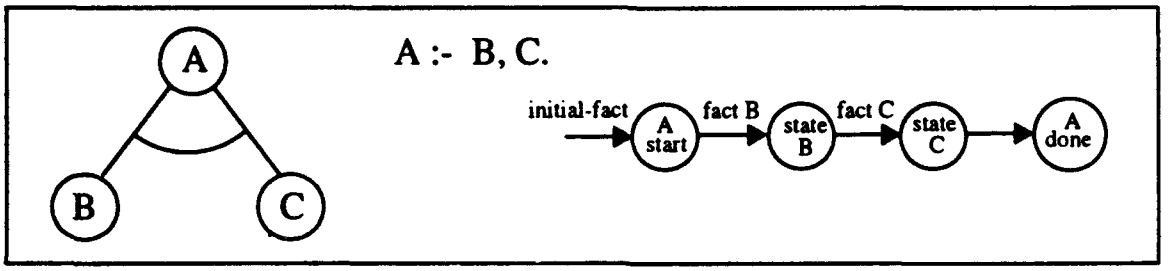


Figure 5: Example of an "AND-Relationship"

Specifically, the "A" node of the AND/OR goal tree corresponds to the "A done" state in the STDWP, and the "B" and "C" nodes are equivalent to the "state B" and "state C" states, respectively. The "A start" state is a special state which is introduced to show the current mode of an operation of a STDWP and to signal that the STDWP is trying to achieve goal "A". Only when the two successor states are satisfied, the "A done" state can be reached. The "A done" state itself signals the achievement and completion of goal "A". The previous discussion is tabulated in Table 5:

TABLE 5: Correspondence of nodes and states

AND/OR Goal Tree	STDWP
A	A done
B	state B
C	state C

In CLIPS, the states of a STDWP are implemented by the *deftemplate* construct. For our example, the *deftemplate* construct looks as follows:

```
(deftemplate rule-A
  (field state
   (type SYMBOL)
   (allowed-symbols   start  B  C  done)))
```

The *deftemplate* construct in CLIPS actually defines a group of related facts, in this case all possible states of rule "A". These facts will be used in CLIPS to control the flow in the STDWP. Thus, the equivalent CLIPS code to the given Prolog rule looks as shown in Figure 6.

```
(deftemplate rule-A
   (field state
      (type SYMBOL)
      (allowed-symbols     start  B  C  done)))

(defrule start-program        ;corresponding notation in the STDWP:
   ?x <- (initial-fact)
=>                                                    initial-fact  A
   (retract ?x)                                                     start
   (assert (rule-A (state start)))

(defrule rule-A-state-B
   ?x <- (rule-A (state start))                       fact B       state
   (fact C)                                                        B
=>
   (retract ?x)
   (assert (rule-A (state B))))

(defrule rule-A-state-C
   ?x <- (rule-A (state B))                           fact C       state
   (fact C)                                                        C
=>
   (retract ?x)
   (assert (rule-A (state C))))

(defrule rule-A-state-done                                        A
   ?x <- (rule-A (state C))                                       done
=>
   (retract ?x)
   (assert (rule-A (state done)))
```

Figure 6: CLIPS code for the "AND-Relationship" shown in Figure 5

2. OR-Relationship

A Prolog example showing "OR-Relationship" and its AND/OR goal tree are shown on the left hand side of Figure 7. Goal "A" can be reached by either subgoal, "B" or "C". Rule "A :- B." takes precedence over "A :- C." because of the textual order of the rules.

After translating the corresponding AND/OR goal tree in a STDWP two branches are obtained, both of which start at the state "A start" which is a branching state and end in the merging state "A done". Note that there is a competing arc between the two branches in the STDWP to emphasize the "OR-Relationship" with a non-disjoint state transition condition. The upper branch corresponds to the rule with higher priority, the lower branch to the rule with lower priority, respectively. This fact is shown with the path priorities "0" and "-1".[1]

The reason is as follows: Two Prolog rules with the same goal participate in an "OR-Relationship" and have an implicit order. The corresponding AND/OR goal tree also has the same implicit order; i.e., from left to right.[2] Thus, the upper branch of the STDWP corresponds to the left branch of the AND/OR goal tree and has a higher priority than the

1. Since CLIPS assigns a salience of "0" to any rule per se this study follows this automatism and assigns a salience of "-1" to the lower branch; the next branch would obtain "-2", and so forth.
2. Therefore, an AND/OR goal tree is equivalent to an AND/OR tree with a depth-first-goal traverser.

lower branch. In order to satisfy this fact, the saliences "0" and "-1" are assigned to the appropriate branches.

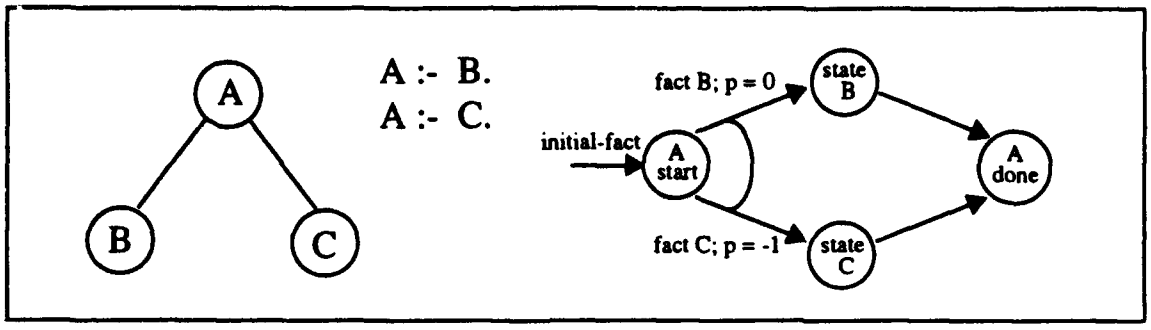


Figure 7: Example of an "OR-Relationship" with single states in each branch

Similar to the example for an "AND-Relationship", a *deftemplate* construct is built with the states "start", "B", "C", and "done" referring to the states in the STDWP. However, an additional *deftemplate* construct is necessary for three reasons caused by the implementation:

(1) to facilitate the required backtracking capability,

(2) to avoid confusion with equal states in different branches, and

(3) to identify the paths associated with "OR-Relationship".

This *deftemplate* construct may be called "A-branches" and shall represent the upper path by "branch A-1", and the lower branch by "branch A-2". Thus, the two deftemplate's are constructed and will look as follows:

```
(deftemplate rule-A
  (field state
    (type SYMBOL)
    (allowed-symbols   start  B  C  done)))
(deftemplate A-branches
  (field branch
    (type SYMBOL)
    (allowed-symbols     A-1  A-2 )))
```

As mentioned earlier, the "A start" state in the STDWP includes the initialization of an operation mode. In this case, the "A start" state acts as a "traffic control state". In the

case of an "AND-Relationship" the traffic control functionality is not necessary since there exists only one path in the STDWP and backtracking is not an issue at all since there is no closest previous branching state. However, if there are different branches as in an "OR-Relationship", then we need to provide traffic rules to control the backtracking path to the closest previous branching state and to start a new branch. That is (see Figure 8), if the transition from "B" to "A done" fails because "fact B" is not true or not available, then the state "A done" will not be reached by the upper branch. Instead, the lower branch is activated by the traffic rule "A-branch-2-start" (after backtracking to the closest previous branching state "A start") which awaits the failure of the upper branch and asserts the fact "(branch A-2)", so that the lower branch is tried. However, this attempt could also fail, if "fact C" does not exist.

If a branch fails, the fact-list will still contain the traffic control facts asserted by the traffic rules and one of the facts that has been most recently asserted by one of the rules in this branch. This circumstance could possibly cause confusion with repeated states or similar constructs within the STDWP. To avoid such an insufficiency an additional traffic rule called "clean-up" takes facts left over from the last failed branch off the fact-list and cleans it up for later assertions.

```
(deftemplate rule-A
        (field state
           (type SYMBOL)
           (allowed-symbols        start  B  C  done)))

(deftemplate A-branches
        (field branch
           (type SYMBOL)
           (allowed-symbols        A-1  A-2)))

(defrule start-program
        ?x <- (initial-fact)                                    initial-fact  A
=>                                                                            start
        (retract ?x)
        (assert (rule-A (state start)))

;-----------------------------------------Traffic rules------
(defrule A-branch-1-start
        ?x <- (rule-A (state start))
=>
        (retract ?x)
        (assert (A-branches (branch A-1))))

(defrule A-branch-2-start
        (declare (salience -1))
        ?x <- (A-branches (branch A-1))
        ?y <- (rule-A (state ?))
=>
        (retract ?x)
        (retract ?y)
        (assert (A-branches (branch A-2))))

(defrule A-branches-clean-up
        (declare (salience -2))
        ?x <- (A-branches (branch A-2))
        ?y <- (rule-A (state ?))
=>
        (retract ?x)
        (retract ?y)
;-------------------------------------------------------------

;-----------------------------------------------A-1-branch----
(defrule A-branch-1-state-B                                     fact B  state
        (A-branches (branch A-1))                                       B
        (fact B)
=>
        (assert (rule-A (state B)))

(defrule A-branch-1-state-done
        ?x <- (A-branches (branch A-1))                                 A
        ?y <- (rule-A (state B))                                        done
=>
        (retract ?x)
        (retract ?y)
        (assert (rule-A (state done)))
;-------------------------------------------------------------

;-----------------------------------------------A-2-branch----
(defrule A-branch-2-state-C
        (A-branches (branch A-2))                               fact C  state
        (fact C)                                                        C
=>
        (assert (rule-A (state C)))

(defrule A-branch-2-state-done
        ?x <- (A-branches (branch A-2))                                 A
        ?y <- (rule-A (state C))                                        done
=>
        (retract ?x)
        (retract ?y)
        (assert (rule-A (state done)))
;-------------------------------------------------------------
```

Figure 8: CLIPS code for the "OR-relationship" in Figure 7

C. A MORE COMPLICATED PROBLEM

1. Implementation

In order to show how basic structures are combined in a STDWP and how a given construct is implemented in CLIPS, the following steps are shown in the following.

First, a simple, RBM's strategic-level alike Prolog program is introduced. An AND/OR goal tree is then constructed after this program which will be followed by the transition to a STDWP. Then the equivalent CLIPS code is produced.

(1) The simple Prolog program is shown in Figure 9.

```
A :- B, C, D.
B :- D, E.
B :- F.
C :- D, E, F.
C :- D, F.
C :- E.
D :- G.
D :- H.
```

Figure 9: A simple Prolog program

(2) The corresponding AND/OR goal tree is shown in Figure 10.

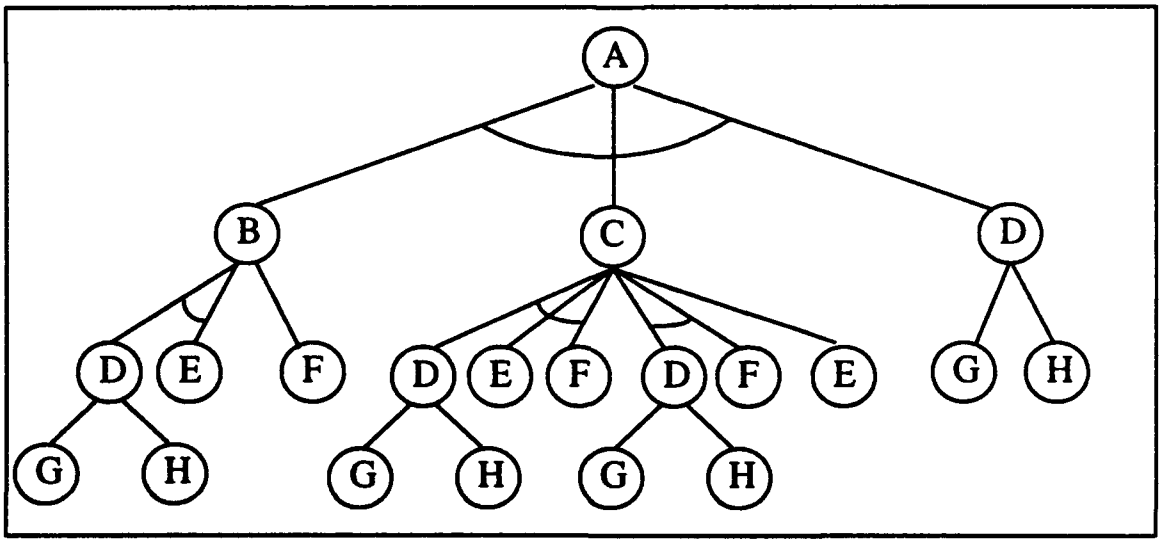


Figure 10: AND/OR goal tree for the Prolog example in Figure 9

(3) Similar to the example for an AND-Relationship, rule "A" of the Prolog code is represented in a STDWP as shown in Figure 11. This construct corresponds to the top level of the AND/OR goal tree shown in Figure 10. The states "B", "C", and "D", however, encapsulate lower-level STDWP's. In this case we talk about a multi-level STDWP.

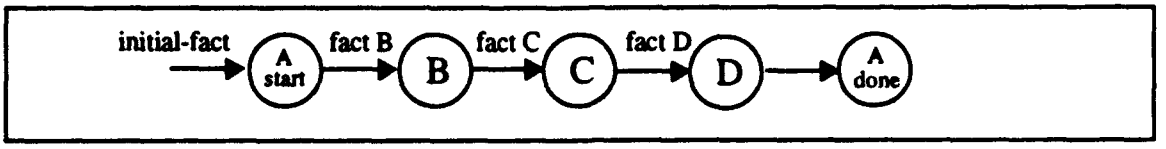


Figure 11: Multi-level STDWP with encapsulated states

(4) The multi-level STDWP shown in Figure 11 is explicitly expanded in Figure 12. [3]

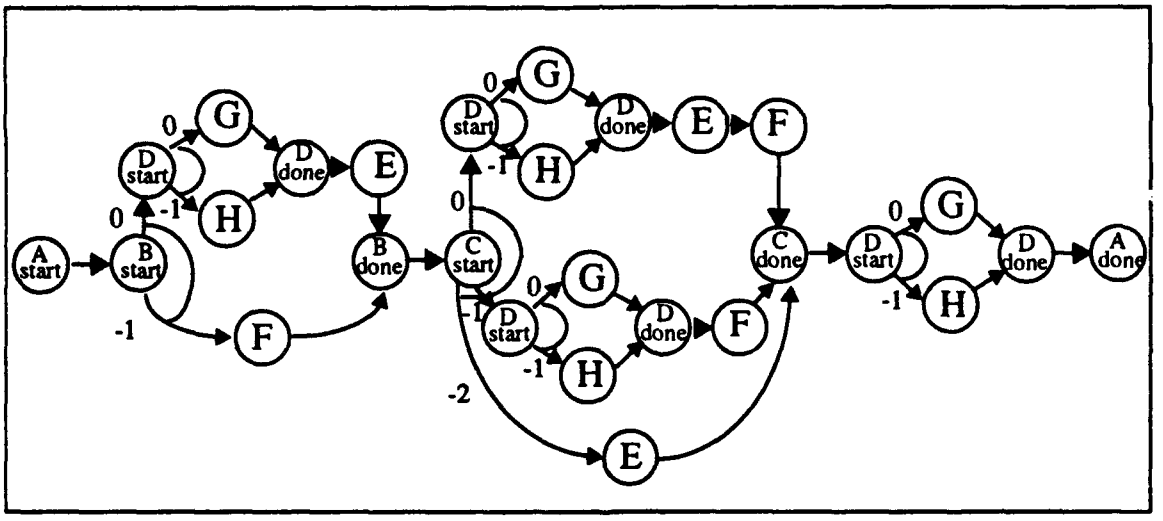


Figure 12: Complete STDWP for the Prolog example in Figure 9

It can be easily inferred from the Prolog code that the subgoals "E", "F", "G", and "H" are facts since there is no rule corresponding to those subgoals. Therefore these facts

3. In order to simplify the drawing, state transition conditions are not shown. However, path priorities and competing arcs are drawn for clarity.

are represented as leaves in the AND/OR goal tree. At least one fact, "E" or "F", is required to reach the goal "B" and "C", respectively; and either condition "G" or "H" satisfies the goal "D". Therefore, any combination of states ("E" or "F", and "G" or "H") eventually facilitates a path to the goal "A" successfully (see Figure 12 and also Figure 10).

