A Simulation Environment for Multimodeling*

Paul A. Fishwick
University of Florida

Abstract

Large scale systems are typically quite difficult to model. Hierarchical decomposition has proven to be one successful method in managing model complexity, by refining model components into models of the same type as the lumped model. Many systems, however, cannot be modeled using this approach since each abstraction level is best defined using a different modeling technique. We present a *multimodel* approach which overcomes this limitation, and we illustrate the technique using a fairly simple scenario: boiling water. State and phase trajectories are presented along with an implementation using the *SimPack* simulation toolkit. Multimodeling has provided us with a mechanism for building models that are capable of producing answers over a wide range of fidelity. **[Key Words: Multimodeling, Process Abstraction, Combined Simulation, Phase Transitions, Simulation Environment]**

1 Introduction

Modeling "in the large" has always been a central topic in computer simulation. Approaches in past work have concentrated on using functional coupling and hierarchical decomposition to alleviate the problems associated with complex, cumbersome models. These techniques have been partially successful but they have lead to the manifestation of simulation languages and methodologies that grow to fit the needs of new applications and extensions. Our approach —multimodeling— advocates the use of existing, well known model types (such as Petri networks, automata, Markov models and block models) within the same model structure: a multimodel.

Multimodeling [9, 7] is the process of engineering a model by combining different model types to form an abstraction network or hierarchy. If we begin to understand a physical system by creating a model, we often find that the model is too limited; the model will answer only a very limited set of questions about system behavior. It is necessary, then, to create many models and link them together, thereby maintaining a multi-level view of a system while permitting an analyst to observe system output at several abstraction levels. Even apparently simple physical systems can exhibit remarkly complex behavior in terms of all models necessary to completely define the system. Consider a pot of boiling water

*The author is grateful for partial funding of this research through a grant from the Florida High Technology Council. Author's Address: Paul A. Fishwick, Dept. of Computer and Information Sciences, University of Florida, Bldg. CSE, Room 301, Gainesville, FL 32611

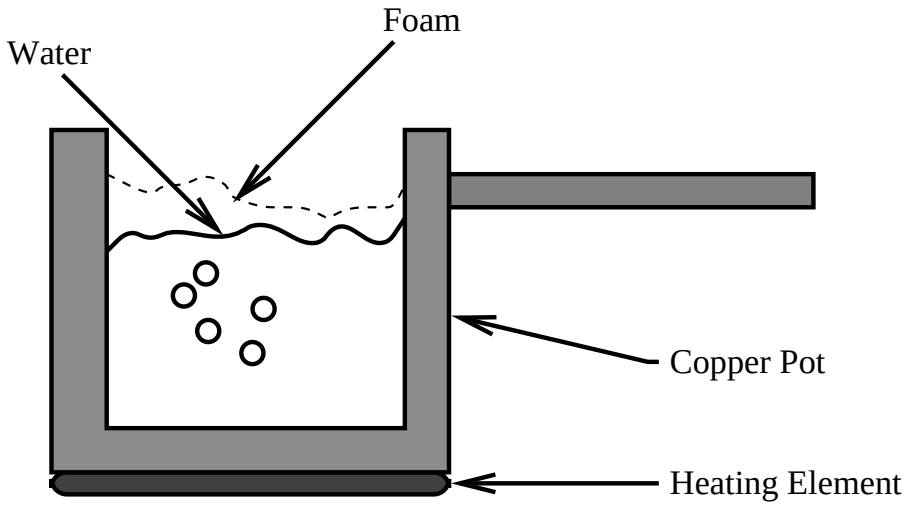


Figure 1: A pot of boiling water.

(see fig. 1). To model this system, we must first determine what questions we are likely to ask of the system. Only then, can we proceed with creating an *a priori* assumption as to system structure. If we are concentrating only on the temperature history of the water then we might consider a simple linear system structure; however, we will run into trouble as soon as we want to ask questions relating to what happens to the system in an exception condition (overflow, underflow) or at a higher level of abstraction (heating,cooling). A significant problem with single level models is that, not only do they attempt to answer a narrow set of questions, but the models are made to be used only by a select number of people. It would be convenient if we could create models that can provide answers to children in an educational setting, as well as to scientists. To enlarge our class of answerable questions, we must combine models together in some seamless fashion.

Recent simulation methodology has developed concepts to model complex systems over multiple levels of abstraction [3, 4, 5]. Ören [14] defined a concept of *multimodel* to formalize models containing several submodels, only one of which is put into effect at any given time. The idea of a multimodel has its roots within the work in *combined* simulation modeling. Combined modeling has traditionally referred to a integration of discrete event and continuous modeling within the same system description. Pritsker [19, 20] first implemented combined modeling in the GASP modeling language. Cellier [1] developed an approach to combined continuous / discrete event models implemented in a GASP language extension. Praehofer [18] extended the Discrete Event System Specification (DEVS) [25] to provide a formalism and a simulation environment for specifying combined continuous / discrete event models.

The previous research in combined modeling has fueled the study of multimodels, but our approach to modeling is significantly different in that we build models for large-scale systems by employing those modeling techniques that have been proven useful for a specific abstraction level. Instead of using only Petri nets or only automata, and then extending these types (i.e., colored Petri nets, augmented automata), we mix and match types to form a multimodel. We do not subscribe to the idea that, in order to model ever increasingly complicated systems, one must continue to extend a singular modeling

approach. Instead, a blending of approaches is warranted. The multimodel approach uses SimPack [6] tools depending on the model types present in the abstraction hierarchy. *SimPack* is a collection of C and C++ libraries and executable programs for computer simulation. In this collection, several different simulation algorithms are supported including discrete event simulation, continuous simulation and multimodel simulation. The purpose of the *SimPack* toolkit is to provide the user with a set of utilities that illustrate the basics of building a working simulation from a model description. Most simulation packages cover one of two areas: discrete event or continuous. Discrete event methods cater to those performing modeling of queuing networks, flexible manufacturing systems and inventory practices. Continuous methods are normally associated with block diagram methods for control and system engineering. Some available software can perform both types of simulation; however, bulk support is usually available in only one form. Classic discrete event simulation languages such as GPSS [21], SLAM [20], SIMSCRIPT [12] and SIMAN [16] have been used extensively over the past decade, while continuous languages such as CSMP, DESIRE [11] and continuous system language derivatives provide adequate support for modeling continuous systems [2] in the form of block models.

As systems become large and complex, the analyst will require simulation software that can support a wide variety of model types. One solution to modeling complex systems in simulation languages is to convert all models into that language. Our approach is quite different — we recognize that, for instance, resource contention is best modeled with Petri nets; queuing problems are best modeled with queuing graphs and some continuous systems are best modeled with engineering block diagrams. Therefore, our approach is to provide a set of C and C++ tools that accommodates a direct translation from these unique graphing approaches into callable routines; we do not force the user to think in terms of a single overall language for all simulation applications. Instead, we believe that most systems will contain model components whose types are different. The perceived need to have an "all in one" simulation language does not match most real world problems where a set of well-tested model types has developed naturally. The *SimPack* emphasis on "diversity" in modeling makes it a natural candidate in which to construct multimodels.

We begin by introducing an example of boiling water. Section 2 outlines the use of three automata levels, one continuous block model level and a small Petri net, to model the boiling water process. In section 3, we formalize the algorithm associated with executing multimodels using DEVS. Examples drawn from computer experiments on the boiling water multimodel are illustrated in section 4, and section 5 presents conclusions and a future direction for this work.

2 Example System: Boiling Water

2.1 Overview

Consider a pot of boiling water on a stovetop electric heating element. Initially, the pot is filled to some predetermined level with water. A small amount of detergent is added to simulate the foaming activity that occurs naturally when boiling certain foods. This system has one input or control – the temperature knob. The knob is considered to be in

one of two states: on or off (on is $190°C$; off is α - ambient temperature). We make the following assumptions in connection with this physical system:

1. The input (knob turning) can change at any time. The input trajectories are piecewise continuous with two possible values (ON,OFF).

2. The liquid level (height) does not increase until the liquid starts to boil.

3. When the liquid starts to boil, a layer of foam increases in height until it either overflows the pot or the knob is turned off.

