A Pragmatic Formal Method (PFM) for Computer System Definition and Execution

Dr. Stephanie White
Grumman Corporate Research Center
Mail Stop A08-35
Bethpage, New York, 11714

Abstract: The PFM Process (WHIT 87) produces a model of a to-be-built target system, provides analysis of model quality, and simulates system operation. PFM incorporates a solution to one of the major problems with Petri nets: a static Petri net structure of a large system is difficult to define. A formal extension of Petri nets called PFM-nets, which are nets of interacting PFM functions, are used to simulate system operation and support the development of a rapid prototype. PFM-nets can be generated (in theory) automatically from a PFM model of a target system. This paper discusses the PFM model and model execution. For clarity, objects in the formal PFM model are capitalized.

PFM FORMAL MODEL

A PFM model is a definition of a target system that is to be built; it is a statement of the functional requirements. The PFM model extends the model developed by Parnas and others on the Software Cost Reduction (SCR) Project at Naval Research Laboratory (HENI 81). The SCR model consists of data templates (see Fig. 1), function tables (see Fig. 2), and mode transition tables. These templates and tables are easy to create and understand but have underlying formalisms. Function tables are extended state machines, where machines operate concurrently with and have state knowledge about other machines.

PFM possesses all the attributes of the SCR methods, using the same templates and tables for model definition. In addition, PFM uses a formal Entity-Relationship (ER) model which supports automation. Function Tables (extended state machines) are mapped to Petri-nets to support simulation and analysis for liveness and reachability. PFM has additional analysis capabilities, presented in (WHIT 87). The ER concept is used to formally define the objects in the PFM model and relationships between these objects. The ER definition can be used as a logical database design for storing target system models and is needed for method automation. The ER model can also be used by the analyst to determine the proper use and power of the PFM method.

In PFM Entity-Relationship diagrams, rectangles represent objects; circles represent two-part relationships; triangles represent three-part relationships; and squares represent four-part relationships. The relationships are only specified in one direction to reduce diagram complexity, but the relationships are complementary. Upper case letters are used for PFM entity names, and relationship numbers in parentheses refer to relationships on diagrams. For example, in Fig. 3, PFM-REQUIREMENTS-MODEL has subpart DEFINED-MODEL, and subpart DERIVED-PFM-NET. These subparts relationships are referred to as (501) and (502) in the figure. The DEFINED-MODEL has subparts OBJECT-ABSTRACTIONS and MODE-CONTROL-ABSTRACTIONS, (see Fig. 4). Both the SYSTEM and ENVIRONMENT contain OBJECT-ABSTRACTIONS, which are composed of DATA, DEMAND-FUNCTIONS, and PERIODIC-FUNCTIONS (see Fig. 5).

PFM Function Relations are defined in Fig. 6. FUNCTIONS are activated/deactivated by EVENTS when CONDITIONS are true. These CONDITION guards make the definition more succinct as all paths do not have to be described. The specification is a partial order rather than linear or branching order. These three-part relationships are defined in the figure by (210) and (211). When relationships are three-part or four-part, a special syntax is used. Relationship (210) reads V ACTIVATED BY VI WHEN I. The roman numerals I, V, VI refer to objects that take part in the relationship. The roman numerals appear in the upper right-hand corner of rectangles. A full set of ER diagrams for PFM are defined in (WHIT 87).

PFM-NETS

In this section, PFM-nets are formally defined and a process is created for transforming a PFM DEFINED-MODEL of a target system to a PFM-net. If techniques developed in this section were automated, a PFM-net could become computer generated from the target system model.

PFM-nets are comparable to Petri nets with inhibitor arcs, and have commensurate power to perform system simulation and analysis. A rapid prototype can also be developed from the PFM DEFINED-MODEL by coding the operational aspects of ACTIONS associated with significant EVENTS and FUNCTIONS. The rapid prototype consists of the PFM-net generated from the uncoded portion of the DEFINED-MODEL, interacting with the coded portion. The PFM rapid prototyping process is effective, according to Bruno and Marchetto's criteria.
Output Data Item: Valve Control Command

Acronym: //VALVE//

Hardware: Furnace

Description: //VALVE//= $Open$ directs the furnace to open the fuel valve permitting fuel flow into the combustion chamber. //VALVE//= $Close$ directs the furnace to close the fuel valve, shutting off the flow of fuel.