The state transition table for the STDWP of Figure 12 is shown in Table 6. It tabulates the inputs which are required to reach the goal state "A done".

TABLE 6: State transition table for the STDWP

Current State	Input	Possible Next State
E	fact E	F; B done; C done
F	fact F	B done; C done
G	fact G	D done
H	fact H	D done

2. Testing

Two test cases may be used to support the above observation and to illustrate how the CLIPS program shown in APPENDIX A determines the path through the STDWP. The first test case provides "fact E" and "fact H" to the STDWP. This will furnish the STDWP

to reach the state "A done". As expected, the corresponding CLIPS code will reach the state "A done" as shown in Figure 13.

```
PATH through STDWP

A-start;
B-start (branch B-1); D-start (branch D-1); * branch D-1 failure *
D-start (branch D-2); H-state; D-done;
E-state; B-done;
C-start (branch C-1); D-start (branch D-1); * branch D-1 failure *
D-start (branch D-2); H-state; D-done;
E-state; * branch C-1 failure *
C-start (branch C-2); D-start (branch D-1); * branch D-1 failure *
D-start (branch D-2); H-state; D-done;
* branch C-2 failure *
C-start (branch C-3); E-state; C-done;
D-start (branch D-1); * branch D-1 failure *
D-start (branch D-2); H-state; D-done;
A-done

* Successful termination *
```

Figure 13: Output of first test case

In the second test case, "fact E" and "fact F" are provided for the CLIPS code. These two states are not sufficient to furnish the STDWP to reach state "A done"; instead, it fails. This behavior can be observed in Figure 14 which shows the output of a sample run with "fact E" and "fact F".

```
PATH through STDWP

A-start;
B-start (branch B-1); D-start (branch D-1); * branch D-1 failure *
D-start (branch D-2); * No D-branch successful *
* branch B-1 failure *
B-start (branch B-2); F-state; B-done;
C-start (branch C-1); D-start (branch D-1); * branch D-1 failure *
D-start (branch D-2); * No D-branch successful *
* branch C-1 failure *
C-start (branch C-2); D-start (branch D-1); * branch D-1 failure *
D-start (branch D-2); * No D-branch successful *
* branch C-2 failure *
C-start (branch C-3); E-state; C-done;
D-start (branch D-1); * branch D-1 failure *
D-start (branch D-2); * No D-branch successful *

* Premature Termination, no success! *
```

Figure 14: Output of second test case

3. Assessment of the Behavior

The CLIPS program was tested for any conceivable scenario and proved the correct behavior by producing the expected sequence of states in the STDWP. This sequence is equivalent to the goal activation achieved by the original Prolog code shown in Figure 9.

Apparently, this behavior is satisfactory - but only for a problems of the given kind. The reason, why it behaves correctly is, that the number of branches in the "OR-Relationships", that occur in the "second level", is limited [4]; i.e., rule "D" consists of two branches, each of which holding just one state, namely "G" and "H". After the failure of the branch with salience "0" only one choice is left, namely the other branch with salience "-1".

If there were more than two branches with multiple states, then the implementation shown in the previous section would quickly reach its boundaries. In this case, the system should encounter a newly competing situation, that is, a competition between all branches which have not failed yet. And besides that, it must be assured that the recently failed branch does not participate in any other branch competitions.

Combined with this study, the attempt has been made to implement "OR-OR-Relationships" by following the translation pattern in the previous section. But it turned out, that the program actually conducts a sequential run through the eligible states regardless to the given saliences. And this seems to be inevitable, due to rule firing control and salience management. Furthermore, the number of traffic rules increased enormously, such that the portion of traffic, clean-up, and control rules shared almost thirty percent of the entire program.

These crucial drawbacks were the motivation for a different implementation approach. It is being presented in the following section.

---

4. An "OR-Relationship" which is itself a portion of another "OR-Relationship's" construct may be called "OR-OR-Relationship" in the following.

D. A PROBLEM WITH "OR-OR-RELATIONSHIPS"

1. Introduction of the Problem

The system to be discussed in this section is shown in its Prolog version in Figure 15, its corresponding STDWP in Figure 16, and accordingly, the CLIPS code may be found in APPENDIX B. The AND/OR Goal Tree is neglected in this context, since it is not important for the further discussion.

```
A :- B, B1.
A :- C, C1.
A :- D, D1.
B :- X, X1, X2.
B :- Y.
B :- Z, Z1.
Y :- ALPHA, ALPHA1.
Y :- BETA, BETA1.
Y :- GAMMA, GAMMA1.
```

Figure 15: A simple Prolog program

Four levels of relationships are joint together in this system. On the first level, an "OR-Relationship" is given by three rules of "A", each of which containing an "AND-Relationship" that is to be considered on the second level. A closer look to the right side of the Prolog rules makes clear that only the first rule with goal "A" contains a further level, that is rule "A:- B, B1." by the subgoal "B". The goal "B", on the other hand, can be reached by three alternative rules. Again, this is considered to be an "OR-Relationship", but now on the third level. The first and the third rule of "B" contain "AND-Relationships", where the second rule is satisfied by subgoal "Y", which leads also to an "OR-Relationship", namely to the three alternative rules of "Y". With this construct, the fourth level is finally reached. To be consistent and complete, a fifth level will be entered by the "AND-Relationships" of the right hand sides of every "Y" rule.

Such a joint construct may be called an "OR-AND-OR-OR-Relationship". The fifth level, given by the "AND-Relationships" for the alternative goals "Y", can be

neglected. This is, because an extra "AND-Relationship" does not require additional traffic rules; more explanations will be given in the next section.

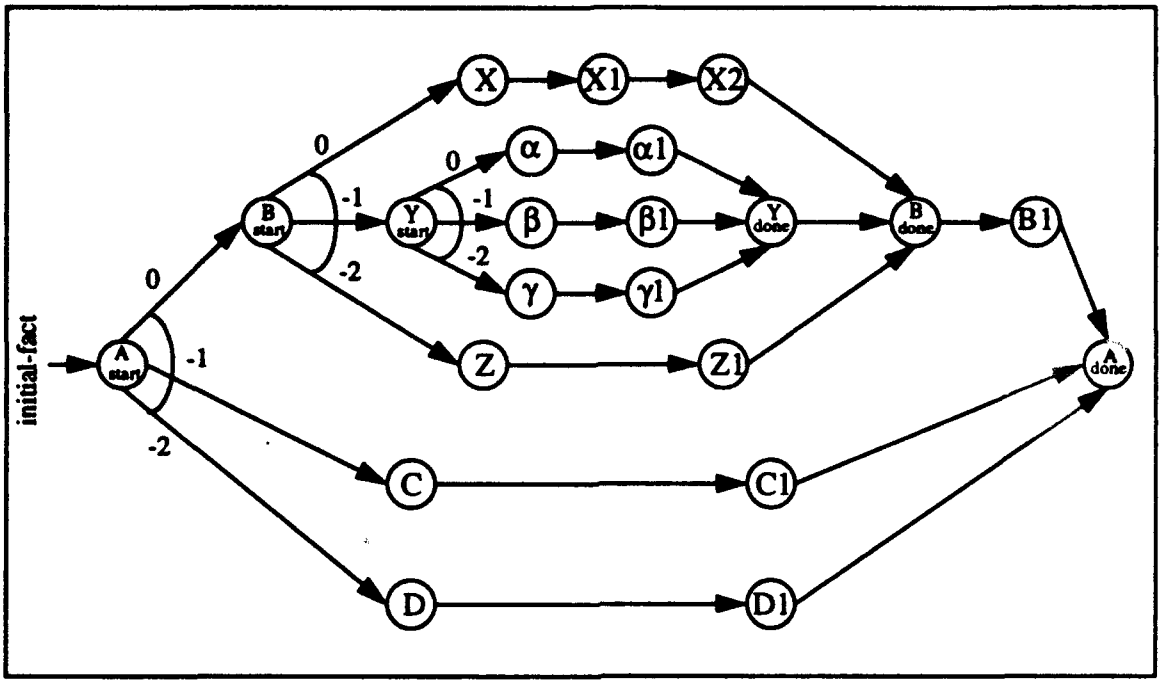


Figure 16: STDWP corresponding to the Prolog code in Figure 15

The derived STDWP is shown in Figure 16. It consists of three "A-branches" leaving the branching state "A start", three "B-branches" starting from "B start", and three "Y-branches" leaving state "Y start", each of which holding multiple states in this branch. Following the definitions of a STDWP, the upper branches have higher priority over lower branches, indicated by numbers (in the diagram of Figure 16 drawn close to the transition arrows). The transition conditions are named after the states. I.e., state "X" requires the transition condition "(condition X)", "B start" may only be reached, if "(condition B)" is available. These transition conditions are provided in the same fashion in the CLIPS code which may be found in APPENDX B. To keep the drawing simple, the transition conditions are not shown in Figure 16, as well as the backtracking paths.

2. Implementation

The most difficult part of the implementation of the STDWP displayed in Figure 16 in CLIPS was twofold:

(1) After a failure of a transition, the correct backtracking to the closest previous branching state needs to be assured. In an "OR-OR-Relationship", there are two branching states which are both acting as closest previous branching states. If a branch fails in the second level, then backtracking to the closest previous branching state of the first level has to be avoided. This means, that the STDWP's construct outside of the current "OR-Relationship" has to be "hidden" of any possible access. To provide this feature, a "status" field with a "hide" and "reveal" slot is added to each template defining an "OR-Relationship" in CLIPS (an example will be given shortly).

(2) After backtracking occurred, the failed branch has to be excluded from any new competition among the branches which leave this closest previous branching state. This feature is ensured by retracting the transition condition which enabled the system to try the branch which turned out to fail; i.e., if the "X-branch" fails in the "B" construct, then the system is backtracking to "B-start". At the same time, "(condition X)" is retracted from the database so that the "X-branch" cannot compete anymore with the remaining *and* eligible branches "Y-branch" and "Z-branch".

Besides the "hide" and "reveal" slots, two more slots have been added to the "status" field of the CLIPS templates, namely "failed" and "succeeded". The slot "failed" is necessary to make sure that the failed partial construct of the STDWP is not eligible anymore. This will be the case when the system has tried all eligible transitions in this portion of the STDWP and has not succeed. The slot "succeeded" is used for the other, the positive case and for "OR-AND-Relationship" constructs. State transitions, following an "OR-Relationship" after a "done" state have to be prevented from backtracking to the closest previous branching state of the successful "OR-Relationship". In the case of a transition failure in the "AND-Relationship", the system must backtrack to the *prior* closest

previous branching state. I.e., in the system introduced in this section, state "B1" follows state "B-done" which is the final state of the "OR-Relationship" construct for "B". If the transition to state "B1" fails, then the responsible closest previous branching state will be state "A-start", and not "B-start". This confusion is avoided by the "succeeded" slot in "deftemplate B", such that the system cannot start any transitions within the "B" construct again.

As an example, the CLIPS's *deftemplate* construct for the "OR-Relationship" of goal "A" is shown in the following to complete the previous explanations (see also APPENDIX B). Note that a *deftemplate A-branches* as it was introduced in section IV.C. is not required any more.

```
(deftemplate A
  (field state
     (type SYMBOL)
     (allowed-symbols   start B B1 C C1 D D1 done))
  (field status
     (type SYMBOL)
     (allowed-symbols    reveal hide succeeded failed)))
```

Backtracking comes into play when a transition failed to be executed. In order to ensure that the backtracking path reaches the responsible closest previous branching state, every branch in the STDWP has to be furnished with a traffic rule; i.e., the traffic rule for the "X-branch" within the "B" construct is "X-failure". This rule controls the backtracking traffic from all states in this branch to state "B-start" which is the responsible state. At this point, the traffic rule "B-failure" comes into action. It activates all remaining and eligible transitions for another competition. As expected, the state transition with the highest priority will be executed then.

3. Testing the Behavior

The system as it was introduced above is started by the CLIPS-internal "initial-fact" which can be accomplished automatically by the "(reset)" command.[5] From here, the

5. For complete and further syntax it is referred to [Ref. 11].

system is going through the states that can be reached by the given transition conditions. These conditions may be added to the CLIPS's database either by a batch file or by asserting the conditions manually.

Two test cases may illustrate the system's behavior implemented by the code as shown in APPENDIX B. They are shown in Figure 17 and 18.

The output shows the path through the STDWP, including the states reached by backtracking. These states are followed by a *... failure * message since a failure is the only cause for a backtracking path.

In the first case, the following conditional scenario is available in the CLIPS database: (condition A), (condition B), (condition Y), (condition Z), (condition ALPHA), (condition GAMMA), (condition D), and (condition D1). The program's output is shown in Figure 17.

```
PATH through STDWP

A start;
B-state; Y-state;
ALPHA-state; * branch Y-1 failure *; Y-state;
GAMMA-state; * branch Y-3 failure *; Y-state;
* branch B-2 failure *; B-state;
Z-state; * branch B-3 failure *; B-state
* branch A-1 failure *; A-state
D-state; D1-state;
A-done;

* Successful termination *
```

Figure 17: Output of the first test case

After starting the system and reaching state "B-start", only two state transitions are activated due to (condition Y) and (condition Z). The transition to "Y-start" is executed because of its higher priority. Again, just two transition activations are eligible, caused by (condition ALPHA) and (condition GAMMA). Both branches will fail since none of the further transition conditions are met. Thus, in both cases the system is backtracking to state "Y-start" causing itself backtracking to state "B-start". The only eligible, activated, and

executed transition to state "Z" is being made, but is backtracking to state "B-start" afterwards since (condition Z1) is not available. This causes backtracking to state "A-start" because all branches in the "B" construct failed. From this point, the final state transitions to state "A-done" are conducted via state "D" and "D1".

In the second test case, the scenario differs from the first one by the following conditions: Instead of (condition GAMMA), the database includes (condition BETA) and (condition BETA1); additionally (condition B1). The output looks as shown in Figure 18.

```
PATH through STDWP

A start;
B-state; Y-state;
ALPHA-state; * branch Y-1 failure *; Y-state;
BETA-state; BETA1-state;
Y-done; B-done; B1-state;
A-done;

* Successful termination *
```

Figure 18: Output of the second test case

In this case, the correct behavior of the "OR-AND-OR-OR-Relationship" of the given STDWP was tested. The conducted path is shown in Figure 18 and may not require further explanations.

Almost all conceivable scenarios have been tested. The system as it is implemented by CLIPS in APPENDIX B behaved in the same manner as it would do in the Prolog version. This proves empirically the logical and the behavioral equivalence between both implementations.