4. The liquid level decreases during the boiling and overflow phases only.

To create a mathematical model, we must start with data and expert knowledge about the domain. If enough data can be gathered in a cost effective way then our model engineering process will be simplified since we will not have to rely solely on heuristics to identify the model. By analyzing a pot of boiling water we may derive simple causal models whose individual transitions may be $knob\_on \Rightarrow water\_getting\_hotter$ or $water\_getting\_hotter \Rightarrow water\_boiling$. An important facet of system modeling is that we choose certain modeling methods that require a categorization of informally specified system components. Key components of any system model are *input*, *output*, *state*, *event*, *time* and *parameter*. Different modeling methods include these components in different ways. For instance, an FSA focuses on state-to-state transitions with input being labeled on each arc. A dataflow model, on the other hand, focuses on the transfer function between input and output.

To create models, we must review the purpose and goals of the simulation. We choose to concentrate on temporal phase information so that we can answer questions of the sort "What can happen immediately after the pot boils?" or "How long does it take for the water to cool to room temperature if the knob is turned off when $T = 90°C$?" In the following sections, we discuss a stepwise refinement process to create multiple models of the boiling water system. Our refinement procedure is shown in figure 2. The hierarchy is representational in nature. We define a hierarchical organization since future question-answering about the system will be facilitated by maintaining such a structure. Often, a question may require only the information present on a specific layer regardless of whether some states in that layer *represent* lower level states; a "lumped" state —by itself— can carry information in addition to the default information it carries by representing a sub-model. If we want to construct a second level abstraction of the total system, we take the topmost FSA and include the second level FSA to form FSA-2. Each level represents a more detailed representation of a given state in the layer above.

2.2 Two Homogeneous Refinements

Homogeneous refinement is the refinement of a model to more detailed models of the same type. For instance, consider a printed circuit board. There are many levels each of which can be modeled using the same block formalism. The model is defined as having type "block" just as a model might have type "Petri net" or "compartmental model." The chip level of the PC board model will contain function blocks that are decomposed into

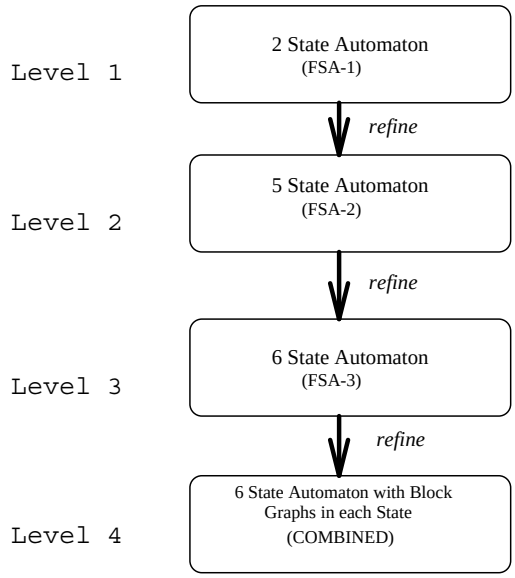


Figure 2: Model refinement procedure.

block networks. In this way, the modeller can build a hierarchy of models without having to represent one model at the lowest abstraction / aggregation level.

Figure 3 displays three levels of finite state automata for the boiling water process. The topmost FSA in fig. 3 displays a simple two state automaton with input. We label this FSA level 1, or "FSA-1." The input is discretized so that the knob control is either ON or OFF. Input can occur at any time, and will facilitate a change in state. A change in input is denoted by $I = i1$ on an arc in fig. 3 defining where the state transition is accomplished when the input becomes $i1$. If the knob is turned on while in state *cold* then the system moves to state *not cold.* When temperature reaches the ambient temperature (denoted by $T = \alpha$) then the system returns to *cold.* The second level includes a detailed representation of state "not Cold." By combining this new FSA with FSA-1, we create FSA-2 (a complete model of the boiling process). FSA-3 is constructed similarly.

Note that a transition condition may sometimes refer to a more detailed state specification than is available at the current level of abstraction. For example, the transition from $Exception$ to $Boiling$ refers to the phase $Overflow$. In model building, such conditions are evidence that further state refinement is necessary; however, the hierarchy is useful not only for model building but also for facilitating question-answering using a variety of abstractions. Alternatively, the modeller may decide to terminate the refinement process leaving the transition non-deterministically specified. The decision whether to continue refinement should be based on the modeling objectives, the accuracy required, and the computational resources available [22, 23].

2.3 Heterogeneous Refinement

Heterogeneous refinement takes homogeneous refinement a step further by loosening the restriction of equivalent model types. For instance, we might have a Petri net at the high abstraction level and we may choose to decompose each transition into a block graph so

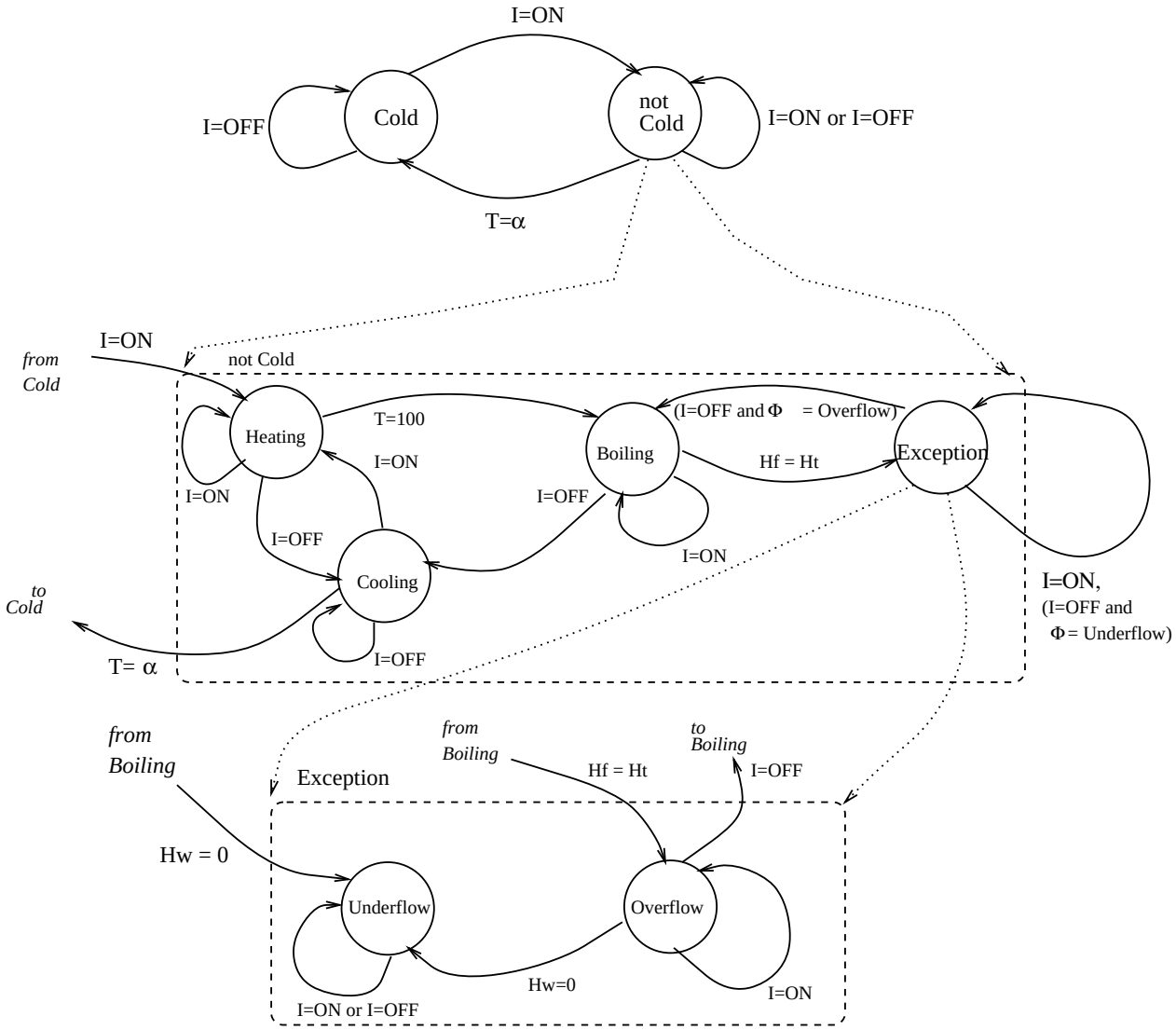


Figure 3: Homogeneous FSA refinement.