Characteristics of Value

Value Encoding: $Open$ , $Close$

Instruction Sequence:

Data Representation:

Timing Characteristics:

Comments:

Fig. 1 SCR Data Template

Demand Function Name: Switch Valve Control Open/Closed

Modes in which function required:

Output Data Item: //VALVE//

Function Request and Output Description:

EVENTS

@T(/RPM//= $Yes$) WHEN
(l(projected temp)<lower limit)
AND/MSW//= $Heat$

@T(/MSW//= $Off$)
(T(combustion fail))
@T(/FUEL//= $No$)
@T(/RPM//= $No$

ACTION //VALVE//= $Open$

@T(/投影 temp)>upper limit)

//VALVE//= $Closed$

Fig. 2 SCR Function Template Without Modes

FIGURE 3 PFM REQUIREMENTS MODEL, FORMAL DEFINITION

FIGURE 4 PFM DEFINED MODEL, FORMAL DEFINITION

FIGURE 5 PFM OBJECT ABSTRACTION, FORMAL DEFINITION
FIGURE 6  PFM FUNCTION RELATIONS, FORMAL DESCRIPTION
"Rapid prototyping is an alternative paradigm to the conventional...software life cycle. However, the prototyping paradigm is ineffective if it is not supported by a development environment that provides an easy derivation of prototypes from formal specifications and makes the implementation process partially automated" [BRUN 86].

GENERATION OF A PFM-NET

A PFM-net is a 7-tuple (P, T, E, M₀, K, W, F) where:

P: A set of PLACES. Each PLACE represents a CONDITION or an EVENT.

T: A set of TRANSITIONS. Each TRANSITION defines a rule for state change in the model.

E: A set of directed EDGES (arcs) that connect PLACES to TRANSITIONS, or TRANSITIONS to PLACES.

M₀: An initial MARKING (distribution of tokens).

K: The maximum capacity of each place for holding tokens.

W: The weight of each edge.

F: A set of FUNCTIONS. Each FUNCTION (Fₚ) contains a subset of the PLACES, TRANSITIONS, and EDGES in the net. Each PLACE (Pᵢ) and TRANSITION (Tᵢ) and EDGE (Eᵢ) in the net is a member of at most one FUNCTION.

Definition of primary and secondary PLACES and TRANSITIONS: A PFM-net is generated by developing a net for each FUNCTION. Each PFM FUNCTION has one and only one OUTPUT variable. Each distinguishable value for the OUTPUT is a CONDITION and a primary PLACE of that FUNCTION. A TRANSITION is defined as primary for Fₓ, if there is an EDGE (arc) from a primary place of Fₓ to the TRANSITION, and there is an EDGE (arc) from the TRANSITION to a primary PLACE of Fₓ. A secondary PLACE is generated for each ACTION caused by the FUNCTION. All transitions that are not primary are secondary.

Example: We use a home heating system problem to illustrate Primary and Secondary places. The problem statement appears in the Appendix. In the function, Set Valve Commands, there are two primary PLACES, P₁ for valve.open and P₂ for valve.closed, and there are five primary TRANSITIONS (t₁, t₂, t₃, t₄, t₅) as shown in Fig. 7.

Valve.closed transitions to valve.open if:

(l₁) Becoming-True (motor-rpm.up-to-speed

WHEN NOT (proj-temp.hot)

Valve.open transitions to valve.closed if:

(t₂) Becoming-True (proj-temp.hot) OR
(t₃) Becoming-True (master-sw.off) OR
(t₄) Becoming-True (fuel-status.failure) OR
(t₅) Becoming-True (combustion-status failure)

The ACTIONS (also called EVENTS) caused by the function Set Valve Commands are Becoming-True (valve.closed) and Becoming-True (valve.open). Each TRANSITION which is connected to the PLACE valve.closed is also connected to the secondary PLACE Becoming-True (valve.closed). Each TRANSITION which is connected to the PLACE valve.open is connected to the secondary PLACE Becoming-True (valve.open) as shown in Fig. 8. When the FUNCTION nets are connected, those secondary places that do not affect a TRANSITION in another FUNCTION are discarded.

FIRING PFM-NETS

For each model a time unit, called a time step, is defined to represent the minimum measurable time interval. PFM-nets are fired once per time step, and every TRANSITION that is enabled at the beginning of the time step is fired, so that concurrent FUNCTIONS in the PFM model are fired in parallel. If a variable changes state during the time step, it remains in the new state for at least the remainder of that time step.