E. THE STRATEGIC LEVEL OF THE RATIONAL BEHAVIOR MODEL

1. Introduction of the Rational Behavior Model

The Rational Behavior Model (RBM) is a tri-level software control architecture for autonomous vehicles. It is composed of a strategic, a tactical, and an execution level. The strategic level mainly consists of goals which are to be achieved by an autonomous robot vehicle under control of RBM. These goals have to be achieved in a certain sequence depending on external (= world) and internal (= vehicle) events to accomplish a given mission. In order to exhibit coherently intelligent behavior globally, some means of understanding and memorizing external and internal events are required by the control architecture. Otherwise, globally intelligent behaviors cannot be achieved by an autonomous robot [Ref. 13]. In RBM, the tasks of understanding and memorizing are performed by the tactical level [Ref. 14]. Thus, the strategic level is free from processing or memorizing external or internal events directly. This arrangement reduces the complexity of the strategic level by a major portion; in particular, the strategic level does not need any state memory associated with the external world or the control of the robot. Instead, it queries the tactical level, whenever it is appropriate. Inferring the returned values of the predicate queries, the strategic level determines next which goal has to be activated at this moment. Specifically, a goal activation means a behavior activation request to the tactical level. Once the requested behavior is completed by the tactical level in conjunction with the execution level, then the goal is achieved.[6] In this way, goals in the strategic level directly affect the behavior of the physical robot under control.

Since the strategic level purely operates by predicate queries and goals, a rule-based programming language is the best choice for implementation. Fundamentally, there are two ways of implementing rule-based programming, these are backward chaining and

---

6. The execution level is the lowest level in RBM. It directly handles the vehicle's hardware. Thus, the execution level is mainly composed of various conventional control loops and device drivers.

forward chaining. Both are equivalent in the sense of pure logical power. They are merely different in syntactical perspective. However, one implementation with one chaining direction may be more efficient than the other. In spite of this well known fact, a translation from one chaining implementation of the strategic level in RBM to the other is not trivial. The reason is due to the fact, that a sequence of logic resolution steps is not unique.[7] In other words, two different chaining implementations may produce two different paths of goal activation and execution during chaining. This means that the two implementations could make the vehicle behave totally differently even though they are logically equivalent. However, in robot control, the sequence of behavior matters. For example, a final goal cannot be achieved without achieving an intermediate goal. In other words, two different implementations for the strategic level of RBM have to be able to produce an identical sequence of goal activations. This is an unique constraint of equivalence of the strategic level of RBM.

The first instance of the strategic level of RBM [Ref. 14] was implemented in Prolog which was used as a backward chaining rule-based programming language. This led to another implementation in Prolog [Ref. 15]. Both implementations have been established for the Adaptive Suspension Vehicle (ASV) [Ref. 16]. Later, the strategic level of the Naval Postgraduate School Autonomous Underwater Vehicle Model II (NPS AUV) [Ref. 9] was also implemented in Prolog (see APPENDIX C).

It is not surprising that Prolog served as programming language for all these implementations. The reason is that goals in robot vehicle control tend to be related in hierarchical order. For example, "execute-auv-mission" is a top goal which can be immediately decomposed into subgoals corresponding to submissions of the entire mission. A hierarchical organization composed of a top goal, some sub-goals, some sub-sub-goals, etc. can be easily expressed in Prolog because of its declarative nature. Moreover, Prolog's inference engine with its depth-first-search strategy forces the rule firing and conflict

7. Chaining is known to be a special case of resolution.

resolution to follow the textual order of the Prolog rules. Therefore, multiple subgoals that participate in an "OR-Relationship" among each other can be easily prioritized by the textual order of the Prolog rules. In this way, the strategic level can be "naturally" declared in Prolog, and the desired sequence of goal activations is evidently produced by Prolog's inference engine.

The forward chaining implementation of the strategic level, however, has been delayed until STDWP was introduced. The extra constraint needed to maintain equivalence of two opposite chaining implementations, an equal sequence of goal activations, was hard to be satisfied in a context of a complex real world problem. A simple syntactical change in the translation from backward chaining to forward chaining is guaranteed to produce a non-functioning strategic level since it will cause a different sequence of goal activations. Fortunately, two sets of equivalences are found in the context of the strategic level of RBM. One is the equivalence of Prolog rules and AND/OR goal trees, and the other is the equivalence of CLIPS forward chaining rules and STDWP.[8]

The graphical equivalence between an AND/OR goal tree and a STDWP has been shown in [Ref. 9]; the logical and behavioral equivalence of corresponding implementations in Prolog vs. CLIPS was empirically proven in the previous chapter.

To support these findings, both graphical tools and either implementation is being applied to the strategic level of RBM for the NPS AUV in the following section.

2. The Implementation of the Strategic Level

The RBM's strategic level was initially implemented in Prolog because of Prolog's hierarchical order and its determination of sequence of primitive goals. The source code of this implementation is presented in APPENDIX C. The corresponding AND/OR goal tree may be found in APPENDIX D and the STDWP in APPENDIX E. The complexity of either graphical representation is considered to be not too big so that it

8. The equivalence relationship is empirically known to be correct. Formal proof is not available yet.

becomes impractical. Both graphs support enough readability and may be used side by side since they provide equivalent logic and behavior.

The implementation in CLIPS was derived from Prolog's source code and was first carried out "manually". Although the fundamental and theoretical background was still not exactly defined by the time of translation, a translation pattern was already found and used.

The CLIPS code as it is presented in APPENDIX C is an enhanced version of the initial program. It differs slightly from the structures introduced and discussed in the previous sections since the translation was completed based on preliminary findings. Nevertheless, a syntactically clean version that accomplishes a "clean" translation according to the implementation structures introduced in the previous chapter is already in progress and will be reported later.

However, the equivalent sequence of the behavioral activation is empirically confirmed for both, the Prolog and the CLIPS implementations [Ref. 9]. That means, that either version of the strategic level, whether Prolog or CLIPS, produces exactly the same sequence of behaviors during mission control and execution when it controls the NPS AUV's simulator [Ref. 9]. This observation was also made when the CLIPS code was used in CLIPS/Ada for the tactical level's software testing in [Ref. 14].

V. SUMMARY AND CONCLUSIONS

STDWP's add a new dimension to conventional STD's by permitting non-disjoint state transition conditions to multiple successor states. The non-determinism introduced by this extension is resolved by path priorities. Thus, although several state transitions are eligible, only one state transition with highest path priority will be selected to carry out a state transition to the next state. With this extension of a conventional STD, a STDWP is ready to be applied to a new class of problems; i.e., for graphical representation of forward chaining production rules for the strategic level of the RBM.

Before the STDWP was introduced, the translation of the backward chaining implementation of the strategic level in a forward chaining implementation was not possible. There were even doubts and confusion about such a translation among researchers at the Naval Postgraduate School because of unique equivalence requirements in the strategic level of the RBM; i.e., two implementations have to be not only logically equivalent, but also identical in the sequence of proving steps. These requirements led to a new class of equivalence of logic.

If an equivalent forward chaining implementation had not been found, there would have been a serious chance that the unique equivalent requirements prevented the strategic level from being qualified as a subset of a rule-based system. However, an equivalent forward chaining implementation of the strategic level was successfully hand-coded utilizing STDWP. Most likely, there always exists two equivalent chaining implementations in each direction because of the found algorithm which allows bi-directional translation[1] [Ref. 7]. However, a formal proof for existence of two opposite chaining implementation is not researched yet. A formal proof will automatically provide another evidence for that a strategic level of RBM is a true subset of a conventional rule-based reasoning system which only concerns about logical equivalence. But this is also another future research topic.

1. The correctness and completeness of this algorithm is not proved yet.

Another discovery from the presented work is complexity, compared between two implementations. Backward chaining implementation is always simpler than the corresponding forward chaining implementation. For example, comparing the number of nodes in the AND/OR goal tree in Figure10 with the number of states in the STDWP in Figure 12 results in a difference of 7, that is 21 nodes vs. 28 states. A similar discovery is reported in a different context [Ref. 17].

The previous discovery can be summarized as follows: an AND/OR goal tree is viewed as a goal decomposition diagram, and a STDWP is considered as a goal execution diagram with explicit goal execution details. The former can be seen as a disassembly diagram, the latter can be interpreted as an assembly diagram.[2] In a computational aspect, the former demands more than the latter implementation. Actually, faster execution was observed in the CLIPS implementation even though the size of the code is about ten times bigger than the equivalent Prolog code. This observation supports the role of the AND/OR goal tree's depth first traverser. The depth first traverser interprets the AND/OR goal tree, and it produces a sequence of goal activations depending on a situation on fly whereas the STDWP needs little computation because it can be seen as a collection of all possible execution paths. Whether or not the translation problem is the same class of problems as that presented by [Ref. 17] is another future research topic.

The final issue is whether an equivalent STDWP is a minimum representation or not. If this is not the case, then it has to be researched how to make a minimum STDWP. This issue seems not to be discussed in Homem de Mello's research. [Ref. 17]

Once the above mentioned research questions are fully answered, then those findings will immediately lead to an economy of mission description in general (mission description for either AUV or human military unit); i.e., an AND/OR goal tree style mission description vs. a STDWP style mission description. The observation of current practices among people in general reveals two side stories. If a mission is well defined and well

2. Loosely speaking, disassembling a device is an easier task than assembling it.

understood, then an AND/OR goal tree style mission description is common. On the other hand, the opposite is true. A military AUV mission and a scientific discovering AUV mission show two examples, one for each side.

The research presented in this study is just opening up whole new issues rather than reducing them by solving existing problems. This recognition might be an herald of an extremely interesting research area.

APPENDIX A. CLIPS IMPLEMENTATION OF THE EXAMPLE IN SECTION IV.C.

```
;*****************************************************************************
;*   Title       : Example for Thesis, Section IV.C.
;*   Name        : example2.2
;*   Version     : 2.2
;*   Author      : Thomas Scholz
;*   Date        : 18 May 1993
;*   Revised     : 15 September 1993
;*   System      : UNIX Sun/Solbourne
;*   Compiler    : Clips 5.1
;*****************************************************************************

(deftemplate rule-A
             (field state
                        (type SYMBOL)
                        (allowed-symbols        start  B  C  D  done)))

(deftemplate rule-B
             (field state
                        (type SYMBOL)
                        (allowed-symbols        start  D  E  F  done)))

(deftemplate B-branches
             (field branch
                        (type SYMBOL)
                        (allowed-symbols        B-1  B-2)))

(deftemplate rule-C
             (field state
                        (type SYMBOL)
                        (allowed-symbols        start  D  E  F  done)))

(deftemplate C-branches
             (field branch
                        (type SYMBOL)
                        (allowed-symbols        C-1  C-2  C-3)))

(deftemplate rule-D
             (field state
                        (type SYMBOL)
                        (allowed-symbols        start   G  H  done)))

(deftemplate D-branches
             (field branch
                        (type SYMBOL)
                        (allowed-symbols        D-1  D-2)))
```

```
;-------------------------------------------------------------Prolog------
; A :- B,C,D.
;-------------------------------------------------------------------------

(defrule start-program
            ?x <- (initial-fact)
=>
            (retract ?x)
            (assert (rule-A (state B)))
            (printout t crlf "PATH through STDWP:" crlf crlf "A-start;"
             crlf))

(defrule rule-A-state-B
            ?x <- (rule-A (state B))
=>
            (retract ?x)
            (assert (rule-B (state start))))

(defrule rule-A-state-C
            (declare (salience -10))
            ?x <- (rule-B (state done))
=>
            (retract ?x)
            (assert (rule-C (state start))))

(defrule rule-A-state-D
            (declare (salience -20))
            ?x <- (rule-C (state done))
=>
            (retract ?x)
            (assert (rule-D (state start))))

(defrule rule-A-state-done
            (declare (salience -30))
            ?x <- (rule-D (state done))
=>
            (retract ?x)
            (assert (rule-A (state done)))
            (printout t "A-done" crlf crlf "* Successful termination *"
             crlf crlf))

(defrule total-failure
            (declare (salience -100))
            (not (rule-A (state done)))
=>
            (printout t  crlf "* Premature Termination, no success! *"
             crlf crlf))
```

```
;-------------------------------------------------------------Prolog------
; B :- D,E.
; B :- F.
;-------------------------------------------------------------------------

;-----------------------------------------------Traffic rules------
(defrule B-branch-1-start
            ?x <- (rule-B (state start))
=>
            (retract ?x)
            (assert (B-branches (branch B-1))))

(defrule B-branch-2-start
            (declare (salience -1))
            ?x <- (B-branches (branch B-1))
            ?y <- (rule-B (state ?))
=>
            (retract ?x)
            (retract ?y)
            (assert (B-branches (branch B-2))))

(defrule B-branches-clean-up
            (declare (salience -2))
            ?x <- (B-branches (branch B-2))
            ?y <- (rule-B (state ?))
=>
            (retract ?x)
            (retract ?y)
            (printout t "* No B-branch successful ** crlf))
;-------------------------------------------------------------------------

;-----------------------------------------------------B-1-branch----
(defrule B-branch-1-state-D
            (B-branches (branch B-1))
=>
            (assert (rule-B (state D)))
            (assert (rule-D (state start)))
            (printout t "B-start (branch B-1); "))

(defrule B-branch-1-state-E
            (B-branches (branch B-1))
            ?x <- (rule-B (state D))
            ?y <- (rule-D (state done))
=>
            (retract ?x)
            (retract ?y)
            (assert (rule-B (state E))))

; -------------------------------------------------------------------------
```

```
; If "(fact E)" exists, "(rule-B (state done))" will follow immediately.
;-------------------------------------------------------------------
(defrule B-branch-1-state-done
            ?x <- (B-branches (branch B-1))
            ?y <- (rule-B (state E))
            (fact E)
=>
            (retract ?x)
            (retract ?y)
            (assert (rule-B (state done)))
            (printout t "E-state; B-done; " crlf))
;-------------------------------------------------------------------

;-----------------------------------------------------B-2-branch---
(defrule B-branch-2-state-F
            (B-branches (branch B-2))
=>
            (assert (rule-B (state F)))
            (printout t "* branch B-1 failure ** crlf
             "B-start (branch B-2); "))

;-------------------------------------------------------------------
; If "(fact F)" exists, "(rule-B (state done))" will follow immediately.
;-------------------------------------------------------------------
(defrule B-branch-2-state-done
            ?x <- (B-branches (branch B-2))
            ?y <- (rule-B (state F))
            (fact F)
=>
            (retract ?x)
            (retract ?y)
            (assert (rule-B (state done)))
            (printout t "F-state; B-done; " crlf))

;-------------------------------------------------------------------


;-----------------------------------------------------------Prolog-----
; C :- D, E, F.
; C :- D, F.
; C :- E.
;-------------------------------------------------------------------

;-----------------------------------------------------Traffic rules--
(defrule C-branch-1-start
            ?x <- (rule-C (state start))
=>
            (retract ?x)
            (assert (C-branches (branch C-1))))

(defrule C-branch-2-start
```

```
            (declare (salience -1))
            ?x <- (C-branches (branch C-1))
            ?y <- (rule-C0 (state ?))
=>
            (retract ?x)
            (retract ?y)
            (assert (C-branches (branch C-2))))

(defrule C-branch-3-start
            (declare (salience -2))
            ?x <- (C-branches (branch C-2))
            ?y <- (rule-C (state ?))
=>
            (retract ?x)
            (retract ?y)
            (assert (C-branches (branch C-3))))

(defrule C-branches-clean-up
            (declare (salience -2))
            ?x <- (C-branches (branch C-3))
            ?y <- (rule-C (state ?))
=>
            (retract ?x)
            (retract ?y)
            (printout t "* No C-branch successful *" crlf))
;-----------------------------------------------------------------

;-------------------------------------------------C-1 branch------
(defrule C-branch-1-state-D
            (C-branches (branch C-1))
=>
            (assert (rule-C (state D)))
            (assert (rule-D (state start)))
            (printout t "C-start (branch C-1); "))

(defrule C-branch-1-state-E
            (C-branches (branch C-1))
            ?x <- (rule-C (state D))
            ?y <- (rule-D (state done))
=>
            (retract ?x)
            (retract ?y)
            (assert (rule-C (state E))))

;-----------------------------------------------------------------
; If "(fact E)" exists, "(rule-C (state F))" will follow immediately.
;-----------------------------------------------------------------
(defrule C-branch-1-state-F
            (C-branches (branch C-1))
            ?x <- (rule-C (state E))
            (fact E)
```

```
=>
            (retract ?x)
            (assert (rule-C (state F)))
            (printout t "E-state; "))

;-------------------------------------------------------------------
; If "(fact F)" exists, "(rule-C (state done))" will follow immediately.
;-------------------------------------------------------------------
(defrule C-branch-1-state-done
            ?x <- (C-branches (branch C-1))
            ?y <- (rule-C (state F))
            (fact F)
=>
            (retract ?x)
            (retract ?y)
            (assert (rule-C (state done)))
            (printout t "F-state; C-done; " crlf))
;-------------------------------------------------------------------

;-----------------------------------------------------C-2 branch------
(defrule C-branch-2-state-D
            (C-branches (branch C-2))
=>
            (assert (rule-C (state D)))
            (assert (rule-D (state start)))
            (printout t "* branch C-1 failure * " crlf
             "C-start (branch C-2); "))

(defrule C-branch-2-state-F
            (C-branches (branch C-2))
            ?x <- (rule-C (state D))
            ?y <- (rule-D (state done))
=>
            (retract ?x)
            (retract ?y)
            (assert (rule-C (state F))))

;-------------------------------------------------------------------
; If "(fact F)" exists, "(rule-C (state done))" will follow immediately.
;-------------------------------------------------------------------
(defrule C-branch-2-state-done
            ?x <- (C-branches (branch C-2))
            ?y <- (rule-C (state F))
            (fact F)
=>
            (retract ?x)
            (retract ?y)
            (assert (rule-C (state done)))
            (printout t "F-state; C-done; " crlf))
;-------------------------------------------------------------------
```

```
;-------------------------------------------------------C-3 branch----
(defrule C-branch-3-state-E
            (C-branches (branch C-3))
=>
            (assert (rule-C (state E)))
            (printout t "* branch C-2 failure * " crlf
             "C-start (branch C-3); "))

;-------------------------------------------------------------------------
; If "(fact E)" exists, "(rule-C (state done))" will follow immediately.
;-------------------------------------------------------------------------
(defrule C-branch-3-state-done
            ?x <- (C-branches (branch C-3))
            ?y <- (rule-C (state E))
            (fact E)
=>
            (retract ?x)
            (retract ?y)
            (assert (rule-C (state done)))
            (printout t "E-state; C-done; " crlf))

;-------------------------------------------------------------------------


;-----------------------------------------------------------Prolog---- --
; D :- G.
; D :- H.
;-------------------------------------------------------------------------

;-----------------------------------------------Traffic rules------
(defrule D-branch-1-start
            ?x <- (rule-D (state start))
=>
            (retract ?x)
            (assert (D-branches (branch D-1))))

(defrule D-branch-2-start
            (declare (salience -1))
            ?x <- (D-branches (branch D-1))
            ?y <- (rule-D (state ?))
=>
            (retract ?x)
            (retract ?y)
            (assert (D-branches (branch D-2))))

(defrule D-branches-clean-up
            (declare (salience -1))
            ?x <- (D-branches (branch D-2))
            ?y <- (rule-D (state ?))
=>
            (retract ?x)
```

```
            (retract ?y)
            (printout t "* No D-branch successful *" crlf))
;-----------------------------------------------------------------

;-------------------------------------------------D-1-branch----
(defrule D-branch-1-state-G
            (D-branches (branch D-1))
=>
            (assert (rule-D (state G)))
            (printout t "D-start (branch D-1); "))

;-------------------------------------------------------------------
; If "(fact G)" exists, "(rule-D (state done))" will follow immediately.
;-------------------------------------------------------------------
(defrule D-branch-1-state-done
            ?x <- (D-branches (branch D-1))
            ?y <- (rule-D (state G))
            (fact G)
=>
            (retract ?x)
            (retract ?y)
            (assert (rule-D (state done)))
            (printout t "G-state; D-done;" crlf))
;-----------------------------------------------------------------

;-------------------------------------------------D-2-branch----
(defrule D-branch-2-state-H
            (D-branches (branch D-2))
=>
            (assert (rule-D (state H)))
            (printout t "* branch D-1 failure * " crlf
             "D-start (branch D-2); "))

;-------------------------------------------------------------------
; If "(fact H)" exists, "(rule-D (state done))" will follow immediately.
;-------------------------------------------------------------------
(defrule D-branch-2-state-done
            ?x <- (D-branches (branch D-2))
            ?y <- (rule-D (state H))
            (fact H)
=>
            (retract ?x)
            (retract ?y)
            (assert (rule-D (state done)))
        (printout t "H-state; D-done;" crlf))
-----------------------------------------------------------------


;*****************************************************************
```