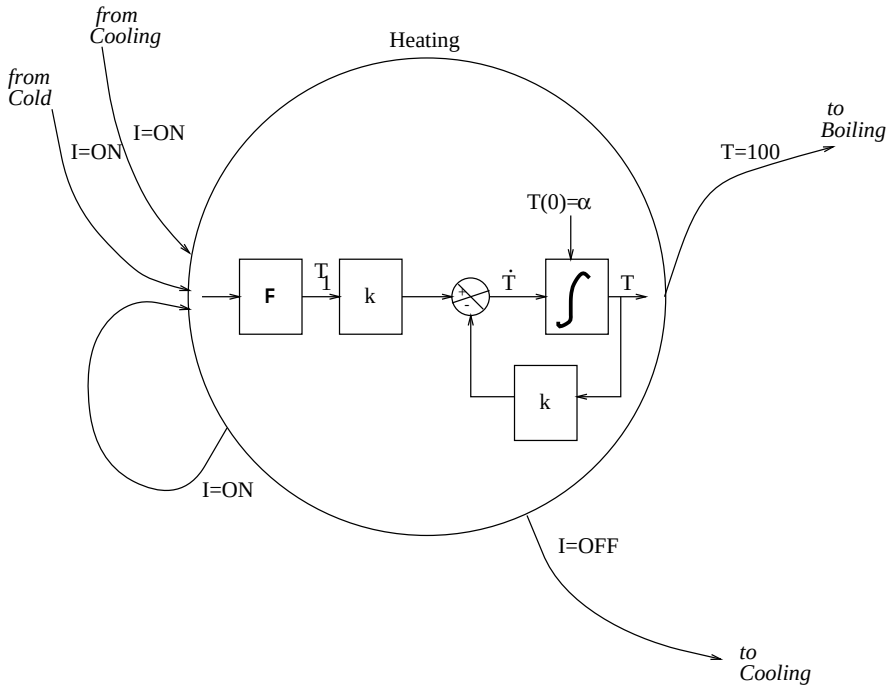


Figure 4: Decomposition of $heating$ state.

that when a transition fires within the Petri net, one may "drop down" into the functional block level. For the last FSA in fig. 3 we choose to represent each state as a continuous model. Specifically, each state will define how three state variables, T (temperature), H_w (height of water), and H_f (height of foam on the top of the water) are updated. Also, the parameter H_t is the height of the pot. In all cases, $H_f \geq H_w$ and $H_w, H_f \leq H_t$. The end result will eventually be a multimodel that will be coordinated by FSA-3.

The low-level continuous models $M_1, \ldots, M_5$ are defined as follows:[1]

1. (M_1) COLD: $T = \alpha$, $\dot{H_w} = 0$, $\dot{H_f} = 0$.
2. (M_2) HEATING: $\dot{T} = k_1(100 - T)$, $\dot{H_w} = 0$, $\dot{H_f} = 0$.
3. (M_3) COOLING: $\dot{T} = k_2(\alpha - T)$, $\dot{H_w} = 0$, $\dot{H_f} = -k_3$.
4. (M_4) BOILING: $T = 100$, $\dot{H_w} = -k_4$, $\dot{H_f} = k_5$.
5. (M_5) OVERFLOW: *same as BOILING* with constraint $H_f = H_t$.
6. (M_6) UNDERFLOW: $T = undefined$, $H_w = H_f = 0$.

The system phase is denoted by Φ and the state variables are:

- T: temperature of water.
- H_w: height of the water.

[1] Models M_2 and M_3 exhibit first order exponential behaviors and are, therefore, rough approximations of the actual boiling water system.

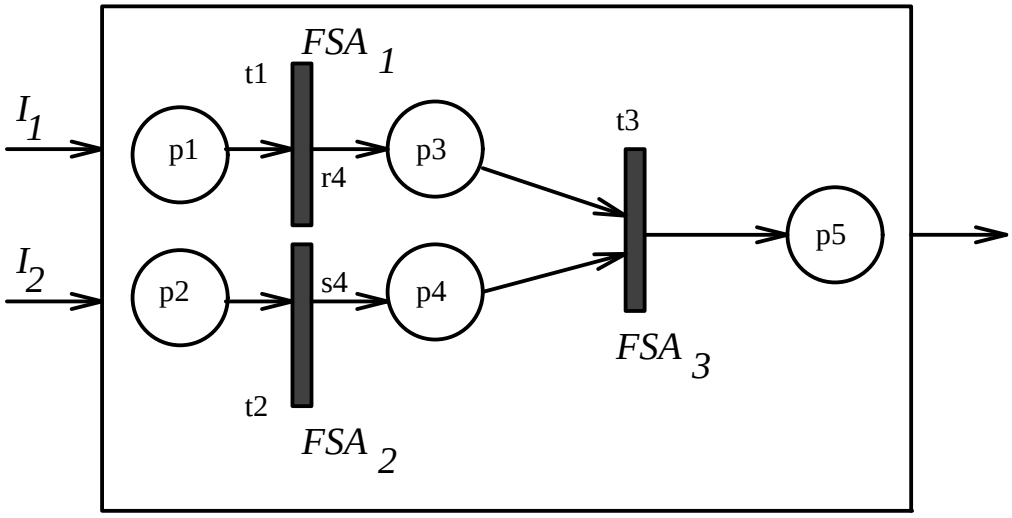


Figure 5: Petri net controller for three FSAs.

- H_f: height of the foam.

Note that the continuous models share a common set of state variables. However, in general, state variables may be different for each M_i model.

There are also some constants such as H_t for the height of top of pot, H_s for the starting height of water when poured into the pot; and k_i rate constants. The initial conditions are: $\Phi = cold$, $T(0) = \alpha$, $H_w(0) = H_f(0) = H_s$ and $knob = OFF$. By including the functional block knowledge, we create one large model called COMBINED that is defined as FSA-3 with each state containing a block model (as shown in fig. 4).

2.4 Petri Network Controller

Even though we have demonstrated an FSA controlled system with a heterogeneity afforded by blending both FSA models with block control models, we may also incorporate additional levels and model types. For instance, a new boiling water multimodel can be created by starting with FSA-3 as a primitive. Consider a scenario with two flasks of liquid; when both of the liquids are boiling, a human operator takes each flask and mixes the liquids into a separate container. For modeling the flasks, we can use two six state FSA controllers which "drive" models $M_1, \ldots, M_6$ as before. A five place Petri net serves to codify the constraint that both liquids must boil before the operator performs the separate function of mixing the liquids into another container. We use an extended Petri net in that 1) tokens can have attributes (i.e., have a *color*), 2) the net can accept a discrete input signal from outside the system (i.e., external events can occur), and 3) transitions can take an arbitrary amount of time (stochastic, timed network). Figure 5 shows a Petri net controller for this purpose. The Petri net, PN, is defined as follows [17]:

- $PNET =< P, T, I, O >$
- $P = \{p_1, \ldots, p_5\}$

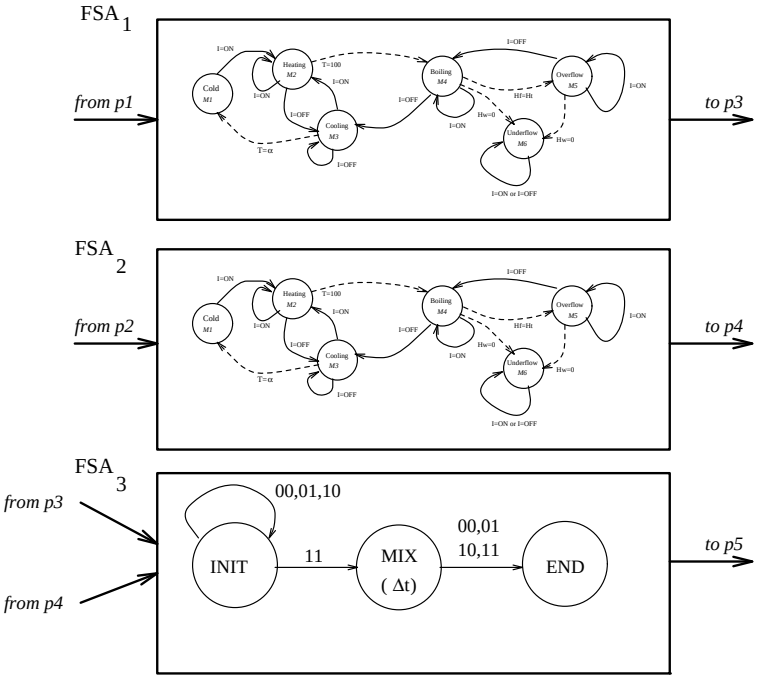


Figure 6: Three FSAs: two vessels and the operator.