In PFM-nets, TRANSITIONS are enabled if the number of tokens in each input place equals the weight of the edge entering the TRANSITION; as is the case in Petri nets. However, the result of firing PFM-net TRANSITIONS is somewhat different from the result of firing Petri net TRANSITIONS, since tokens are not always deleted from an input PLACE to a TRANSITION, when that TRANSITION is fired. The following describes this in detail:

FUNCTION States and Primary PLACES: The OUTPUT of each activated PFM FUNCTION should always be in one and only one State; thus, one and only one primary PLACE should be marked. A token should be removed from a PLACE if the State changes, but not be removed if the State remains the same.

States (CONDITIONS) used as guards: If Sₓ₁ is a State (primary PLACE) of function Fₓ and Sₓ₁ is a Guard that affects State change in a different FUNCTION Fᵧ, the token should remain in Sₓ₁ when a TRANSITION in Fₓ is fired as the TRANSITION in Fᵧ does not affect the State of Fₓ.

EVENTS: When a TRANSITION with an EVENT in an input PLACE fires, the EVENT should lose its token
FIGURE 7  SET VALVE COMMANDS, HOME HEATING SYSTEM
PRIMARY PLACES AND TRANSITIONS IN PFM-NET

FIGURE 8  SET VALVE COMMANDS, HOME HEATING SYSTEM
SECONDARY PLACES ADDED FOR EVENTS IN PFM-NET
since the EVENT is only true for one time step. When a TRANSITION with an EVENT in an output place fires, the EVENT should receive a token since the EVENT is now true for one time step.

Firing rules: In order for PFM-nets to perform as previously described, the following rules are used to govern the firing of PFM-nets for the simple case where edges have a weight of one. The rules can be extended to cover the case where the weight of edges is greater than one. The following notation is used in the definition of PFM firing rules:

- \( P_i \): place,
- \( t_j \): transition,
- \( F \): function
- \( I(t_j) \): an input place of \( t_j \),
- \( O(t_j) \): an output place of \( t_j \).

Let \( p_i \in F_x \) (where \( \epsilon \) indicates set membership) if and only if \( p_i \) is a primary PLACE of the FUNCTION \( F_x \).

Let \( t_j \in F_x \) if and only if \( t_j \) is a primary TRANSITION of the FUNCTION \( F_x \).

Rule (1): If \( p_i = I(t_j) \) and \( p_k = O(t_j) \) and \( (p_i, t_j, p_k) \in F_x \), then a token is removed from \( p_i \) and placed on \( p_k \) when \( t_j \) fires.

Rule (2): If \( p_i = I(t_j) \) where \( p_i \in F_x \) and \( t_j \in F_x \), a token is not removed from \( p_i \) when \( t_j \) fires.

Rule (3): If \( p_i = I(t_j) \) and \( p_i \) is not a primary PLACE of any FUNCTION, \( p_i \) loses a token when \( t_j \) fires.

Rule (4): If \( p_i = O(t_j) \) and \( p_i \) is not a primary place of any function, \( p_i \) gains a token when \( t_j \) fires.

Example: The rules will be illustrated using a specific PFM-net as an example. In the net in Fig. 9, \( P_{11}, P_{12} \) and \( t_{11}, t_{12} \) are primary PLACES and TRANSITIONS of \( F_1 \); \( P_{21}, P_{22}, t_{21}, t_{22} \) are primary PLACES and TRANSITIONS of \( F_2 \); \( P_{11}, P_{22} \) are secondary PLACES representing EVENTS.

The net in Fig. 9 has initial marking \( M_0 \). The PLACES in the figure which represent the initial State contain dots representing tokens. They are also shaded. For \( (P_{11}, P_{12}, P_{11}, P_{21}, P_{22}, P_{22}) \), \( M_0 = (0, 1, 0, 0, 1, 0) \).

Firing \( t_{12} \) results in MARKING \( M_1 \) where:

\[ M_1 = (1, 0, 1, 0, 1, 0) \]

Explanation: Since \( (P_{12}, t_{12}, P_{11}) \in F_1 \), a token is moved from \( P_{12} \) to \( P_{11} \) (rule 1). Since \( P_{21} \) is not a primary PLACE, it receives a token when \( t_{12} \) fires by (Rule 4). A token is not removed from \( P_{22} \) since \( P_{22} \in F_2 \) and \( t_{12} \notin F_2 \) (Rule 2).