APPENDIX B. CLIPS IMPLEMENTATION OF THE EXAMPLE IN SECTION IV.D.

```
;*************************************************************************
;*
;*   Title        : Example for Thesis, section IV.D.
;*   Name         : new1.5
;*   Version      : 1.5
;*   Author       : Thomas Scholz
;*   Date         : 9 September 1993
;*   Revised      : 15 September 1993
;*   System       : Sun/Solbourne UNIX
;*   Compiler     : Clips 6.0
;*   Prolog-Code  :
;*                        A :- B,B1.
;*                        A :- C,C1.
;*                        A :- D,D1.
;*                        B :- X,X1,X2.
;*                        B :- Y.
;*                        B :- Z,Z1.
;*                        Y :- ALPHA, ALPHA1.
;*                        Y :- BETA,BETA1.
;*                        Y :- GAMMA,GAMMA1.
;*
;*   Remarks     : No more sequential behavior. After a branch failure, all
;*                    remaining choices compete again.
;*                    Implementation of "OR-AND-OR-OR - Relationship".
;*************************************************************************

; ----------------------------Templates for OR-Relationships------------
(deftemplate A
        (field state
                (type SYMBOL)
               (allowed-symbols        start  B  B1  C  C1  D  D1  done))
        (field status
                (type SYMBOL)
                (allowed-symbols        reveal  hide succeeded failed)
                (default                reveal)))

(deftemplate B
        (field state
                (type SYMBOL)
               (allowed-symbols        start  X  X1  X2  Y  Z  Z1  done))
        (field status
                (type SYMBOL)
                (allowed-symbols        reveal  hide succeeded failed)
                (default                reveal)))
```

```
(deftemplate Y
        (field state
                (type SYMBOL)
                (allowed-symbols        start  ALPHA  ALPHA1  BETA  BETA1
                                        GAMMA  GAMMA1  done))
        (field status
                (type SYMBOL)
                (allowed-symbols        reveal  hide succeeded failed)
                (default                reveal)))
; ---------------------------------------------------------------------
; ----------------------------------------------------start program-----
(defrule start-program
        ?x <- (initial-fact)
        (condition A)
=>
        (retract ?x)
        (assert (A (state start)))
        (printout t crlf "PATH through STDWP" crlf crlf "A start;" crlf))

(defrule A-failure
        (declare (salience -10))
        ?z <- (initial-fact)
=>
        (retract ?z)
        (printout t "* Total failure *. System halted." crlf crlf))
; ---------------------------------------------------------------------
; ------------------------------------------A-1 branch-------------
(defrule A-start-B
        ?x <- (A (state start))
        (condition B)
=>
        (modify ?x (state B)(status hide))
        (assert (B (state start)))
        (printout t "B-state; "))

(defrule B-failure
        (declare (salience -10))
        ?x <- (A (state B)(status hide))
        ?y <- (B (state ?n)(status reveal))
        ?z <- (condition B)
=>
        (retract ?z)
        (modify ?x (state start)(status reveal))
        (modify ?y (state ?n)(status failed))
        (printout t "* branch A-1 failure *; A-state" crlf))

; ------------------------------------------------B-1 branch----
(defrule B-start-X
        ?x <- (B (state start)(status reveal))
        (A (state B)(status hide))
        (condition X)
```

```
=>
        (modify ?x (state X)(status hide))
;       (modify ?y (status hide))
        (printout t "X-state; "))

(defrule X-failure
        (declare (salience -10))
        ?x <- (B (state ?n)(status hide))
        ?y <- (condition X)
=>
        (retract ?y)
        (modify ?x (state start)(status reveal))
        (printout t "* branch B-1 failure *; B-state" crlf))

(defrule B-start-X1
        ?x <- (B (state X))
        (condition X1)
=>
        (modify ?x (state X1))
        (printout t "X1-state; "))

(defrule B-start-X2
        ?x <- (B (state X1))
        (condition X2)
=>
        (modify ?x (state X2))
        (printout t "X2-state; "))

(defrule B-done1
        ?x <- (B (state X2))
        ?y <- (A (state B))
=>
        (modify ?x (state done)(status reveal))
        (modify ?y (status reveal))
        (printout t crlf "B-done; "))
; ------------------------------------------------end of B-1 branch-----
; ------------------------------------------------B-2 branch------------
(defrule B-start-Y
        (declare (salience -1))
        ?x <- (B (state start)(status reveal))
        (condition Y)
=>
        (modify ?x (state Y)(status hide))
        (assert (Y (state start)))
        (printout t "Y-state; " crlf))

(defrule Y-failure
        (declare (salience -10))
        ?x <- (B (state Y)(status hide))
        ?y <- (Y (state start)(status reveal))
        ?z <- (condition Y)
```

```
=>
        (retract ?z)
        (modify ?x (state start)(status reveal))
        (modify ?y (status failed))
        (printout t "* branch B-2 failure *; B-state; " crlf))

; ------------------------------------------------Y-1 branch----------
(defrule Y-start-ALPHA
        ?x <- (Y (state start)(status reveal))
        (condition ALPHA)
=>
        (modify ?x (state ALPHA))
        (printout t "ALPHA-state; "))

(defrule ALPHA-failure
        (declare (salience -10))
;       ?y <- (B (state Y)(status hide))
        ?x <- (Y (state ?n)(status reveal))
        ?z <- (condition ALPHA)
=>
        (retract ?z)
        (modify ?x (state start)(status reveal))
;       (modify ?y (status failed))
        (printout t "* branch Y-1 failure *; Y-state; " crlf))

(defrule Y-start-ALPHA1
        ?x <- (Y (state ALPHA))
        (condition ALPHA1)
=>
        (modify ?x (state ALPHA1))
        (printout t "ALPHA1-state; "))

(defrule Y-done-1
        ?x <- (Y (state ALPHA1))
        ?y <- (B (state Y))
;       ?z <- (A (status hide))
=>
        (modify ?x (state done)(status succeeded))
        (modify ?y (state done)(status reveal))
;       (modify ?z (status reveal))
        (printout t crlf "Y-done; B-done; "))
; ------------------------------------------------end of Y-1 branch---
; ------------------------------------------------Y-2 branch---------
(defrule Y-start-BETA
        (declare (salience -1))
        ?x <- (Y (state start)(status reveal))
        (condition BETA)
=>
        (modify ?x (state BETA))
        (printout t "BETA-state; "))
```

```
(defrule BETA-failure
        (declare (salience -10))
;       ?x <- (B (state Y)(status hide))
        ?x <- (Y (state ?n)(status reveal))
        ?z <- (condition BETA)
=>
        (retract ?z)
        (modify ?x (state start)(status reveal))
;       (modify ?y (status failed))
        (printout t "* branch Y-2 failure *; Y-state; " crlf))

(defrule Y-start-BETA1
        ?x <- (Y (state BETA))
        (condition BETA1)
=>
        (modify ?x (state BETA1))
        (printout t "BETA1-state; "))

(defrule Y-done-2
        ?x <- (Y (state BETA1))
        ?y <- (B (state Y))
;       ?z <- (A (status hide))
=>
        (modify ?x (state done)(status succeeded))
        (modify ?y (state done)(status reveal))
;       (modify ?z (status reveal))
        (printout t crlf "Y-done; B-done; "))
; ------------------------------------------------------end of Y-2 branch--
; ------------------------------------------------------Y-3 branch--------
(defrule Y-start-GAMMA
        (declare (salience -2))
        ?x <- (Y (state start))
        (condition GAMMA)
=>
        (modify ?x (state GAMMA))
        (printout t "GAMMA-state; "))

(defrule GAMMA-failure
        (declare (salience -10))
        ?x <- (Y (state ?n)(status reveal))
;       ?y <- (B (state Y)(status hide))
        ?z <- (condition GAMMA)
=>
        (retract ?z)
;       (modify ?x (status failed))
          odify ?x (state start)(status reveal))
        (printout t "* branch Y-3 failure *; Y-state; " crlf))

(defrule Y-start-GAMMA1
        ?x <- (Y (state GAMMA))
        (condition GAMMA1)
```

```
=>
        (modify ?x (state GAMMA1))
        (printout t "GAMMA1-state; "))

(defrule Y-done-3
        ?x <- (Y (state GAMMA1))
        ?y <- (B (state Y))
;       ?z <- (A (state B)(status hide))
=>
        (modify ?x (state done)(status succeeded))
        (modify ?y (state done)(status reveal))
;       (modify ?z (status reveal))
        (printout t crlf "Y-done; B-done; "))
; ------------------------------------------------end of Y-3 branch---
; ------------------------------------------------B-3 branch---------
(defrule B-start-Z
        (declare (salience -2))
        ?x <- (B (state start)(status reveal))
        (A (state B)(status hide))
        (condition Z)
=>
        (modify ?x (state Z)(status hide))
;       (modify ?y (status hide))
        (printout t "Z-state; "))

(defrule Z-failure
        (declare (salience -10))
        ?x <- (B (state ?n)(status hide))
        ?y <- (condition Z)
=>
        (retract ?y)
        (modify ?x (state start)(status reveal))
        (printout t "* branch B-3 failure *; B-state" crlf))

(defrule B-start-Z1
        ?x <- (B (state Z))
        (condition Z1)
=>
        (modify ?x (state Z1))
        (printout t "Z1-state; "))

(defrule B-done-2
        ?x <- (B (state Z1))
        ?y <- (A (state B))
=>
        (modify ?x (state done)(status reveal))
        (modify ?y (status reveal))
        (printout t crlf "B-done; A-done;" crlf crlf
         "* Successful termination *" crlf crlf))
; ------------------------------------------------end of B-3 branch-----
```

```
(defrule A-start-B1
        ?x <- (A (state B))
        (B (state done)(status reveal))
        (condition B1)
=>
        (modify ?x (state B1))
        (printout t "B1-state; "))

(defrule A-done-1
        ?x <- (A (state B1))
=>
        (modify ?x (state done)(status succeeded))
       (printout t crlf "A-done;" crlf crlf "* Successful termination *"
crlf crlf))

; ------------------------------------------------end of A-1 branch-----
; ------------------------------------------------A-2 branch--------
(defrule A-start-C
        (declare (salience -1))
        ?x <- (A (state start))
        (condition C)
=>
        (modify ?x (state C))
        (printout t "C-state; "))

(defrule C-failure
        (declare (salience -10))
        ?x <- (A (state ?)(status reveal))
        ?y <- (condition C)
=>
        (retract ?y)
        (modify ?x (state start)(status reveal))
        (printout t "* branch A-2 failure *; A-state; " crlf))

(defrule A-start-C1
        ?x <- (A (state C))
        (condition C1)
=>
        (modify ?x (state C1))
        (printout t "C1-state; "))

(defrule A-done-2
        ?x <- (A (state C1))
=>
        (modify ?x (state done)(status succeeded))
       (printout t crlf "A-done;" crlf crlf "* Successful termination *"
crlf crlf))
; ----------------------------------------------end of A-2 branch------
```