- $T = \{t_1, t_2, t_3\}$
- $I : T \rightarrow P^{\infty}$
- $O : T \rightarrow P^{\infty}$
- $\mu : P \rightarrow Z_0^+$, for $i \in \{1, \ldots, 5\}, \mu(p_i) = 0$
- $I(t_1) = \{p_1\}, I(t_2) = \{p_2\}, I(t_3) = \{p_3, p_4\}$
- $O(t_1) = \{p_3\}, O(t_2) = \{p_4\}, O(t_3) = \{p_5\}$

Initially, the Petri net is in state $(0, 0, 0, 0, 0)$, meaning that there are no markers in any of the five places. There are two inputs to the Petri net: I_1 and I_2. These inputs represent discrete signals with values of either ON or OFF representing the state of the knob controlling the temperature. The two places, p_1 and p_2, accept a discretely sampled input signal from outside the system; tokens will have one of two possible attributes corresponding to ON and OFF. A heterogeneous refinement of the Petri net specifies that transitions t_1 and t_2 refine to FSA_1 and FSA_2, respectively. In this way, the input to the Petri net corresponds directly to the input signal for FSA_1 and FSA_2. Likewise, transition t_3 is refined into another FSA (FSA_3) which defines the states of the human operator: initial state ($INIT$), state of mixing the two liquids (MIX), and a final state (END). We specify an arbitrary amount of time for the second, mixing, state where the time lapse (Δt) is sampled from a normal probability distribution, for instance. Figure 6 displays FSA_1, FSA_2, and FSA_3. Let the state spaces for FSA_1 and FSA_2 be $\{r_1, \ldots, r_6\}$ and $\{s_1, \ldots, s_6\}$ respectively. When the state of FSA_1 is r_4 and the state of FSA_2 is s_4 then both liquids are in a boiling state.

The input to each FSA is a knob controlling the desired temperature, and the output from each FSA (designating the current FSA state at any given time) is determined as follows: *When an FSA is in the boiling state, output a token, otherwise do not output anything.* This policy is illustrated, on fig. 5 by attaching the FSA state names (r_4,s_4) next to the relevant transitions (t_1, t_2).

Now, we are in a position to describe the multimodel execution for the Petri net controller. The Petri net is controlled by the knob input, which produces colored tokens into places p_1 (for knob 1) and p_2 (for knob 2). There are two colors : ON and OFF. As this discrete signal is passed to a transition, the transition "fires." Transition firing, for a Petri net controller, means that the systems drops down an abstraction level to both FSA_1 and FSA_2 in parallel. The input signals continue to drive each FSA moving it from state to state. Each FSA state is further refined into a continuous block model which was illustrated in section 2.3. A combination of internal and external events causes a change in FSA state. Neither FSA produces an output until it enters the boiling state: r_4 for FSA_1 and s_4 for FSA_2. At this time, a token is produced and is put in places p_3 and p_4, depending on which flask is boiling. When both flasks are boiling, there will be tokens in p_3 and p_4, causing t_3 to fire. Since t_3 is further refined into FSA_3, we execute FSA_3 to simulate the action of the human operator — in this case, we model the operator by the amount of time spent mixing the liquids. The token produced by t_3 is realized by FSA_3 as an input value of 11 (representing tokens in both p_3 and p_4) which causes an internal state transition to the mixing state (MIX) of FSA_3. The mixing takes place for Δt and then the multimodel simulation ends.

3 DEVS Representation of FSA-Controlled Multimodels

To better understand the operation of the FSA-controller part of the multimodel simulation, we present a formalization using DEVS. An *FSA-controlled multimodel* is specified by a structure $< FSA, MODELS, map, TRANSITIONS >$, where

- FSA is a finite state automaton whose states are called *phases*. The input induced transitions, $(phase, input) \rightarrow phase$, form the external events of the multimodel.
- $MODELS$ is a set of models, M_i, each being specified as a system in the general formalism given earlier.
- map is a mapping from the states of FSA to $MODELS$; thus, each $phase$ is assigned a model $\in$ $MODELS$ which is intended to be the one and only active model when the multimodel is in that $phase$.
- $TRANSITIONS$ is a set of conditions, potentially one for each transition (pair of $phases$) in the FSA. These form the internal transitions of the multimodel. Formally, a transition condition associated with a $phase$ is a predicate on the state set of the model associated with that phase by map.

We can present the set of models for boiling water as an FSA-controlled multimodel. Our first step will be to transform the hierarchical system structure into a "flat" structure

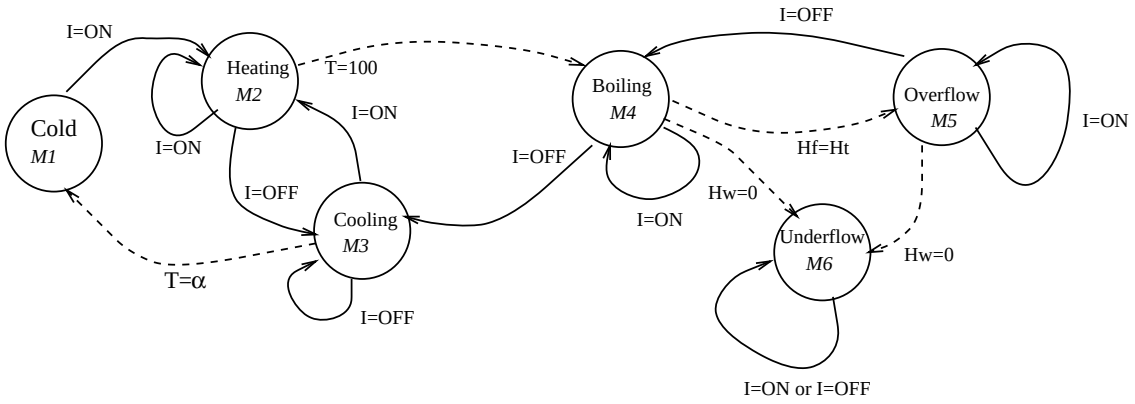


Figure 7: Six state automaton controller for boiling water multimodel (FSA-3).

by collapsing the hierarchy. Using the phase graph notation we create the graph in figure 7 by compressing the three levels in fig. 3 into a single level graph. This graph provides the FSA coordinator for the multimodel. The input induced (external) transitions are shown as solid arcs of the FSA. $MODELS$ is the set of continuous models $M_{1...5}$ given earlier; map is shown as the labeling of the phases with elements from $MODELS$; $TRANSITIONS$ is shown as the set of conditions attached to the dotted arcs in fig. 7.

Even though classical system theory [10, 15] provides strong definitions for individual systems and system networks there is little concentration on the concept of an "event." Events were not critical to the study of systems within the classical theory. Simulation researchers such as Zeigler [22] and Nance [13] extended the classical theory to formally specify discrete event models and the roles of events, state and time within simulation models. We now present a brief review of the resulting DEVS formalism as a prelude to mapping the FSA-controlled multimodel into a DEVS equivalent [22].

Time, in discrete event systems, is assumed to be real-valued ($T = \mathcal{R}_0^+$). The DEVS structure [24] $< U, Y, S, \delta, \lambda, ta >$ is as follows:

- U is the *input* event set.
- Y is the *output* set.
- S is the *sequential state* set.
- $\delta_{ext} : Q \times U \rightarrow S$ is the external transition function. $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the *total* state set of the model; (s, e) represents the state of having been in sequential state s for an elapsed time e.
- $\delta_{int} : S \rightarrow S$ is the internal transition function.
- $\lambda : S \rightarrow Y$ is the output function.
- $ta : S \rightarrow \mathcal{R}_{0,\infty}^+$ is the time advance function.