Firing \( t_{21} \) results in MARKING \( M_2 \), where:

\[ M_2 = (1, 0, 0, 1, 0, 0) \]

Explanation: \( P_{11} \) keeps its token since \( t_{11} \) didn't fire. \( P_{21} \) loses its token since \( t_{21} \) fired and \( P_{21} \) isn't a primary PLACE (Rule 3). A token is moved from \( P_{22} \) to \( P_{21} \) since \( (P_{22}, t_{21}, P_{21}) \in F_2 \) (Rule 1).

Firing \( t_{22} \) results in MARKING \( M_3 \) where:

\[ M_3 = (1, 0, 0, 0, 1, 1) \]

Explanation: Since \( (P_{21}, t_{22}, P_{22}) \in F_2 \), a token is moved from \( P_{21} \) to \( P_{22} \) (Rule 1). A token is added to \( P_{22} \) since it is an output place of \( t_{22} \) and is not a primary PLACE (Rule 4).

Firing \( t_{11} \) results in MARKING \( M_4 \) where:

\[ M_4 = M_0 = (0, 1, 0, 0, 1, 0) \]

Explanation: Since \( (P_{11}, t_{11}, P_{12}) \in F_1 \), a token is moved from \( P_{11} \) to \( P_{12} \) (Rule 1). A token is removed from \( P_{22} \) (Rule 3). The place \( P_{22} \) is not an input place of \( t_{11} \) so it is not affected.
WHEN, WHILE, AND EVENTS WITH MULTIPLE CONDITIONS

In a discrete simulation, two EVENTS can occur in the same time step. It is necessary to specify in the DEFINED-MODEL whether one EVENT must occur before the other or whether they may occur in parallel. If the EVENT which causes A to become true must occur prior to the EVENT that causes B to become true, the word WHILE is used:

e.g., Becoming-True(B) WHILE A --> ACTION

If the EVENT which causes A to become true can occur before or at the same time as the EVENT which causes B to become true, the word WHEN is used:

e.g., Becoming-True(B) WHEN A --> ACTION

The following PFM expressions define temporal relations.

(1) \[\text{Boolean-Expression (CONDITIONi)}\] implies over some interval of time, a statement involving a set of CONDITION operands and a set of Boolean operators (OR, AND, NOT) is true.

(2) \[\text{Becoming-True (CONDITIONa)} \text{ WHEN } \text{Boolean-Expression (CONDITIONi)}\] implies CONDITIONa and \[\text{Boolean-Expression (CONDITIONi)}\] could become true at the same instant or \[\text{Boolean-Expression (CONDITIONi)}\] could become true prior to \[\text{Becoming-True (CONDITIONa)}\]. \[\text{Boolean-Expression (CONDITIONi)}\] cannot become true after CONDITIONa.

(3) \[\text{Becoming-True (CONDITIONa)} \text{ WHILE } \text{Boolean-Expression (CONDITIONi)}\] implies \[\text{Boolean-Expression (CONDITIONi)}\] is true prior to \[\text{Becoming True(CONDITIONa)}\].

(4) Becoming-True \[\text{Boolean-Expression (CONDITIONi)}\] implies at some instant the Boolean function has become true.

Each of the expressions (2 through 4) requires an implementation in PFM-nets. Expression (1), \[\text{Boolean-Expression (CONDITIONi)}\] does not cause a TRANSITION but is a term in each of the expressions (2 through 4).

Implementing "WHEN": In expression (2), \[\text{Becoming-True(CONDITIONa)} \text{ WHEN } \text{Boolean-Expression (CONDITIONi)}\], the condition could occur before the EVENT or the EVENT and CONDITION could occur at the same time. Because the EVENT is only true for one time step, Becoming-True \[\text{Boolean-Expression CONDITIONi)}\] = \[\text{Becoming-True(CONDITIONa)} \text{ AND } \text{Boolean-Expression (CONDITIONi)}\]. An example is shown in Fig. 10. The TRANSITION is fired if the required number of tokens are in all its normal input PLACES and zero tokens are in all its inhibitor input PLACES. (An inhibitor place is connected to the transition that it inhibits by an edge with a small circle. A token in the inhibitor PLACE prevents the TRANSITION from firing.)