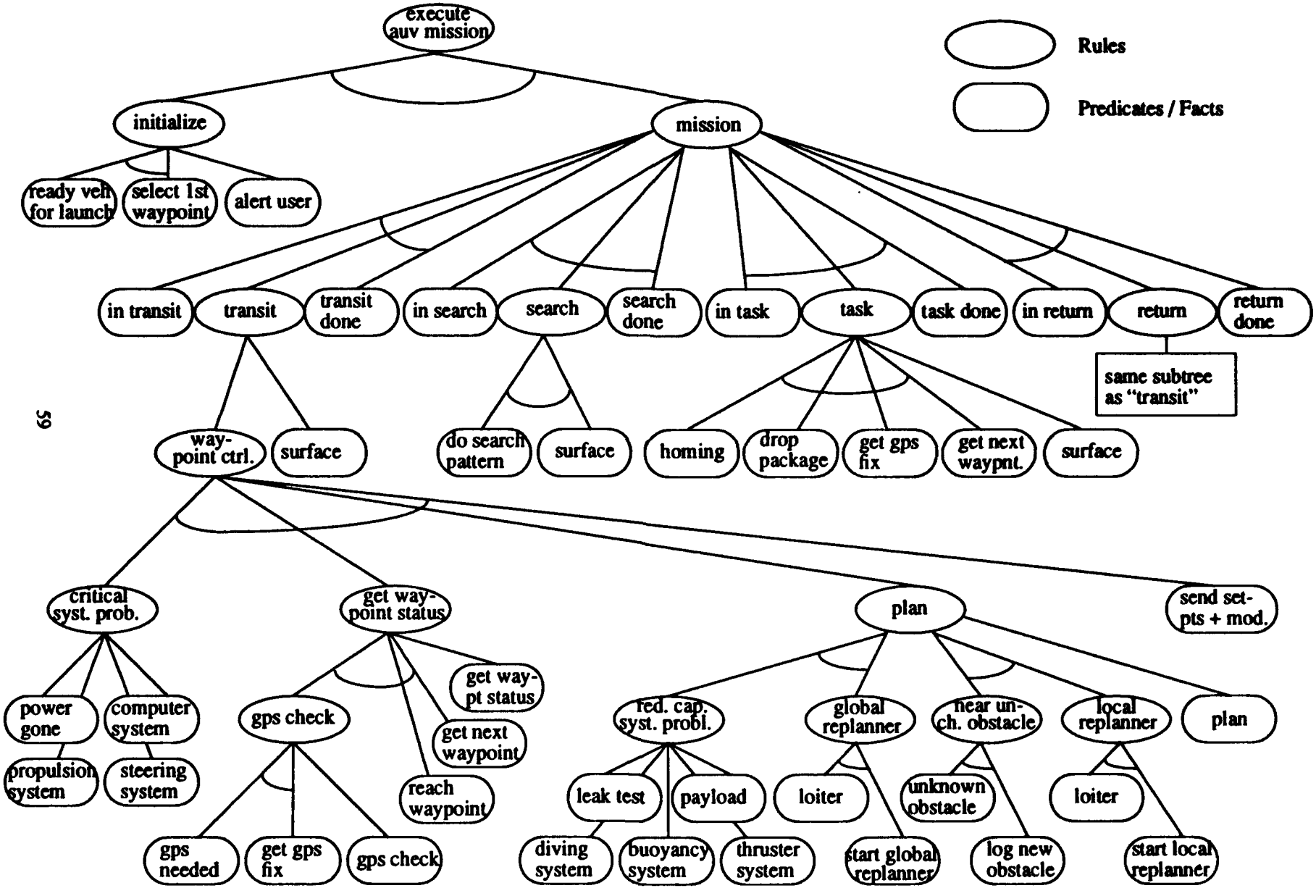
```
; ------------------------------------------------A-3 branch-------------
(defrule A-start-D
        (declare (salience -2))
        ?x <- (A (state start))
        (condition D)
=>
        (modify ?x (state D))
        (printout t "D-state; "))

(defrule D-failure
        (declare (salience -10))
        ?x <- (A (state ?)(status reveal))
        ?y <- (condition D)
=>
        (retract ?y)
        (modify ?x (status failed))
        (printout t "* branch A-3 failure *; A-state." crlf crlf)
      (printout t "* No A-branch successful *. System halted." crlf crlf))

(defrule A-start-D1
        ?x <- (A (state D))
        (condition D1)
=>
        (modify ?x (state D1))
        (printout t "D1-state; "))

(defrule A-done-3
        ?x <- (A (state D1))
=>
        (modify ?x (state done)(status succeeded))
       (printout t crlf "A-done;" crlf crlf "* Successful termination *"
crlf crlf))
; ------------------------------------------------end of A-3 branch------
;*****************************************************************************
```

APPENDIX C. PROLOG IMPLEMENTATION OF THE RBM'S STRATEGIC LEVEL

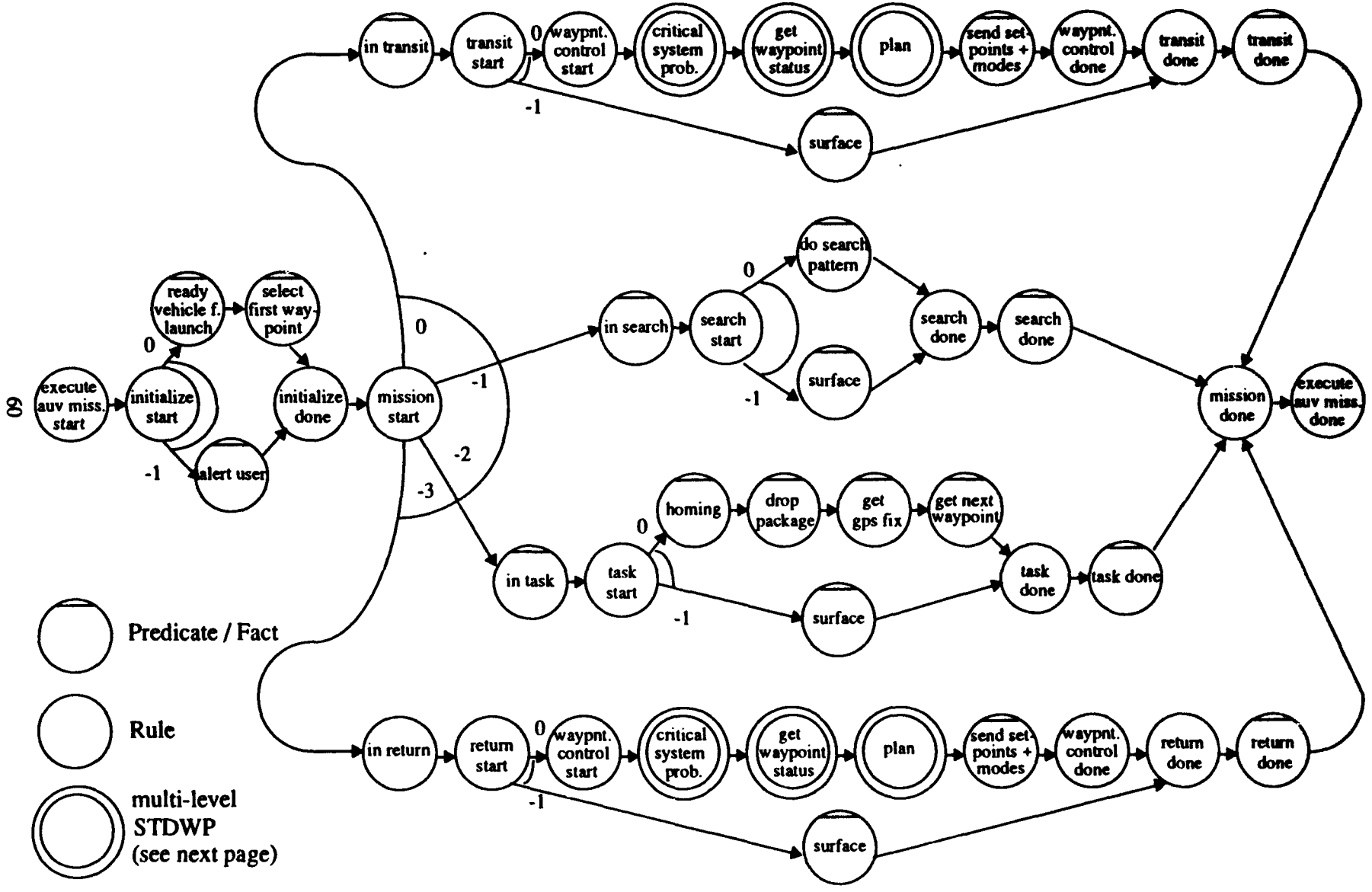
```
/*   Strategic Level for the RBM AUV Mission Controller/Coordinator
     by Byrnes, Kwak, Healey, Marco for use at Florida competition

      Version: 2.3  Oct 27, 1992                */

/* ---------------------MISSION SPECIFICATION ------------- */

initialize :- ready_veh_for_launch(ANS1),
        ANS1 == 1, select_first_waypoint(ANS2).
initialize :- alert_user(ANS), fail.

mission :- in_transit_p(ANS1), ANS1 == 1, transit, !,
transit_done_p(ANS2),
        ANS2 == 1, fail.
mission :- in_search_p(ANS1), ANS1 == 1, search, !,
        search_done_p(ANS2), ANS2 == 1, fail.
mission :- in_task_p(ANS1), ANS1 == 1, task, !, task_done_p(ANS2),
        ANS2 == 1, fail.
mission :- in_return_p(ANS1), ANS1 == 1, return, !, return_done_p(ANS2),
        ANS2 == 1, wait_for_recovery(ANS3).

transit :- waypoint_control.
transit :- surface(ANS).

search :- do_search_pattern(ANS), ANS == 1.
search :- surface(ANS).

task :- homing(ANS1), ANS1 == 1, drop_package(ANS2), ANS2 == 1,
       get_gps_fix(ANS3), ANS3 == 1, get_next_waypoint(ANS4), ANS4 == 1.
task :- surface(ANS).

return :- waypoint_control.
return :- surface(ANS).


/* ------------------------------ NPS AUV DOCTRINE --------------*/

execute_auv_mission :- initialize, repeat, mission.

waypoint_control :- not(crit_system_prob), get_waypoint_status,
        plan, send_setpoints_and_modes(ANS).

 /* need to monitor accuracy of trar   relative to current and GPS  */
 /*obstacle avoidance in Tac, replanning in Strategic */

get_waypoint_status :- gps_check, reach_waypoint(ANS1), ANS1 == 1,
```

```
        get_next_waypoint(ANS2).
get_waypoint_status.

gps_check :- gps_needed(ANS1), ANS1 == 1, get_gps_fix(ANS1).
gps_check.


  /* Planning: reduced capability (cut mission short), system degradation
     (continue mission at lower performance), obstacle avoidance, normal.
g system*/

plan :- red_cap_system_prob, global_replan.

/*  Subset of system probs requiring replan (TBD)  */

plan :- near_uncharted_obstacle, local_replan.
plan.

near_uncharted_obstacle :- unk_obstacle_p(ANS1), ANS1 == 1,
log_new_obstacle(ANS2).

local_replan :- loiter(ANS1), start_local_replanner(ANS2).

global_replan :- loiter(ANS1), start_global_replanner(ANS2).

crit_system_prob :- power_gone_p(ANS), ANS == 1.
crit_system_prob :- computer_system_inop_p(ANS), ANS == 1.
crit_system_prob :- propulsion_system_p(ANS), ANS == 1.
crit_system_prob :- steering_system_inop_p(ANS), ANS == 1.

red_cap_system_prob :- diving_system_p(ANS), ANS == 1.
red_cap_system_prob :- bouyancy_system_p(ANS), ANS == 1.
red_cap_system_prob :- thruster_system_p(ANS), ANS == 1.
red_cap_system_prob :- leak_test_p(ANS), ANS == 1.
red_cap_system_prob :- payload_prob_p(ANS), ANS == 1.
```

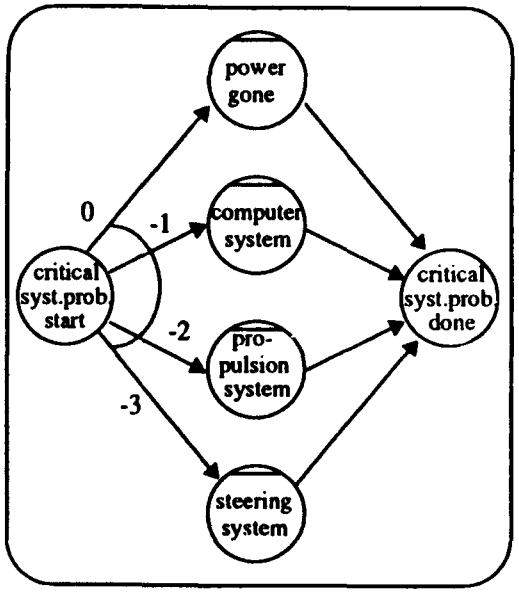
APPENDIX D. AND/OR GOAL TREE OF THE STRATEGIC LEVEL



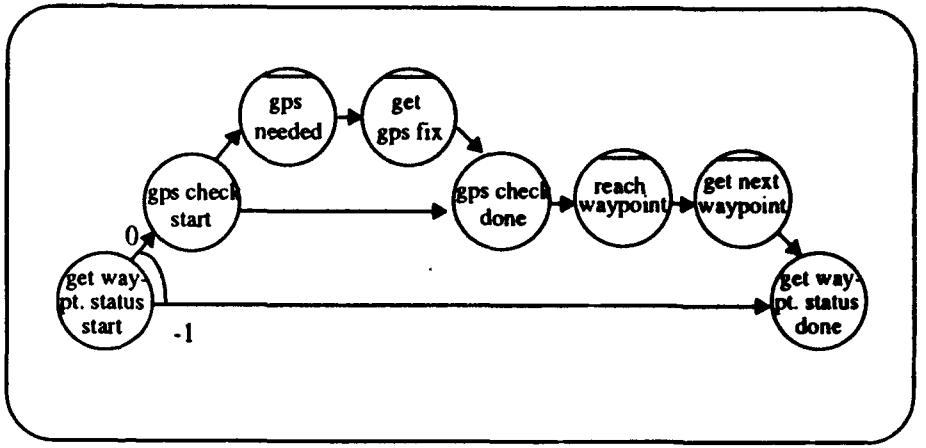
APPENDIX E. STDWP OF THE STRATEGIC LEVEL