The DEVS formalism, as stated in its title (Discrete Event System Specification), is a shorthand means for specifying a general system as formalized earlier. We can think of it pictorially as a box containing some process. This box

accepts inputs and produces outputs. We map an FSA-controlled multimodel $< FSA, MODELS, map, TRANSITIONS >$ into a DEVS equivalent as follows. Let the DEVS be defined as $< U, Y, S, \delta, \lambda, ta >$, where:

- U is the input set of the FSA.
- Y is the output set of the FSA (not of interest here).
- $Q_M = T \times H_w \times H_f$. Q_M is the common state set of $MODELS$.
- $S = P \times Q_M$ where P is the set of $phases$ of the FSA. Note that a typical element of S is (p, q) where p is a phase and q is a state in Q_M. Also, a typical element of the total state set of Q is $((p, q), e)$.
- $\delta_{ext} : Q \times U \rightarrow S$ is defined by:
  $\delta_{ext}((p, q), e, u) = (fsa(p, u), \delta_{M(p)}(q, e))$

  where $fsa(p, u)$ is the the $phase$ into which the FSA enters when it receives an input u in $phase\ p$, and $\delta_{M(p)}$ is the transition function of $M(p)$, the model associated with $phase\ p$ by map. This formalizes our earlier interpretation: when the FSA receives an input it immediately switches phases and the state of the multimodel is updated to the state corresponding to an elapsed time of e using the model that was in control during that time.
- To define ta, we recall that a set of transition conditions is associated with a given phase p. Let $C_{p->p\prime}$ be the condition for an internal transition from p to $p\prime$. Let $T_{p->p\prime}$ be the time at which $C_{p->p\prime}$ first becomes true when $M(p)$ starts in state q. $T_{p->p\prime}$ is the minimum time t such that $C_{p->p\prime}(\delta_{M(p)}(q, t))$ is true.[2] Now, $ta(p, q)$ is the minimum of the $T_{p->p\prime}$, where $p\prime$ ranges over the transitions $C_{p,p\prime}$ in $TRANSITIONS$. Note that this minimum could be ∞ when none of the conditions is satisfied along the state trajectory starting from q in model $M(p)$.
- $\delta_{int} : S \rightarrow S$ is defined by:

  $\delta_{int}(p, q) = (p^*, \delta_{M(p)}(q, e))$

  where p^* is the phase dictated by the winning condition, i.e., the phase $p\prime$ such $T_{p->p\prime}$ is equal to $ta(p, q)$.[3]
- $\lambda : S \rightarrow Y$, the output function, can be defined in terms of the output of the currently active submodel, but is not of interest here.

In the preceding formulation of an FSA-controlled multimodel, the phases (i.e., states) of the FSA are mapped to $MODELS$. Often, however, it is possible to provide a more concrete representation of the phases by associating them with subsets of states in the

[2] We assume that such a minimum is well-defined, as it will be for continuous systems.

[3] Applying the tie-breaking rules if necessary.

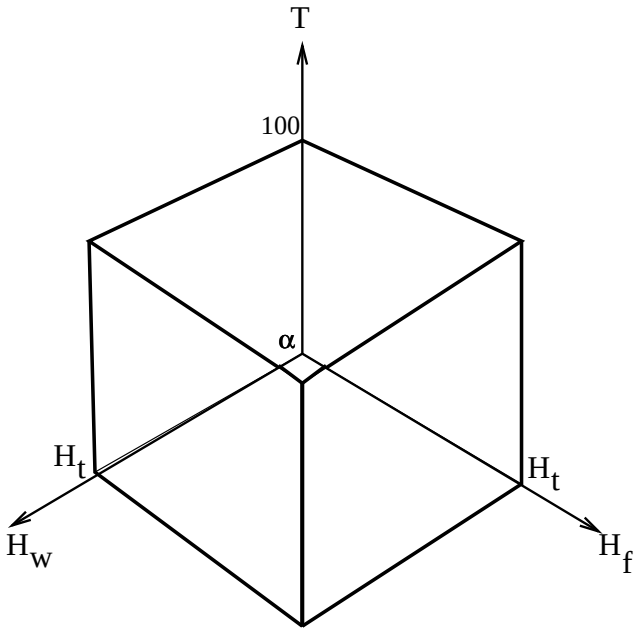


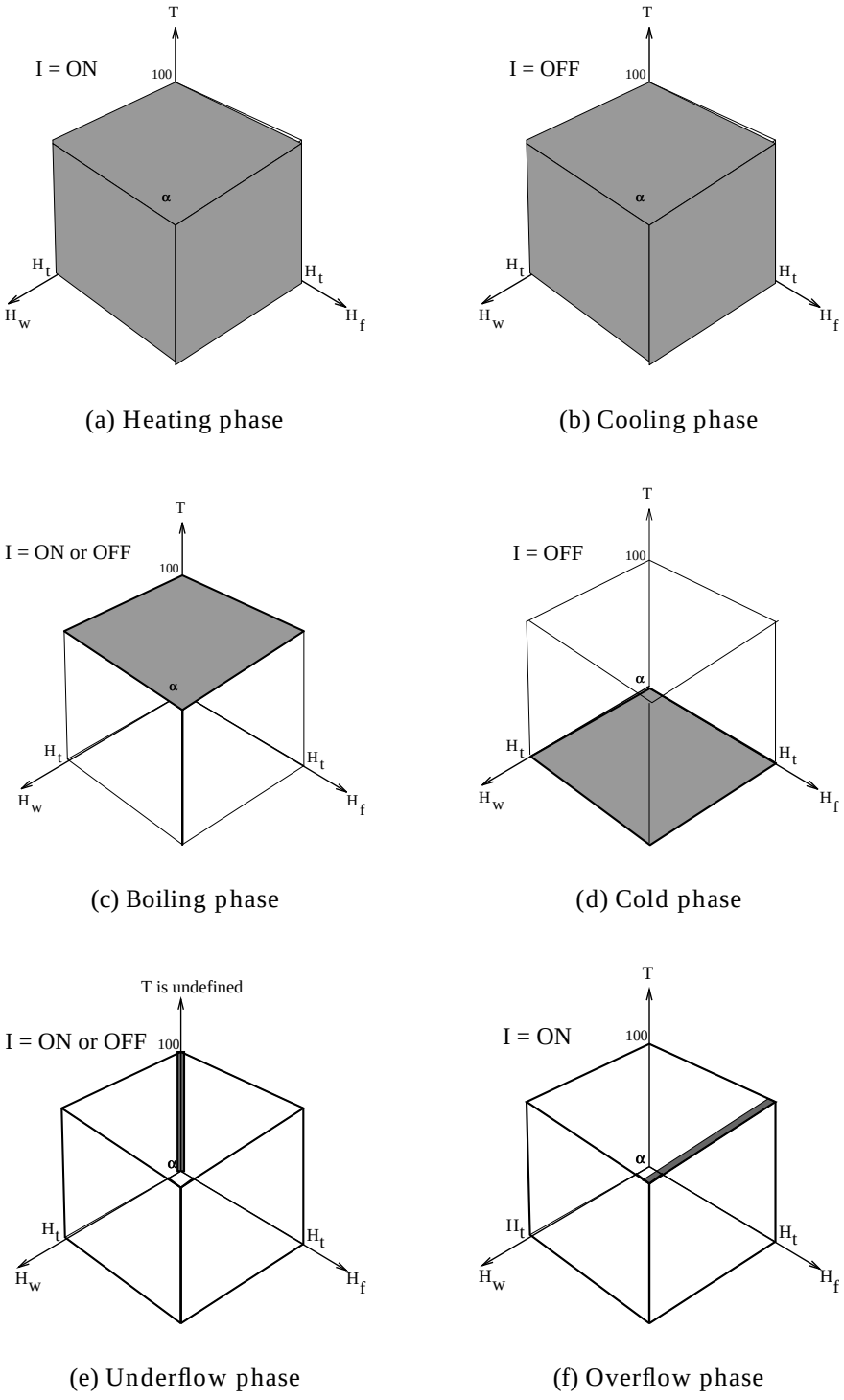
Figure 8: Continuous state space for boiling water system.

common state space, Q_M, of the $MODELS$. Since the input u participates in determining the next phase, both input and phase jointly determine partitions of Q_M in which the partition blocks are placed into correspondence with the phases.