Implementing "WHILE": To implement expression (3), \[\text{Becoming-True(CONDITIONa)} \text{ WHILE } \text{Boolean-Expression (CONDITIONi)}\], it is necessary to make sure that the CONDITION \[\text{Boolean-Expression (CONDITIONi)}\] is true prior to the event \[\text{Becoming-True (CONDITIONa)}\]. As an example, the case where \[\text{Boolean-Expression (CONDITIONi)}\] = \[\text{COND1 AND COND2}\] is considered, and depicted in Fig. 11.

\[\text{Becoming-True(CONDITIONa)} \text{ WHILE } \text{Becoming-True(CONDITIONa)} \text{ AND } \text{COND1 AND COND2 AND NOT [Becoming-True(COND1) OR Becoming-True(COND2)]}\]

Inhibitor Arcs, associated with the EVENTS Becoming-True(COND1) and Becoming-True(COND2), insure that the TRANSITION does not fire if either of these EVENTS occurs at the same time as Becoming-True(CONDITIONa).

Implementing EVENTS with multiple CONDITIONS: An EVENT with multiple CONDITIONS is a Condition Expression preceded by the words "Becoming-True," or "Becoming-False." (A Condition Expression is an expression in which the operands are CONDITIONS, and the operators are the boolean connectives OR, AND, NOT). Events with multiple CONDITIONS can be treated as combinations of two basic types.

Let COND1, COND2 be two conditions; then,

(1) \[\text{Becoming-True(COND1 OR COND2)} = \text{Becoming-True(COND1) AND NOT Becoming-True(COND2)} \text{ OR } \text{Becoming-True(COND2) AND NOT Becoming-True (COND1)} \text{ OR } \text{Becoming-True(COND1) AND Becoming-True(COND2)}\] where COND1, COND2 are two CONDITIONS.

(2) \[\text{Becoming-True(COND1 AND COND2)} = \text{Becoming-True (COND1) WHILE COND2} \text{ OR } \text{Becoming-True (COND2) WHILE COND1} \text{ OR } \text{Becoming-True(COND1) AND Becoming-True(COND2)}\]

Expression (1) is implemented in Fig. 12. The implementation of expression (2) is slightly more complex, but would be handled similarly.
BECOMING TRUE CONDITION A

BEC. TRUE MOTOR-RPM UP-TO-SPEED

WHEN CONDITION a

FIGURE 10 IMPLEMENTING WHEN IN A PFM-NET

FIGURE 11 IMPLEMENTING WHILE IN A PFM-NET

FIGURE 12 IMPLEMENTING EVENTS WITH MULTIPLE CONDITIONS
PFM-NETS COMPARED WITH OTHER PETRI NET EXTENSIONS

A number of researchers have extended Petri nets because they have found them too simple to model real systems. The extensions have made the nets more powerful for modeling (simulation), but less powerful for analysis (e.g., determining reachability or deadlock). This research is summarized by Peterson (PETE 81). Patil extended petri nets to include constraints (PATI 70). A constraint is a set of places. A transition can fire if and only if the resulting marking does not include all the places in the constraint. Baer extended Petri nets by adding decision power to a transition (BAER 73). If a transition is marked as disjunctive, only one of the edges leaving the transition carries tokens, rather than all edges as in an unextended Petri net. Merlin associated time intervals with a transition (MERL 76). The transition must wait a certain period of time after it is enabled before it can fire, and has a deadline within which it must fire. Hack associated priorities with a transition (HACK 75). If two transitions are enabled, the one with the highest priority is fired first. Each of the above extensions changes the firing rules and creates a more powerful model. Each of the above models has the capability to test for an unmarked place. This capability, which is called zero testing, allows a Petri net to simulate a Turing machine, and thus model any system (PETE 81). The addition of Inhibitor Arcs to a Petri net is the simplest extension, which adds the capability to perform zero testing.

Since PFM-nets contain Inhibitor Arcs, PFM-nets are as powerful as other extensions. PFM-nets have other advantages. The analyst does not have to define the net. If techniques defined in this section were automated, a PFM-net could be computer generated from the DEFINED-MODEL, which is modular and relatively simple to construct (WHIT 87).

SIMULATION & PROTOTYPING WITH PFM-NETS

In this section, PFM-nets will be used to demonstrate PFM simulation capability. The simulation shows the FUNCTION that determines the projected temperature, interacting with the FUNCTIONS which control the motor and valve.

An initial token MARKING must be chosen in order to simulate SYSTEM interaction. For many state variables the initial state will already exist in the DEFINED-MODEL. The net is fired continuously until the desired number of time steps are complete, a chosen MARKING is reached, or the simulation terminates (no TRANSITION can be fired).