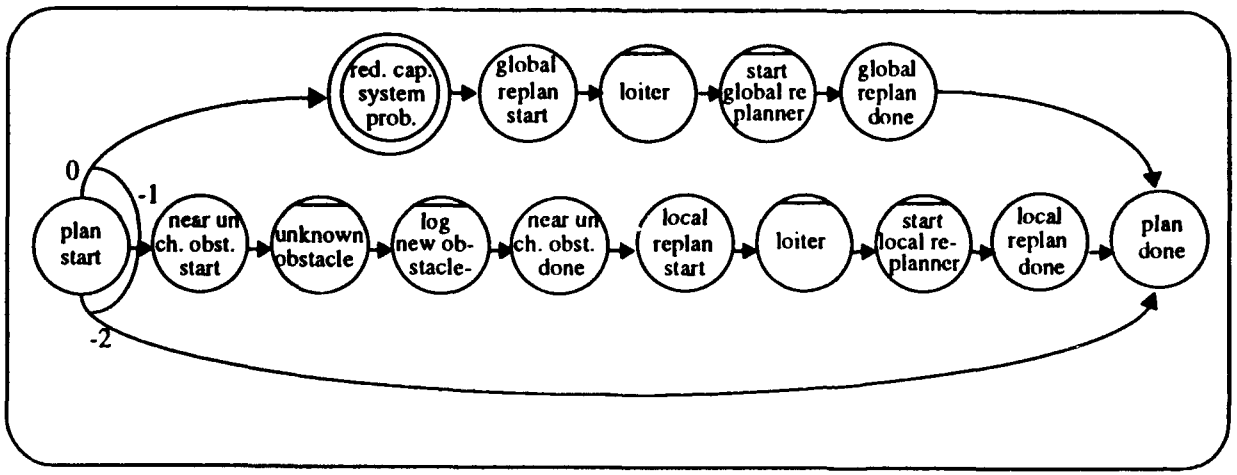
Multi-level STDWP's (see previous page)