Figure 8 displays the common three dimensional state space (T, H_w, H_f) for MODELS, and figure 9 illustrates each phase by a shaded region of state space. Table 1 provides the formal correspondence of phases and input values with partition blocks. Let $\pi(u, p)$ be the partition block corresponding to the pair (u, p) where u is an input value and p is a phase. The mapping π can be viewed as the labeling of partition blocks by input-phase pairs. In this case, the internal event transitions can be expressed in terms of boundary crossings of partition blocks. This means that a condition of the form $C_{p \to p'}$ can be written as the membership of a state in the partition block $\pi(u, p')$ where u is the input prevailing in phase p.[4] For example, in the boiling water example, the transition condition $C_{HEATING \to BOILING}$ under the input $I = ON$ is specified by the entry for $(BOILING, ON)$ in table 1. This transition corresponds to reaching the boundary given by plane $T = 100$ in figure 9(c).

Our discussion of phase space partitioning applies equally as well to the Petri net controller presented in section 2.4. Recall that each state of the FSA controller represents a partition of lower level state space. Now consider the Petri net which coordinates the two FSAs: each state in the Petri net represents a partition of the state space formed as a cross product of the state spaces of FSA_1 and FSA_2. PN state $(1, 1, 0)$ maps to a point, (r_4, s_4), in the lower level state space product. A PN state such as $(0, 1, 0)$ (meaning that FSA_2 is in the boiling state) maps to the region of lower level state space defined by the set $\{(r_1, s_4)$,(r_2, s_4),(r_3, s_4),(r_5, s_4), $(r_6, s_4)\}$.

[4] In our simplified formalization of section 3.5, the prevailing input u is assumed to be stored in the state, q. A more detailed formalization would make it an explicit component of q.



(a) Heating phase

(b) Cooling phase

(c) Boiling phase

(d) Cold phase

(e) Underflow phase

(f) Overflow phase

Figure 9: Phase partitions.

Table 1: Partitioning of boiling water state space.

$Phase$	I	T	H_w	H_f
COLD	ON	ϕ	ϕ	ϕ
COLD	OFF	$= \alpha$	$> 0 \wedge < H_t$	$> 0 \wedge < H_t$
HEATING	ON	$> \alpha \wedge < 100$	$> 0 \wedge < H_t$	$> 0 \wedge < H_t$
HEATING	OFF	ϕ	ϕ	ϕ
COOLING	ON	ϕ	ϕ	ϕ
COOLING	OFF	$> \alpha \wedge < 100$	$> 0 \wedge < H_t$	$> 0 \wedge < H_t$
BOILING	ON	$= 100$	$> 0 \wedge < H_t$	$> 0 \wedge < H_t$
BOILING	OFF	$= 100$	$> 0 \wedge < H_t$	$> 0 \wedge < H_t$
OVERFLOW	ON	$= 100$	$> 0 \wedge < H_t$	$= H_t$
OVERFLOW	OFF	ϕ	ϕ	ϕ
UNDERFLOW	ON	ϕ	$= 0$	$= 0$
UNDERFLOW	OFF	ϕ	$= 0$	$= 0$

4 Multimodel Simulation

We constructed a multi-level simulation in C that permits an execution of an FSA-controlled multimodel. For the boiling water example, there are four abstraction levels: three FSAs and one level containing six sets of equations (or block models). Our approach was to encode the model, as input to the simulator, as follows:

1. *Output Type* (1 digit). The type of output: 1) output the state variable time trajectory specified in *Output Value*, 2) output the state trajectory for the abstraction level specified in *Output Value*, 3) output the input state trajectory, and 4) output a phase trajectory.

2. *Output Value* (1 digit). Used in conjunction with the *Object Type*.

3. *Number of States*. The number of states in the lowest FSA level.

4. *External FSA Transition Table* (1 line per state). The lowest level FSA (level 3 for boiling water) transition function for external events. There are as many columns as there are possible input values.

5. *Internal FSA Transition Table* (1 line per state). The lowest level FSA transition function for internal events. Each internal event is associated with a condition and there is an integer value assigned to each condition. For instance, condition $T = 100$ is assigned the value 2.

6. *FSA Abstractions*. The number of FSA abstraction levels (3 for boiling water).

7. *Abstraction Mapping Table* (1 line per level). This contains the mapping from lower level to higher level FSA states so that state/phase output can be provided at any level during the simulation run.

Phases are labeled: 1: cold, 2: heating, 3: cooling, 4: boiling, 5: overflow, 6: underflow; and depending on the abstraction level, there are two other assignments: 2: not cold (for level 1) and 5: exception for level 2. Conditions are assigned: 1) $T = \alpha$, 2) $T = 100$, 3) $H_w = 0$, 4) $H_f = H_t$. Given these assignments, the multimodel simulator is executed with the data file (`bwater.in`) below within Unix as follows: `bwater < bwater.in > bwater.out`.

```
2 3
6
1 2
3 2
3 2
3 4
4 5
6 6
0 0 0 0
0 4 0 0
1 0 0 0
0 0 6 5
0 0 6 0
0 0 0 0
3
1 2 2 2 2 2
1 2 3 4 5 5
1 2 3 4 5 6
```

File `bwater.out` contains coordinate values for an individual run based on the input file. Figures 10 through 13 display the following:

- Figure 10: input trajectory; turning the knob on and off over some time period. This was chosen at random to display phase switching.
- Figure 11: temperature trajectory (T vs. time). Note that the temperature rises to $T = 100$ (in a first order lag) in response to the step input and then it falls before reaching the ambient temperature of $T = 20$. The temperature levels off at $T = 100$. The temperature is undefined when the phase changes to an overflow condition just prior to $t = 20$.
- Figure 12: water height trajectory (H_w vs. time). The height of the water starts at 5 and then moves down using a constant slope, with steps, until it reaches a zero level indicating an underflow condition.
- Figure 13: foam height trajectory (H_f vs. time).

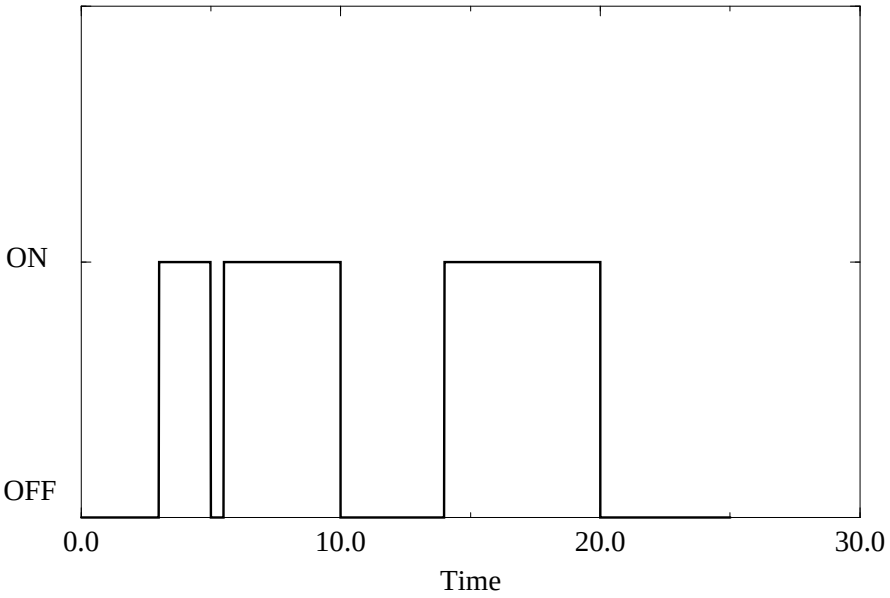


Figure 10: Knob input.