The PFM FUNCTIONS to be simulated are defined in Fig. 13. INPUTS, OUTPUTS, and associated hardware interfaces are defined in associated Data Templates. The PFM-net generated from this model is shown in Fig.14. The net is a direct derivation of the DEFINED-MODEL, and contains the three functions: Control Valve (F₁), Control Motor (F₂), and Control Projected-Temp (F₃). These FUNCTIONS appear as cycles of PLACES and TRANSITIONS in the net. The EVENTS which cause TRANSITION in the DEFINED-MODEL, (e.g., Becoming-True (projected-temp.cold)) are also shown in the net, connected by means of a directed arc (EDGE) to the TRANSITION which they trigger. Each of these EVENTS is caused by a FUNCTION in the net, and thus is also connected to a primary TRANSITION of the generating FUNCTION by a directed arc (EDGE). The net in Fig. 14 depicts the initial State of the model. The furnace is off. The valve is closed, the motor is off and the projected house temperature is neither "hot" nor "cold".

Code or user interaction is required to trigger a TRANSITION (t₃₄) causing projected house temperature to change from "neither" to "cold", and generating a "cold" event, Becoming-True (projected-temp.cold) (see Fig. 15, sheet 1 of 4). The tokens in the EVENT PLACE, Becoming-True (projected-temp.cold), and the CONDITION PLACE (motor.off) cause a TRANSITION (t₃₁) in F₂ to fire, changing the motor State from off to on and generating a motor.on EVENT, Becoming-True (motor.on) (see Fig. 15, sheet 2 of 4). Code, user interaction, or a PFM-net of the ENVIRONMENT is needed to signal that the motor is up to speed, by adding a token to the PLACE, Becoming-True (motor-rpm.up-to-speed) (see Fig. 15, sheet 3 of 4). The motor on EVENT, Becoming-True (motor.on) has signalled the hardware motor and thus has lost its token. The PLACES for the motor up to speed EVENT, Becoming-True (motor-rpm.up-to-speed), and the valve closed CONDITION, (valve.closed) contain tokens. This triggers a TRANSITION (t₃₅) in F₁ to fire, changing the valve State from closed to open, and generating a valve open EVENT, Becoming-True (valve.open) (see Fig. 15, sheet 4 of 4). The furnace controller is now in the State valve.open, motor.on, projected-temp.cold, and has turned the furnace on.

A SYSTEM can rapidly be prototyped, or the ENVIRONMENT can be modeled, by using code to replace FUNCTIONS (a set of primary PLACES and TRANSITIONS) or EVENTS in a PFM-net. Code associated with events can send messages to a terminal screen or another system when the EVENT receives a token. A program which prototypes a FUNCTION is activated by tokens in the net, and adds tokens to the net, to interact with the remainder of the net.
FIGURE 13 DEFINED PFM MODEL USED IN SIMULATION

FIGURE 14 PFM-NET GENERATED FROM DEFINED MODEL
CONCLUSION

PFM formalizes, synthesizes, and improves methods proven practical on large embedded systems; see (WHIT 87) for a comparative analysis of eight commonly used methods. The model is relatively easy to construct. As in the SCR method, data is entered by the analyst using templates and state transition tables which support logical analysis and completeness checking. The SCR method is formalized via an Entity-Relationship model that supports tool development. Translation from state transition tables to Petri-like nets supports model execution and allows the analyst to solve many liveness and reachability problems. Prototype support tools are needed to test PFM theory.

REFERENCES


APPENDIX

Home heating system specification. A computer system interacting with a temperature sensing device and oil furnace is used to control the temperature of a house. The temperature sensor compares the ambient temperature in the house with the set temperature and sends the difference, the temperature error, to the controller. For purposes of comfort and furnace efficiency, the total change of temperature allowed in the house will be four degrees. The oil furnace has a motor which drives a fan to supply combustion air, and also drives a fuel pump.

When the house gets too cold, the software controller sends a signal activating the motor. When the motor reaches normal operating speed, the controller receives a signal and responds by sending two signals activating the ignition, and opening the oil valve. The fuel is ignited at this time and the furnace begins to heat the water. Fuel and optical combustion sensors signal the controller if abnormalities occur.

The furnace is alternately activated and deactivated by the controller to maintain the temperature within the required limits. When the furnace is deactivated, first the oil valve is closed, and five seconds later to allow for the valve lag time, the motor and ignition are deactivated. There is a three-second lag time before the motor stops.