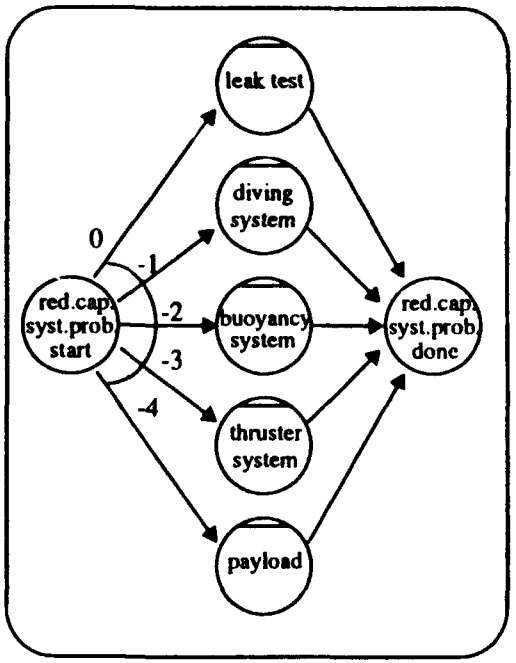
critical system problem



get waypoint status



plan



reduced capability system problem

APPENDIX F. CLIPS IMPLEMENTATION OF THE STRATEGIC LEVEL

```
;*****************************************************************************
;*
;*  Title       : Strategic Level for the NPS AUV II
;*  Name        : strlev6.20
;*  Version     : 6.20
;*  Author      : Thomas Scholz
;*  Date        : 10 August 1993
;*  Revised     :
;*  System      : Sun3  UNIX; server's name "virgo"
;*  Compiler    : CLIPS/Ada 4.40
;*  Description : This program is the strategic level of the NPS AUV II,
;*                top level of the Rational Behavioral Model design
;*
;*  Remarks     : The function names (i.e. "diving_system_prob_p") match the
;*                names that have been used in the tactical level code of
;*                Fritz Thornton [Ref. 13]
;*
;*****************************************************************************

;**********************NPS AUV - RBM Mission Controller/Coordinator**************


;********************************** Templates ***********************************

(deftemplate execute-auv-mission
     (field state
          (type SYMBOL)
          (allowed-symbols       start initialize mission done)))

(deftemplate initialize
     (field state
          (type SYMBOL)
          (allowed-symbols       start
                                 ready-vehicle-for-launch
                                 select-first-waypoint
                                 alert-user
                                 done)))

(deftemplate initialize-branches
     (field branch
          (type INTEGER)         (range 1 2))
     (field status
          (type SYMBOL)
          (allowed-symbols       try failed)))
```

```
(deftemplate mission
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                            transit in-transit-p transit-done-p
                            search in-search-p search-done-p
                            task in-task-p task-done-p
                            return in-return-p return-done-p
                            wait-for-recovery
                            done)))

(deftemplate mission-branches
    (field branch
        (type INTEGER)      (range 1 4))
    (field status
        (type SYMBOL)
        (allowed-symbols    try failed)))

(deftemplate transit
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                            waypoint-control
                            surface
                            done)))

(deftemplate transit-branches
    (field branch
        (type INTEGER)      (range 1 2))
    (field status
        (type SYMBOL)
        (allowed-symbols    try failed)))

(deftemplate search
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                            do-search-pattern
                            surface
                            done)))

(deftemplate search-branches
    (field branch
        (type INTEGER)      (range 1 2))
    (field status
        (type SYMBOL)
        (allowed-symbols    try failed)))

(deftemplate task
    (field state
```

```
        (type SYMBOL)
        (allowed-symbols    start
                            homing drop-package
                            get-gps-fix
                            get-next-waypoint
                            surface
                            done)))

(deftemplate task-branches
    (field branch
        (type INTEGER)      (range 1 2))
    (field status
        (type SYMBOL)
        (allowed-symbols    try failed)))

(deftemplate return
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                            waypoint-control
                            surface
                            done)))

(deftemplate return-branches
    (field branch
        (type INTEGER)      (range 1 2))
    (field status
        (type SYMBOL)
        (allowed-symbols    try failed)))

(deftemplate waypoint-control
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                            crit-system-prob
                            get-waypoint-status
                            plan
                            send-setpoints-and-modes
                            done)))

(deftemplate get-waypoint-status
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                            gps-check
                            reach-waypoint
                            get-next-waypoint
                            done)))

(deftemplate get-waypoint-status-branches
    (field branch
```

```
        (type INTEGER)      (range 1 2))
    (field status
        (type SYMBOL)
        (allowed-symbols    try failed)))

(deftemplate gps-check
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                            gps-needed
                            get-gps-fix
                            done)))

(deftemplate gps-check-branches
    (field branch
        (type INTEGER)      (range 1 2))
    (field status
        (type SYMBOL)
        (allowed-symbols    try failed)))

(deftemplate plan
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                            red-cap-system-prob
                            global-replan
                            near-uncharted-obstacle
                            local-replan
                            done)))

(deftemplate plan-branches
    (field branch
        (type INTEGER)      (range 1 3))
    (field status
        (type SYMBOL)
        (allowed-symbols    try failed)))

(deftemplate near-uncharted-obstacle
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                            unknown-obstacle-p
                            log-new-obstacle
                            done)))

(deftemplate local-replan
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                            loiter
                            start-local-replanner
```

```
                    done)))

(deftemplate global-replan
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                    loiter
                    start-global-replanner
                    done)))

(deftemplate crit-system-prob
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                    power-gone-p
                    computer-system-inop-p
                    propulsion-system-p
                    steering-system-inop-p
                    not-done
                    done)))

(deftemplate crit-system-prob-branches
    (field branch
        (type INTEGER)      (range 1 4)))
;   (field status
;       (type SYMBOL)
;       (allowed-symbols    try failed)))

(deftemplate red-cap-system-prob
    (field state
        (type SYMBOL)
        (allowed-symbols    start
                    diving-system-p
                    bouyancy-system-p
                    thruster-system-p
                    leak-test-p
                    payload-prob-p
                    done)))

(deftemplate red-cap-system-prob-branches
    (field branch
        (type INTEGER)      (range 1 5)))
;   (field status
;       (type SYMBOL)
;       (allowed-symbols    try failed)))

; ************************************** Rules **********************************

;-------------------------------------------------execute-auv-mission--------

(defrule start-program
```

```
     ?x <- (start)
=>
     (retract ?x)
     (assert (execute-auv-mission (state start))))

(defrule execute-auv-mission-state-initialize
     ?x <- (execute-auv-mission (state start))
=>
     (retract ?x)
     (assert (execute-auv-mission (state initialize)))
     (assert (initialize (state start))))

(defrule execute-auv-mission-state-mission
     ?x <- (execute-auv-mission (state initialize))
     ?y <- (initialize (state done))
=>
     (retract ?x)
     (retract ?y)
     (assert (execute-auv-mission (state mission)))
     (assert (mission (state start))))

(defrule execute-auv-mission-state-done
     ?x <- (execute-auv-mission (state mission))
     ?y <- (mission (state done))
=>
     (retract ?x)
     (retract ?y)
     (assert (execute-auv-mission (state done)))
     (printout t crlf "************************************" crlf
               "* MISSION EXECUTED SUCCESSFULLY. *" crlf
               "* AUV IS WAITING FOR RECOVERY... *" crlf
               "************************************" crlf crlf))

;=========================================================================

;----------------------------------initialize traffic rules-----------

(defrule initialize-branch-1-start
     ?x <- (initialize (state start))
=>
     (retract ?x)
     (assert (initialize-branches (branch 1)(status try))))

(defrule initialize-branch-2-start
     (declare (salience -10))
     ?x <- (initialize-branches (branch 1)(status failed))
=>
     (retract ?x)
     (assert (initialize-branches (branch 2)(status try))))
```

```
(defrule initialize-branches-clean-up
     (declare (salience -20))
     ?x <- (initialize-branches (branch 2)(status failed))
     ?y <- (initialize (state ?))
=>
     (retract ?x)
     (retract ?y)
     (printout t "No initialize branch successful!" crlf crlf))

(defrule initialize-branch-failure
     (declare (salience -100))
     ?x <- (initialize-branches (branch ?n)(status try))
     ?y <- (initialize (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (initialize-branches (branch ?n)(status failed))))

;-----------------------------------initialize branch 1----------------

(defrule initialize-1-state-ready-vehicle-for-launch
     (initialize-branches (branch 1)(status try))
=>
     (assert (initialize (state ready-vehicle-for-launch))))

(defrule initialize-1-state-select-first-waypoint
     (initialize-branches (branch 1)(status try))
     ?x <- (initialize (state ready-vehicle-for-launch))
     (test (= (ready_vehicle_for_launch) 1))
=>
     (retract ?x)
     (assert (initialize (state select-first-waypoint))))

(defrule initialize-1-state-done
     ?x <- (initialize-branches (branch 1)(status try))
     ?y <- (initialize (state select-first-waypoint))
=>
     (select_first_waypoint)
     (retract ?x)
     (retract ?y)
     (assert (initialize (state done))))

;-----------------------------------initialize branch 2----------------

(defrule initialize-2-state-alert-user
     (initialize-branches (branch 2)(status try))
=>
     (assert (initialize (state alert-user))))

(defrule initialize-2-state-done
     ?x <- (initialize-branches (branch 2)(status try))
```

```
      ?y <- (initialize (state alert-user))
=>
      (alert_user)
      (retract ?x)
      (retract ?y)
      (assert (initialize (state done))))


;=============================================================================
;-----------------------------------------mission traffic rules----------

(defrule mission-branch-1-start
      ?x <- (mission (state start))
=>
      (retract ?x)
      (assert (mission-branches (branch 1)(status try))))

(defrule mission-branch-2-start
      (declare (salience -10))
      ?x <- (mission-branches (branch 1)(status failed))
=>
      (retract ?x)
      (assert (mission-branches (branch 2)(status try))))

(defrule mission-branch-3-start
      (declare (salience -20))
      ?x <- (mission-branches (branch 2)(status failed))
=>
      (retract ?x)
      (assert (mission-branches (branch 3)(status try))))

(defrule mission-branch-4-start
      (declare (salience -30))
      ?x <- (mission-branches (branch 3)(status failed))
=>
      (retract ?x)
      (assert (mission-branches (branch 4)(status try))))

(defrule mission-branches-clean-up
      (declare (salience -40))
      ?x <- (mission-branches (branch 4))
=>
      (retract ?x)
      (assert (mission (state start))))

;(defrule mission-branch-repeat
;     (declare (salience -10000))
;     ?x <- (mission-branches (branch ?n)(status try))
;     ?y <- (mission (state ?))
;=>
;     (retract ?x)
```

```
;     (retract ?y)
;     (assert (mission (state start))))

(defrule mission-branch-failure
     (declare (salience -1000))
     ?x <- (mission-branches (branch ?n)(status try))
     ?y <- (mission (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (mission-branches (branch ?n)(status failed))))

;-----------------------------------------mission branch 1----------

(defrule mission-1-state-in-transit-p
     (mission-branches (branch 1)(status try))
=>
     (assert (mission (state in-transit-p))))

(defrule mission-1-state-transit
     (mission-branches (branch 1)(status try))
     ?x <- (mission (state in-transit-p))
     (test (= (in_transit_p) 1))
=>
     (retract ?x)
     (assert (mission (state transit)))
     (assert (transit (state start))))

(defrule mission-1-state-transit-done-p
     ?x <- (mission-branches (branch 1)(status try))
;    (mission-branches (branch 1)(status try))
     ?y <- (mission (state transit))
     ?z <- (transit (state done))
     (test (= (transit_done_p) 1))
=>
     (retract ?x)
     (retract ?y)
     (retract ?z)
     (assert (mission (state transit-done-p)))
     (assert (mission-branches (branch 1)(status try)))
     (printout t crlf "**********************************" crlf
                 "*      TRANSIT SUCCESSFUL.     *" crlf
                 "**********************************" crlf crlf))

;-----------------------------------------mission branch 2----------

(defrule mission-2-state-in-search-p
     (mission-branches (branch 2)(status try))
=>
     (assert (mission (state in-search-p))))
```

```
(defrule mission-2-state-search
     (mission-branches (branch 2)(status try))
     ?x <- (mission (state in-search-p))
     (test (= (in_search_p) 1))
=>
     (retract ?x)
     (assert (mission (state search)))
     (assert (search (state start))))

(defrule mission-2-state-search-done-p
     ?x <- (mission-branches (branch 2)(status try))
;    (mission-branches (branch 2)(status try))
     ?y <- (mission (state search))
     ?z <- (search (state done))
     (test (= (search_done_p) 1))
=>
     (retract ?x)
     (retract ?y)
     (retract ?z)
     (assert (mission (state search-done-p)))
     (assert (mission-branches (branch 2)(status try)))
     (printout t crlf "**********************************" crlf
               "*      SEARCH SUCCESSFUL.     *" crlf
               "**********************************" crlf crlf))

;-----------------------------------------mission branch 3----------------

(defrule mission-3-state-in-task-p
     (mission-branches (branch 3)(status try))
=>
     (assert (mission (state in-task-p))))

(defrule mission-3-state-task
     (mission-branches (branch 3)(status try))
     ?x <- (mission (state in-task-p))
     (test (= (in_task_p) 1))
=>
     (retract ?x)
     (assert (mission (state task)))
     (assert (task (state start))))

(defrule mission-3-state-task-done-p
     ?x <- (mission-branches (branch 3)(status try))
;    (mission-branches (branch 3)(status try))
     ?y <- (mission (state task))
     ?z <- (task (state done))
     (test (= (task_done_p) 1))
=>
     (retract ?x)
     (retract ?y)
     (retract ?z)
```

```
    (assert (mission (state task-done-p)))
    (assert (mission-branches (branch 3)(status try)))
    (printout t crlf "**************************************" crlf
              "*       TASK SUCCESSFUL.      *" crlf
              "**************************************" crlf crlf))

;-----------------------------------------mission branch 4----------

(defrule mission-4-state-in-return-p
    (mission-branches (branch 4)(status try))
=>
    (assert (mission (state in-return-p))))

(defrule mission-4-state-return
    (mission-branches (branch 4)(status try))
    ?x <- (mission (state in-return-p))
    (test (= (in_return_p) 1))
=>
    (retract ?x)
    (assert (mission (state return)))
    (assert (return (state start))))

(defrule mission-4-state-return-done-p
    ?x <- (mission-branches (branch 4)(status try))
;   (mission-branches (branch 4)(status try))
    ?y <- (mission (state return))
    ?z <- (return (state done))
    (test (= (return_done_p) 1))
=>
    (retract ?x)
    (retract ?y)
    (retract ?z)
    (wait_for_recovery)
    (printout t crlf "**************************************" crlf
              "*       RETURN SUCCESSFUL.      *" crlf
              "**************************************" crlf crlf)
    (assert (mission (state done))))

;=============================================================================

;-----------------------------------------transit traffic rules-------

(defrule transit-branch-1-start
    ?x <- (transit (state start))
=>
    (retract ?x)
    (assert (transit-branches (branch 1)(status try))))

(defrule transit-branch-2-start
    (declare (salience -10))
    ?x <- (transit-branches (branch 1)(status failed))
```

```
=>
    (retract ?x)
    (assert (transit-branches (branch 2)(status try))))

(defrule transit-branches-clean-up
    (declare (salience -20))
    ?x <- (transit-branches (branch 2)(status failed))
=>
    (retract ?x)
    (printout t "No transit branch successful!" crlf crlf))

(defrule transit-branch-failure
    (declare (salience -100))
    ?x <- (transit-branches (branch ?n)(status try))
    ?y <- (transit (state ?))
=>
    (retract ?x)
    (retract ?y)
    (assert (transit-branches (branch ?n)(status failed))))

;----------------------------------------------------transit branch 1---------

(defrule transit-1-state-waypoint-control
    (transit-branches (branch 1)(status try))
=>
    (assert (transit (state waypoint-control)))
    (assert (waypoint-control (state start))))

(defrule transit-1-state-done
    ?x <- (transit-branches (branch 1)(status try))
    ?y <- (transit (state waypoint-control))
    ?z <- (waypoint-control (state done))
=>
    (retract ?x)
    (retract ?y)
    (retract ?z)
    (assert (transit (state done))))

;----------------------------------------------------transit branch 2---------

(defrule transit-2-state-surface
    (transit-branches (branch 2)(status try))
=>
    (assert (transit (state surface))))

(defrule transit-2-state-done
    ?x <- (transit-branches (branch 2)(status try))
    ?y <- (transit (state surface))
=>
    (retract ?x)
    (retract ?y)
```

```
      (surface))

;     (assert (transit (state done))))    ; is transit done???

;=============================================================
;-----------------------------------------search traffic rules--------

(defrule search-branch-1-start
      ?x <- (search (state start))
=>
      (retract ?x)
      (assert (search-branches (branch 1)(status try))))

(defrule search-branch-2-start
      (declare (salience -10))
      ?x <- (search-branches (branch 1)(status failed))
=>
      (retract ?x)
      (assert (search-branches (branch 2)(status try))))

(defrule search-branches-clean-up
      (declare (salience -20))
      ?x <- (search-branches (branch 2)(status failed))
=>
      (retract ?x)
      (printout t "No search branch successful!" crlf crlf))

(defrule search-branch-failure
      (declare (salience -100))
      ?x <- (search-branches (branch ?n)(status try))
      ?y <- (search (state ?))
=>
      (retract ?x)
      (retract ?y)
      (assert (search-branches (branch ?n)(status failed))))

;-----------------------------------------------search branch 1----------

(defrule search-1-state-do-search-pattern
      (search-branches (branch 1)(status try))
=>
      (assert (search (state do-search-pattern))))

(defrule search-1-state-done
      ?x <- (search-branches (branch 1)(status try))
      ?y <- (search (state do-search-pattern))
      (test (= (do_search_pattern) 1))
=>
      (retract ?x)
      (retract ?y)
      (assert (search (state done))))
```

```
;-----------------------------------------------search branch 2---------

(defrule search-2-state-surface
     (search-branches (branch 2)(status try))
=>
     (assert (search (state surface))))

(defrule search-2-state-done
     ?x <- (search-branches (branch 2)(status try))
     ?y <- (search (state surface))
=>
     (retract ?x)
     (retract ?y)
     (surface))

;    (assert (transit (state done))))    ; is transit done???

;=============================================================================
;-----------------------------------------------task traffic rules-------

(defrule task-branch-1-start
     ?x <- (task (state start))
=>
     (retract ?x)
     (assert (task-branches (branch 1)(status try))))

(defrule task-branch-2-start
     (declare (salience -10))
     ?x <- (task-branches (branch 1)(status failed))
=>
     (retract ?x)
     (assert (task-branches (branch 2)(status try))))

(defrule task-branches-clean-up
     (declare (salience -20))
     ?x <- (task-branches (branch 2)(status failed))
=>
     (retract ?x)
     (printout t "No task branch successful!" crlf crlf))

(defrule task-branch-failure
     (declare (salience -100))
     ?x <- (task-branches (branch ?n)(status try))
     ?y <- (task (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (task-branches (branch ?n)(status failed))))
```

```
;----------------------------------------------------task branch 1---------

(defrule task-1-state-homing
     (task-branches (branch 1)(status try))
=>
     (assert (task (state homing))))

(defrule task-1-state-drop-package
     (task-branches (branch 1)(status try))
     ?x <- (task (state homing))
     (test (= (homing) 1))
=>
     (retract ?x)
     (assert (task (state drop-package))))

(defrule task-1-state-get-gps-fix
     (task-branches (branch 1)(status try))
     ?x <- (task (state drop-package))
     (test (= (drop_package) 1))
=>
     (retract ?x)
     (assert (task (state get-gps-fix))))

(defrule task-1-state-get-next-waypoint
     (task-branches (branch 1)(status try))
     ?x <- (task (state get-gps-fix))
     (test (= (get_gps_fix) 1))
=>
     (retract ?x)
     (assert (task (state get-next-waypoint))))

(defrule task-1-state-done
     ?x <- (task-branches (branch 1)(status try))
     ?y <- (task (state get-next-waypoint))
     (test (= (get_next_waypoint) 1))
=>
     (retract ?x)
     (retract ?y)
     (assert (task (state done))))

;----------------------------------------------------task branch 2---------

(defrule task-2-state-surface
     (task-branches (branch 2)(status try))
=>
     (assert (task (state surface))))

(defrule task-2-state-done
     ?x <- (task-branches (branch 2)(status try))
     ?y <- (task (state surface))
=>
```

```
      (retract ?x)
      (retract ?y)
      (surface))

;     (assert (task (state done))))    ; is task done???

;=============================================================================
;-----------------------------------------------return traffic rules--------

(defrule return-branch-1-start
      ?x <- (return (state start))
=>
      (retract ?x)
      (assert (return-branches (branch 1)(status try))))

(defrule return-branch-2-start
      (declare (salience -10))
      ?x <- (return-branches (branch 1)(status failed))
=>
      (retract ?x)
      (assert (return-branches (branch 2)(status try))))

(defrule return-branches-clean-up
      (declare (salience -20))
      ?x <- (return-branches (branch 2)(status failed))
=>
      (retract ?x)
      (printout t "No return branch successful!" crlf crlf))

(defrule return-branch-failure
      (declare (salience -100))
      ?x <- (return-branches (branch ?n)(status try))
      ?y <- (return (state ?))
=>
      (retract ?x)
      (retract ?y)
      (assert (return-branches (branch ?n)(status failed))))

;-----------------------------------------------return branch 1---------

(defrule return-1-state-waypoint-control
      (return-branches (branch 1)(status try))
=>
      (assert (return (state waypoint-control)))
      (assert (waypoint-control (state start))))

(defrule return-1-state-done
      ?x <- (return-branches (branch 1)(status try))
      ?y <- (return (state waypoint-control))
      ?z <- (waypoint-control (state done))
=>
```

```
    (retract ?x)
    (retract ?y)
    (retract ?z)
    (assert (return (state done))))
;   (assert (mission (state start))))

;-----------------------------------------------return branch 2---------

(defrule return-2-state-surface
    (return-branches (branch 2)(status try))
=>
    (assert (return (state surface))))

(defrule return-2-state-done
    ?x <- (return-branches (branch 2)(status try))
    ?y <- (return (state surface))
=>
    (retract ?x)
    (retract ?y)
    (surface))

;   (assert (return (state done))))   ; is return done???

;=================================================================================
;-----------------------------------------------waypoint-control--------

(defrule waypoint-control-state-crit-system-prob
    ?x <- (waypoint-control (state start))
=>
    (retract ?x)
    (assert (waypoint-control (state crit-system-prob)))
    (assert (crit-system-prob (state start))))

(defrule waypoint-control-state-get-waypoint-status
    ?x <- (waypoint-control (state crit-system-prob))
    ?y <- (crit-system-prob (state not-done))
=>
    (retract ?x)
    (retract ?y)
    (assert (waypoint-control (state get-waypoint-status)))
    (assert (get-waypoint-status (state start))))

(defrule waypoint-control-state-plan
    ?x <- (waypoint-control (state get-waypoint-status))
    ?y <- (get-waypoint-status (state done))
=>
    (retract ?x)
    (retract ?y)
    (assert (waypoint-control (state plan)))
    (assert (plan (state start))))
```

```
(defrule waypoint-control-state-done
     ?x <- (waypoint-control (state plan))
     ?y <- (plan (state done))
=>
     (retract ?x)
     (retract ?y)
     (send_setpoints_and_modes)
     (assert (waypoint-control (state done))))

;=============================================================================
;-------------------------------------get-waypoint-status traffic rules---


(defrule get-waypoint-status-branch-1-start
     ?x <- (get-waypoint-status (state start))
=>
     (retract ?x)
     (assert (get-waypoint-status-branches (branch 1)(status try))))

(defrule get-waypoint-status-branch-2-start
     (declare (salience -10))
     ?x <- (get-waypoint-status-branches (branch 1)(status failed))
=>
     (retract ?x)
     (assert (get-waypoint-status-branches (branch 2)(status try))))

(defrule get-waypoint-status-branches-clean-up
     (declare (salience -20))
     ?x <- (get-waypoint-status-branches (branch 2)(status failed))
=>
     (retract ?x)
     (printout t "No get-waypoint-status branch successful!" crlf crlf))

(defrule get-waypoint-status-branch-failure
     (declare (salience -100))
     ?x <- (get-waypoint-status-branches (branch ?n)(status try))
     ?y <- (get-waypoint-status (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (get-waypoint-status-branches (branch ?n)(status failed))))

;-------------------------------------get-waypoint-status branch 1--------

(defrule get-waypoint-status-1-state-gps-check
     (get-waypoint-status-branches (branch 1)(status try))
=>
     (assert (get-waypoint-status (state gps-check)))
     (assert (gps-check (state start))))

(defrule get-waypoint-status-1-state-reach-waypoint
```

```
     (get-waypoint-status-branches (branch 1)(status try))
     ?x <- (get-waypoint-status (state gps-check))
     ?y <- (gps-check (state done))
=>
     (retract ?x)
     (retract ?y)
     (assert (get-waypoint-status (state reach-waypoint))))


(defrule get-waypoint-status-1-state-done
     ?x <- (get-waypoint-status-branches (branch 1)(status try))
     ?y <- (get-waypoint-status (state reach-waypoint))
     (test (= (reach_waypoint_p) 1))
=>
     (retract ?x)
     (retract ?y)
     (get_next_waypoint)
     (assert (get-waypoint-status (state done))))

;---------------------------------------get-waypoint-status branch 2--------

(defrule get-waypoint-status-2-state-done
     ?x <- (get-waypoint-status-branches (branch 2)(status try))
=>
     (retract ?x)
     (assert (get-waypoint-status (state done))))

;=============================================================================
;-----------------------------------------------gps-check traffic rules---


(defrule gps-check-branch-1-start
     ?x <- (gps-check (state start))
=>
     (retract ?x)
     (assert (gps-check-branches (branch 1)(status try))))

(defrule gps-check-branch-2-start
     (declare (salience -10))
     ?x <- (gps-check-branches (branch 1)(status failed))
=>
     (retract ?x)
     (assert (gps-check-branches (branch 2)(status try))))

(defrule gps-check-branches-clean-up
     (declare (salience -20))
     ?x <- (gps-check-branches (branch 2)(status failed))
=>
     (retract ?x)
     (printout t "No gps-check branch successful!" crlf crlf))
```

```
(defrule gps-check-branch-failure
     (declare (salience -100))
     ?x <- (gps-check-branches (branch ?n)(status try))
     ?y <- (gps-check (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (gps-check-branches (branch ?n)(status failed))))

;-------------------------------------------------gps-check branch 1--------

(defrule gps-check-1-state-gps-needed
     (gps-check-branches (branch 1)(status try))
=>
     (assert (gps-check (state gps-needed))))

(defrule gps-check-1-state-get-gps-fix
     (gps-check-branches (branch 1)(status try))
     ?x <- (gps-check (state gps-needed))
     (test (= (gps_needed_p) 1))
=>
     (retract ?x)
     (assert (gps-check (state get-gps-fix))))

(defrule gps-check-1-state-done
     ?x <- (gps-check-branches (branch 1)(status try))
     ?y <- (gps-check (state get-gps-fix))
=>
     (retract ?x)
     (retract ?y)
     (get_gps_fix)
     (assert (gps-check (state done))))

;-------------------------------------------------gps-check branch 2--------

(defrule gps-check-2-state-done
     ?x <- (gps-check-branches (branch 2)(status try))
=>
     (retract ?x)
     (assert (gps-check (state done))))

;=============================================================================
;-------------------------------------------------plan traffic rules--------


(defrule plan-branch-1-start
     ?x <- (plan (state start))
=>
     (retract ?x)
     (assert (plan-branches (branch 1)(status try))))
```

```
(defrule plan-branch-2-start
     (declare (salience -10))
     ?x <- (plan-branches (branch 1)(status failed))
=>
     (retract ?x)
     (assert (plan-branches (branch 2)(status try))))

(defrule plan-branch-3-start
     (declare (salience -20))
     ?x <- (plan-branches (branch 2)(status failed))
=>
     (retract ?x)
     (assert (plan-branches (branch 3)(status try))))

(defrule plan-branches-clean-up
     (declare (salience -30))
     ?x <- (plan-branches (branch 3)(status failed))
=>
     (retract ?x)
     (printout t "No plan branch successful!" crlf crlf))

(defrule plan-branch-failure
     (declare (salience -100))
     ?x <- (plan-branches (branch ?n)(status try))
     ?y <- (plan (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (plan-branches (branch ?n)(status failed))))

;-------------------------------------------------plan branch 1------------

(defrule plan-1-state-red-cap-system-prob
     (plan-branches (branch 1)(status try))
=>
     (assert (plan (state red-cap-system-prob)))
     (assert (red-cap-system-prob (state start))))

(defrule plan-1-state-global-replan
     (plan-branches (branch 1)(status try))
     ?x <- (plan (state red-cap-system-prob))
     ?y <- (red-cap-system-prob (state done))
=>
     (retract ?x)
     (retract ?y)
     (assert (plan (state global-replan)))
     (assert (global-replan (state start))))

(defrule plan-1-state-done
     ?x <- (plan-branches (branch 1)(status try))
     ?y <- (plan (state global-replan))
```

```
    ?z <- (global-replan (state done))
=>
    (retract ?x)
    (retract ?y)
    (retract ?z)
    (assert (plan (state done))))

;-----------------------------------------------plan branch 2------------

(defrule plan-2-state-near-uncharted-obstacle
    (plan-branches (branch 2)(status try))
=>
    (assert (plan (state near-uncharted-obstacle)))
    (assert (near-uncharted-obstacle (state start))))

(defrule plan-2-state-local-replan
    (plan-branches (branch 2)(status try))
    ?x <- (plan (state near-uncharted-obstacle))
    ?y <- (near-uncharted-obstacle (state done))
=>
    (retract ?x)
    (retract ?y)
    (assert (plan (state local-replan)))
    (assert (local-replan (state start))))

(defrule plan-2-state-done
    ?x <- (plan-branches (branch 2)(status try))
    ?y <- (plan (state local-replan))
    ?z <- (local-replan (state done))
=>
    (retract ?x)
    (retract ?y)
    (retract ?z)
    (assert (plan (state done))))

;-----------------------------------------------plan branch 3------------

(defrule plan-3-state-done
    ?x <- (plan-branches (branch 3)(status try))
=>
    (retract ?x)
    (assert (plan (state done))))

;=============================================================================
;-----------------------------------------------near-uncharted-obstacle-----

(defrule near-uncharted-obstacle-state-unknown-obstacle-p
    (near-uncharted-obstacle (state start))
=>
    (assert (near-uncharted-obstacle (state unknown-obstacle-p))))
```

```
(defrule near-uncharted-obstacle-state-log-new-obstacle
     ?x <- (near-uncharted-obstacle (state unknown-obstacle-p))
     (test (= (unknown_obstacle_p) 1))
=>
     (retract ?x)
     (assert (near-uncharted-obstacle (state log-new-obstacle))))

(defrule near-uncharted-obstacle-state-done
     ?x <- (near-uncharted-obstacle (state log-new-obstacle))
=>
     (log_new_obstacle)
     (retract ?x)
     (assert (near-uncharted-obstacle (state done))))

;=============================================================================
;-------------------------------------------------------local-replan-----

(defrule local-replan-state-loiter
     (local-replan (state start))
=>
     (assert (local-replan (state loiter))))

(defrule local-replan-state-start-local-replanner
     ?x <- (local-replan (state loiter))
=>
     (loiter)
     (retract ?x)
     (assert (local-replan (state start-local-replanner))))

(defrule local-replan-state-done
     ?x <- (local-replan (state start-local-replanner))
=>
     (start_local_replanner)
     (retract ?x)
     (assert (local-replan (state done))))

;=============================================================================
;-------------------------------------------------------global-replan----

(defrule global-replan-state-loiter
     (global-replan (state start))
=>
     (assert (global-replan (state loiter))))

(defrule global-replan-state-start-global-replanner
     ?x <- (global-replan (state loiter))
=>
     (loiter)
     (retract ?x)
     (assert (global-replan (state start-global-replanner))))
```

```
(defrule global-replan-state-done
     ?x <- (global-replan (state start-global-replanner))
=>
     (start_global_replanner)
     (retract ?x)
     (assert (global-replan (state done))))

;=============================================================================
;-------------------------------------crit-system-prob traffic rules-----


(defrule crit-system-prob-branch-1-start
     ?x <- (crit-system-prob (state start))
=>
     (retract ?x)
     (assert (crit-system-prob-branches (branch 1))))

(defrule crit-system-prob-branch-2-start
     (declare (salience -10))
     ?x <- (crit-system-prob-branches (branch 1))
     ?y <- (crit-system-prob (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (crit-system-prob-branches (branch 2))))

(defrule crit-system-prob-branch-3-start
     (declare (salience -20))
     ?x <- (crit-system-prob-branches (branch 2))
     ?y <- (crit-system-prob (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (crit-system-prob-branches (branch 3))))

(defrule crit-system-prob-branch-4-start
     (declare (salience -30))
     ?x <- (crit-system-prob-branches (branch 3))
     ?y <- (crit-system-prob (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (crit-system-prob-branches (branch 4))))

(defrule crit-system-prob-branches-clean-up
     (declare (salience -40))
     ?x <- (crit-system-prob-branches (branch 4))
     ?y <- (crit-system-prob (state ?))
=>
     (retract ?x)
     (retract ?y)
```

```
    (assert (crit-system-prob (state not-done)))
    (printout t "No crit-system-prob branch successful!" crlf crlf))

;-----------------------------------------------crit-system-prob-1-----

(defrule crit-system-prob-1-state-power-gone-p
    (crit-system-prob-branches (branch 1))
=>
    (assert (crit-system-prob (state power-gone-p))))

(defrule crit-system-prob-1-state-done
    ?x <- (crit-system-prob-branches (branch 1))
    ?y <- (crit-system-prob (state power-gone-p))
    (test (= (power_gone_p) 1))
=>
    (retract ?x)
    (retract ?y)
    (assert (crit-system-prob (state done))))

;-----------------------------------------------crit-system-prob-2-----

(defrule crit-system-prob-2-state-computer-system-inop-p
    (crit-system-prob-branches (branch 2))
=>
    (assert (crit-system-prob (state computer-system-inop-p))))

(defrule crit-system-prob-2-state-done
    ?x <- (crit-system-prob-branches (branch 2))
    ?y <- (crit-system-prob (state computer-system-inop-p))
    (test (= (computer_system_prob_p) 1))
=>
    (retract ?x)
    (retract ?y)
    (assert (crit-system-prob (state done))))

;-----------------------------------------------crit-system-prob-3-----

(defrule crit-system-prob-3-state-propulsion-system-p
    (crit-system-prob-branches (branch 3))
=>
    (assert (crit-system-prob (state propulsion-system-p))))

(defrule crit-system-prob-3-state-done
    ?x <- (crit-system-prob-branches (branch 3))
    ?y <- (crit-system-prob (state propulsion-system-p))
    (test (= (propulsion_system_prob_p) 1))
=>
    (retract ?x)
    (retract ?y)
    (assert (crit-system-prob (state done))))
```

```
;-----------------------------------------------crit-system-prob-4-----

(defrule crit-system-prob-4-state-steering-system-inop-p
     (crit-system-prob-branches (branch 4))
=>
     (assert (crit-system-prob (state steering-system-inop-p))))

(defrule crit-system-prob-4-state-done
     ?x <- (crit-system-prob-branches (branch 4))
     ?y <- (crit-system-prob (state steering-system-inop-p))
     (test (= (steering_system_prob_p) 1))
=>
     (retract ?x)
     (retract ?y)
     (assert (crit-system-prob (state done))))

;=============================================================================
;---------------------------------------red-cap-system-prob traffic rules-----

(defrule red-cap-system-prob-branch-1-start
     ?x <- (red-cap-system-prob (state start))
=>
     (retract ?x)
     (assert (red-cap-system-prob-branches (branch 1))))

(defrule red-cap-system-prob-branch-2-start
     (declare (salience -10))
     ?x <- (red-cap-system-prob-branches (branch 1))
     ?y <- (red-cap-system-prob (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (red-cap-system-prob-branches (branch 2))))

(defrule red-cap-system-prob-branch-3-start
     (declare (salience -20))
     ?x <- (red-cap-system-prob-branches (branch 2))
     ?y <- (red-cap-system-prob (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (red-cap-system-prob-branches (branch 3))))

(defrule red-cap-system-prob-branch-4-start
     (declare (salience -30))
     ?x <- (red-cap-system-prob-branches (branch 3))
     ?y <- (red-cap-system-prob (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (red-cap-system-prob-branches (branch 4))))
```

```
(defrule red-cap-system-prob-branch-5-start
     (declare (salience -40))
     ?x <- (red-cap-system-prob-branches (branch 4))
     ?y <- (red-cap-system-prob (state ?))
=>
     (retract ?x)
     (retract ?y)
     (assert (red-cap-system-prob-branches (branch 5))))

(defrule red-cap-system-prob-branches-clean-up
     (declare (salience -50))
     ?x <- (red-cap-system-prob-branches (branch 5))
     ?y <- (red-cap-system-prob (state ?))
=>
     (retract ?x)
     (retract ?y)
     (printout t "No red-cap-system-prob branch successful!" crlf crlf))

;-----------------------------------------------red-cap-system-prob-1-----

(defrule red-cap-system-prob-1-state-diving-system-p
     (red-cap-system-prob-branches (branch 1))
=>
     (assert (red-cap-system-prob (state diving-system-p))))

(defrule red-cap-system-prob-1-state-done
     ?x <- (red-cap-system-prob-branches (branch 1))
     ?y <- (red-cap-system-prob (state diving-system-p))
     (test (= (diving_system_prob_p) 1))
=>
     (retract ?x)
     (retract ?y)
     (assert (red-cap-system-prob (state done))))

;-----------------------------------------------red-cap-system-prob-2-----

(defrule red-cap-system-prob-2-state-bouyancy-system-p
     (red-cap-system-prob-branches (branch 2))
=>
     (assert (red-cap-system-prob (state bouyancy-system-p))))

(defrule red-cap-system-prob-2-state-done
     (declare (salience -10))
     ?x <- (red-cap-system-prob-branches (branch 2))
     ?y <- (red-cap-system-prob (state bouyancy-system-p))
     (test (= (buoyancy_system_prob_p) 1))
=>
     (retract ?x)
     (retract ?y)
     (assert (red-cap-system-prob (state done))))
```

```
;-----------------------------------------------red-cap-system-prob-3-----

(defrule red-cap-system-prob-3-state-thruster-system-p
     (red-cap-system-prob-branches (branch 3))
=>
     (assert (red-cap-system-prob (state thruster-system-p))))

(defrule red-cap-system-prob-3-state-done
     ?x <- (red-cap-system-prob-branches (branch 3))
     ?y <- (red-cap-system-prob (state thruster-system-p))
     (test (= (thruster_system_prob_p) 1))
=>
     (retract ?x)
     (retract ?y)
     (assert (red-cap-system-prob (state done))))

;-----------------------------------------------red-cap-system-prob-4-----

(defrule red-cap-system-prob-4-state-leak-test-p
     (red-cap-system-prob-branches (branch 4))
=>
     (assert (red-cap-system-prob (state leak-test-p))))

(defrule red-cap-system-prob-4-state-done
     ?x <- (red-cap-system-prob-branches (branch 4))
     ?y <- (red-cap-system-prob (state leak-test-p))
     (test (= (leak_test_p) 1))
=>
     (retract ?x)
     (retract ?y)
     (assert (red-cap-system-prob (state done))))

;-----------------------------------------------red-cap-system-prob-5-----

(defrule red-cap-system-prob-5-state-payload-prob-p
     (red-cap-system-prob-branches (branch 5))
=>
     (assert (red-cap-system-prob (state payload-prob-p))))

(defrule red-cap-system-prob-5-state-done
     ?x <- (red-cap-system-prob-branches (branch 5))
     ?y <- (red-cap-system-prob (state payload-prob-p))
     (test (= (payload_prob_p) 1))
=>
     (retract ?x)
     (retract ?y)
     (assert (red-cap-system-prob (state done))))
;===============================================================================
;*******************************************************************************
```