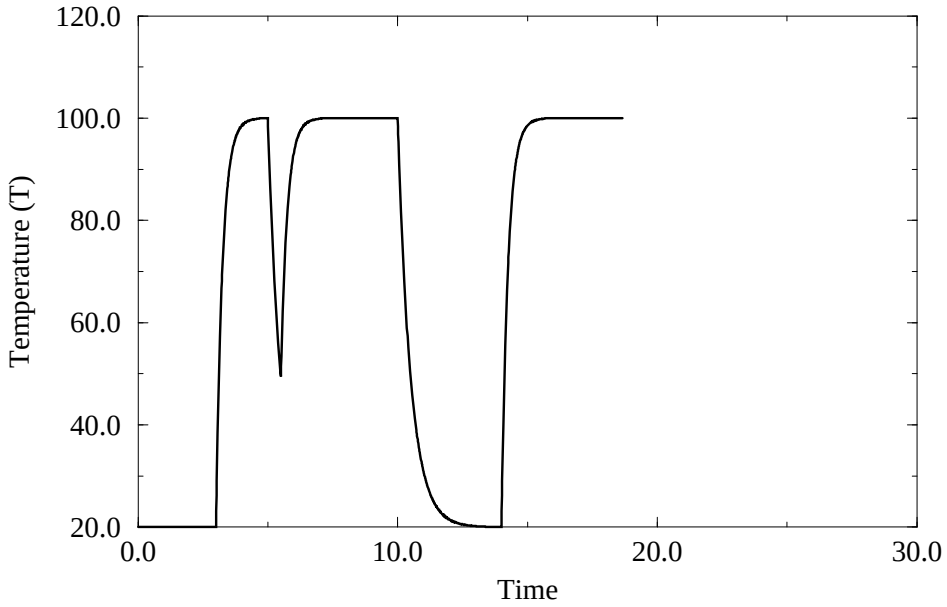


Figure 11: Water Temperature (T).

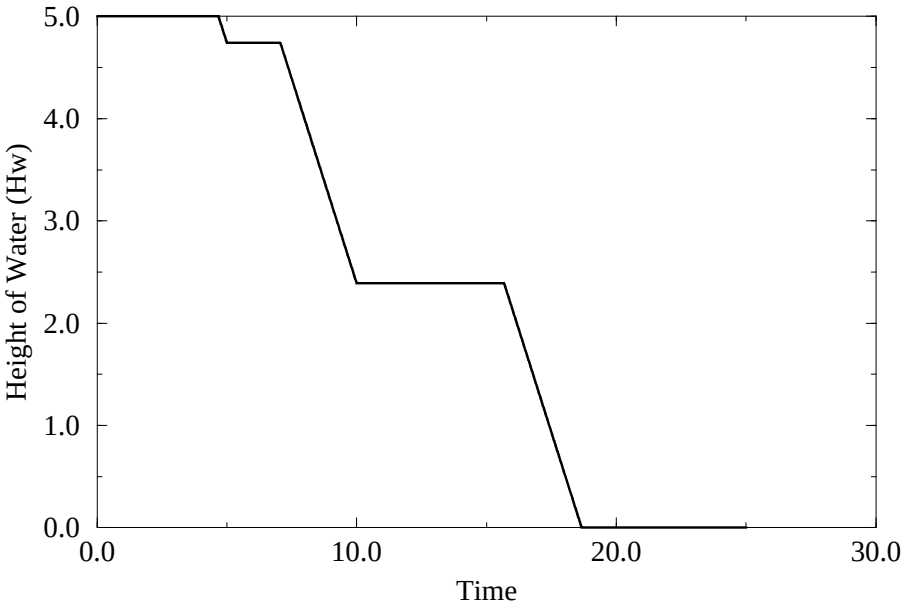


Figure 12: Water Height (Hw).

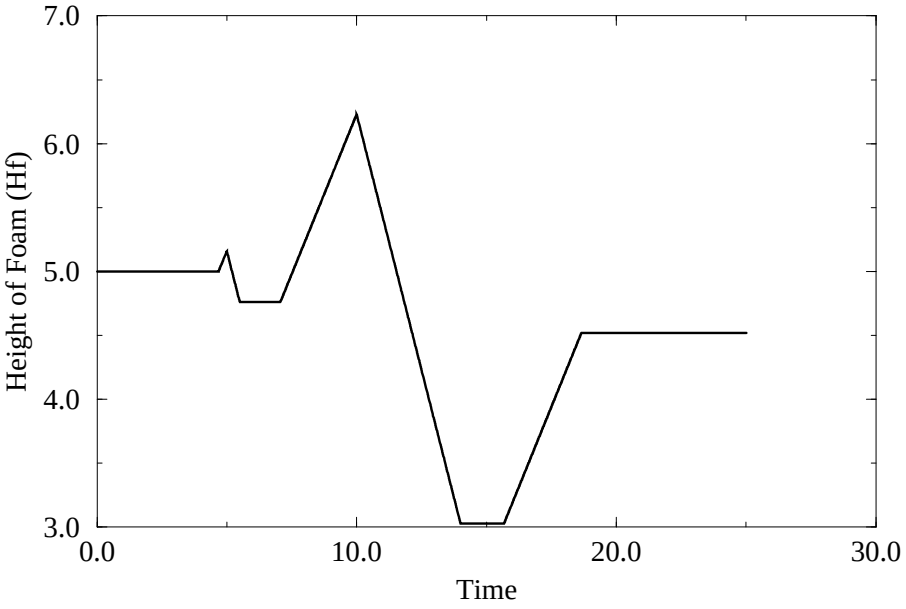


Figure 13: Foam Height (Hf).

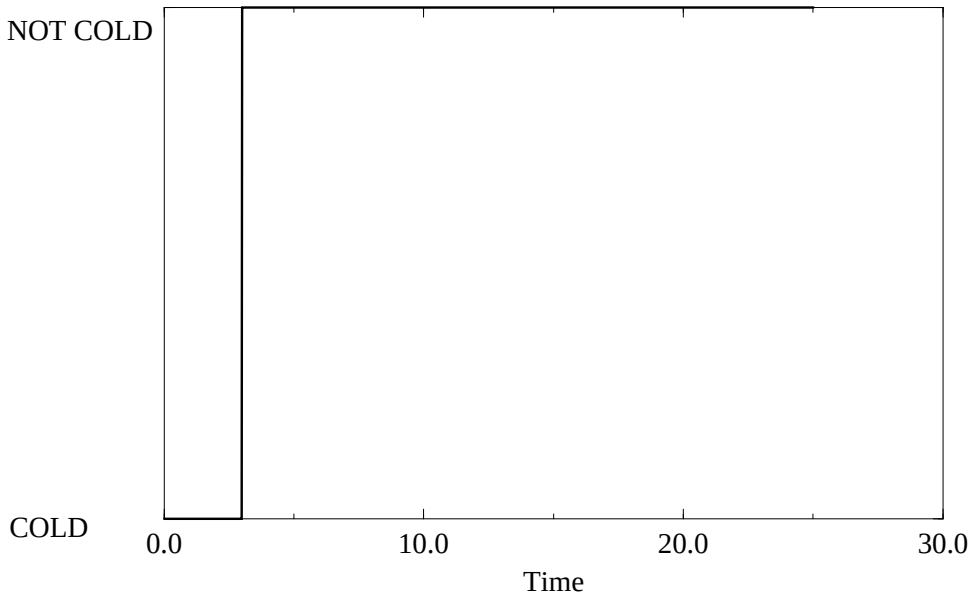


Figure 14: FSA-1 phase trajectory.

Figure 14 indicates a phase trajectory for abstraction level one (FSA-1). The behavior at this high level is simple: the system moves from state *cold* to *not cold* when the temperature increases as a result of the step input from the knob. There is no return to phase *cold* since the water evaporates completely before the temperature can be reduced. Figure 15 displays the phases present in level 3 (FSA-3); from this plot we obtain a qualitative explanation of how the system behaves.