LIST OF REFERENCES

[Ref. 1] Thornton, F. P. B., *A Concurrent, Object-Based Implementation for the Tactical Level of the Rational Behavior Model*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1993.

[Ref. 2] Epp, S. S., *Discrete Mathematics with Applications*, Wadsworth Publishing Company, Belmont, California, 1990.

[Ref. 3] Zissos, D., *Problems and Solutions in Logic Design*, 2nd ed., Oxford University Press, Oxford, Great Britain, 1979.

[Ref. 4] Hill, F. J., Peterson, G. R., *Introduction to Switching Theory and Logical Design*, 2nd ed., John Wiley & Sons, New York, New York, 1974.

[Ref. 5] Krieger, M., *Basic Switching Circuit Theory*, The McMillan Company, New York, New York, 1967.

[Ref. 6] Mano, M. M., *Computer Logic Design*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1972.

[Ref. 7] Naval Postgraduate School - Department of Computer Science Technical Report NPS CS-93-003, *The State Transition Diagram with Path Priority and Its Applicability to the Translation between Backward and Forward Implementations of the Rational Behavior Model*, by Kwak, S. H., Scholz, T., and Byrnes, R. B., Naval Postgraduate School, Monterey, California, 24 May 1993.

[Ref. 8] Yourdon, E., *Modern Structured Analysis*, Yourdon Press, Englewood Cliffs, New Jersey, 1989.

[Ref. 9] Byrnes, R. B., The Rational Behavior Model: *A Multi-Paradigm, Tri-Level Software Architecture for the Control of Autonomous Vehicles*, Dissertation, Naval Postgraduate School, Monterey, California, 1993.

[Ref. 10] Byrnes, R. B., McPherson, D. L., Kwak, S. H., McGhee, R. B., and Nelson, M.L., *An Experimental Comparison of Hierarchical and Subsumption Software Architectures for Control of an Autonomous Underwater Vehicle*, Proceedings of 1992 IEEE Symposium on Autonomous Underwater Vehicle Technology, pp. 235-241, Washington, D.C., June 2-3, 1992.

[Ref. 11] Giarratano, J. C., *CLIPS User's Guide*, Software Technology Branch, NASA Lyndon B. Johnson Space Center, Houston, Texas, 1991.

[Ref. 12] Naval Postgraduate School - Department of Computer Science Technical Report NPS CS-92-003, *Rational Behavior Model: A Tri-Level Multiple Paradigm Architecture for Robot Vehicle Control Software*, by Kwak, S. H., McGhee, R. B., and Bihari, T. E., Naval Postgraduate School, Monterey, California, 1992.

[Ref. 13] Bellingham, J.G., Consi, T.R., *State Configured Layered Control*, Proceedings of the IARP 1st Workshop on Mobile Robots for Subsea Environments, pp.75-80, Monterey, CA., October 23-26, 1990.

[Ref. 14] Kwak, S.H., McGhee, R.B., *Rule-Based Motion Coordination for the Adaptive Suspension Vehicle*, Interim Scientific Report NPS52-88-011, May 1988.

[Ref. 15] Kwak, S.H., McGhee, R.B., *Rule-Based Motion Coordination for the Adaptive Suspension Vehicle on Ternary-Type Terrain*, Technical Report NPSCS-91-006, Naval Postgraduate School, Monterey, CA., 1990.

[Ref. 16] Pugh, D.R., et al., *Technical Description of the Adaptive Suspension Vehicle*, The International Journal of Robotics Research, Vol. 9, No. 2, pp. 24-42, 1990.

[Ref. 17] Homem de Mello, L.S., and Sanderson, A.C., *Representations of Mechanical Assembly Sequences*, IEEE Transactions on Robotics and Automation, Vol. 7, No.2, April 1991, pp. 211-227.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
Professor Dr. Ted Lewis Chairman, Department of Computer Science Code CS/Lt Naval Postgraduate School Monterey, CA 93943	1
Professor Dr. Se-Hung Kwak Department of Computer Science, Code CS/Kw Naval Postgraduate School Monterey, CA 93943	3
Professor Dr. Yuh-Jeng Lee Department of Computer Science, Code CS/Le Naval Postgraduate School Monterey, CA 93943	1
Professor Dr. Robert B. McGhee Department of Computer Science, Code CS/Mz Naval Postgraduate School Monterey, CA 93943	1
Professor Dr. Anthony J. Healey Mechanical Engineering Department, Code ME/Hy Naval Postgraduate School Monterey, CA 93943	1
Professor Dr. Karl Mosler Professur für Statistik und Quantitative Methoden Fachbereich Wirtschafts- und Organisationswissenschaften Universität der Bundeswehr Hamburg Holstenhofweg 85 22043 Hamburg, Germany	1

MAJ Dr. Ronald B. Byrnes, USA
Software Technology Branch, ARL
Georgia Institute of Technology
115 O'Keefe Building
Atlanta, GA 30332-0800 1

LCDR Donald P. Brutzman, USN
Operations Research Department, Code OR/Br
Naval Postgraduate School
Monterey, CA 93943 1

LCDR Thomas Scholz, German Navy
Kommandant "S68 SEEADLER"
2. Schnellbootgeschwader
Hafenstr. 2
24376 Kappeln, Germany 1