5 Conclusions

We have used a system of boiling water to demonstrate how to create a multimodel containing three distinct types of models: 1) Petri net controller, 2) FSA controller, and 3) block. This demonstrates that models for complex systems can be constructed from networks of different model types. We found that by utilizing heterogeneous inter-level coupling, and homomorphic behavior preservation, we were able to create a solution to the problem of using only a single level model. With the multimodel, we can answer questions from “Why did the water start to boil?” to “How long will cooling take if the knob is turned off at time X?” In a bottom-up approach to multimodel design, we divided up the total boiling water process into distinct phases. Coordination of models for these phases was facilitated by the phase graph associated with the FSA at the final level of abstraction. Additionally, we extended the “single vessel system” to include two vessels containing liquid, where an operation was performed when the liquid in both vessels entered a boiling phase. In addition, a DEVS specification was provided to formalize the multimodel concept in a system theoretic manner. The SimPack software toolkit served as a good base for multimodeling since it contains the basic elements present in various well-known —and often used— models such as Petri nets, block models and automata. A C-based boiling water simulation was created using the SimPack tools and

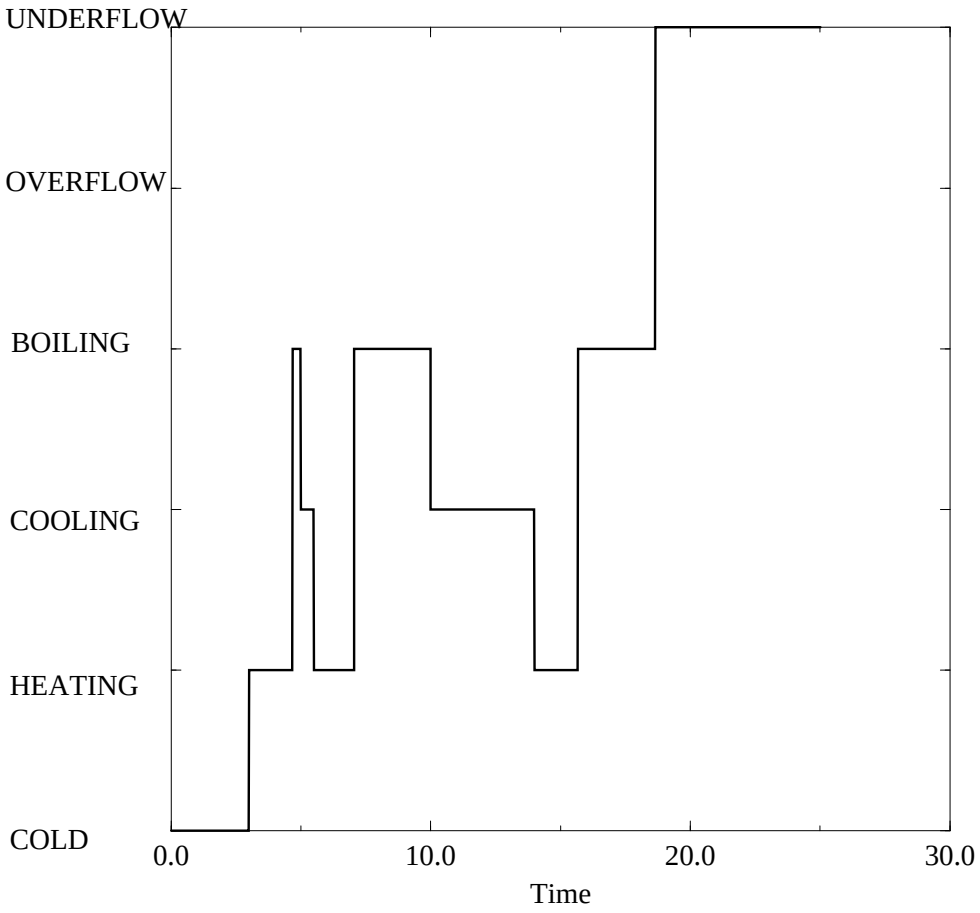


Figure 15: FSA-3 phase trajectory.

was demonstrated in section 4. SimPack is available and may be freely obtained by contacting the author. An accompanying text, including SimPack and multimodeling examples is nearing completion [8].

Although we have demonstrated our multimodel approach on the system of boiling water, we believe our method to be applicable to many more scenarios where different levels of abstraction are coded in the model form most appropriate for those levels. The problem of semi-automating the process of taking a conceptual, non-executable object based model of the system and converting it into an executable model remains a very hard problem, and while we have not completely solve the problem, we believe the multimodel approach to contain inherent solutions to a wider class of dynamic systems problems. For future work, we plan on making it easier for SimPack users to graphically construct multimodels using a window environment.

References

[1] Francois E. Cellier. *Combined Continuous System Simulation by Use of Digital Computers: Techniques and Tools*. PhD thesis, Swiss Federal Institute of Technology Zurich, 1979.

[2] Francois E. Cellier. *Continuous System Modeling*. Springer Verlag, 1991.

[3] Paul A. Fishwick. *Hierarchical Reasoning: Simulating Complex Processes over Multiple Levels of Abstraction*. PhD thesis, University of Pennsylvania, 1986.

[4] Paul A. Fishwick. The Role of Process Abstraction in Simulation. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):18 – 39, January / February 1988.

[5] Paul A. Fishwick. Abstraction Level Traversal in Hierarchical Modeling. In Bernard. P. Zeigler, Maurice Elzas, and Tuncer Oren, editors, *Modelling and Simulation Methodology: Knowledge Systems Paradigms*, pages 393 – 429. Elsevier North Holland, 1989.

[6] Paul A. Fishwick. SimPack: Getting Started with Simulation Programming in C and C++. In *1992 Winter Simulation Conference*, pages 154 – 162, Arlington, VA, December 1992.

[7] Paul A. Fishwick. An Integrated Approach to System Modelling using a Synthesis of Artificial Intelligence, Software Engineering and Simulation Methodologies. *ACM Transactions on Modeling and Computer Simulation*, 2(3), 1993.

[8] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall, 1993. (to be published as a textbook).

[9] Paul A. Fishwick and Bernard P. Zeigler. A Multimodel Methodology for Qualitative Model Engineering. *ACM Transactions on Modeling and Computer Simulation*, 2(1):52 – 81, 1992.

[10] R. E. Kalman, P. L. Falb, and M. A. Arbib. *Topics in Mathematical Systems Theory*. McGraw-Hill, New York, 1962.

[11] Granino A. Korn. *Interactive Dynamic System Simulation*. McGraw Hill, 1989.

[12] H. M. Markowitz, P. J. Kiviat, and R. Villaneuva. *Simscript II.5 Programming Language*. CACI, Inc., Los Angeles, CA, 1987.

[13] Richard E. Nance. The Time and State Relationships in Simulation Modeling. *Communications of the ACM*, 24(4):173 – 179, April 1981.

[14] Tuncer I. Oren. Dynamic Templates and Semantic Rules for Simulation Advisors and Certifiers. In Paul Fishwick and Richard Modjeski, editors, *Knowledge Based Simulation: Methodology and Application*, pages 53 – 76. Springer Verlag, 1991.

[15] Louis Padulo and Michael A. Arbib. *Systems Theory: A Unified State Space Approach to Continuous and Discrete Systems*. W. B. Saunders, Philadelphia, PA, 1974.

[16] C. D. Pegden, R. P. Sadowski, and R. E. Shannon. *Introduction to Simulation using SIMAN*. Systems Modeling Corporation, Sewickley, PA., 1990.

[17] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[18] Herbert Praehofer. Systems Theoretic Formalisms for Combined Discrete Continuous System Simulation. *International Journal of General Systems*, 19(3):219–240, 1991.

[19] A. A. B. Pritsker. *The GASP IV Simulation Language*. John Wiley and Sons, 1974.

[20] A. A. B. Pritsker. *Introduction to Simulation and SLAM II*. Halsted Press, 1986.

[21] Thomas J. Schriber. *An Introduction to Simulation using GPSS/H*. John Wiley, 1991.

[22] Bernard P. Zeigler. *Theory of Modelling and Simulation*. John Wiley and Sons, 1976.

[23] Bernard P. Zeigler. *Multi-Facetted Modelling and Discrete Event Simulation*. Academic Press, 1984.

[24] Bernard P. Zeigler. DEVS Representation of Dynamical Systems: Event-Based Intelligent Control. *Proceedings of the IEEE*, 77(1):72 – 80, January 1989.

[25] Bernard P. Zeigler. *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*. Academic Press, 1990.