Describing and Verifying Integrated Services of Home Network Systems

Pattara Leelaprute[1], Masahide Nakamura[2], Tatsuhiro Tsuchiya[1],
Ken-ichi Matsumoto[2], Tohru Kikuno[1]
[1]*Graduate School of Information Science and Technology, Osaka University, Japan*
{*pattara, t-tutiya, kikuno*}*@ist.osaka-u.ac.jp*
[2]*Graduate School of Information Science, Nara Institute of Science and Technology, Japan*
{*masa-n, matumoto*}*@is.naist.ac.jp*

Abstract

This paper presents a framework to specify and verify integrated services of a home network system (HNS). We first develop a modeling language to describe the HNS and the integrated services. Complementing our previous work, the language captures each appliance as an object consisting of properties and methods, encapsulating the underlying protocols and platforms. We then present a method that verifies the integrated services with symbolic model checking, by translating the proposed language into the SMV (Symbolic Model Verifier) language. Thus, it is possible to validate if the integrated service is specified as intended, automatically and exhaustively. Using the proposed framework, service developers can effectively detect design flaws in a single integrated service, as well as feature interactions among multiple services, in early stages of service development.

1. Introduction

With the emerging technologies in ubiquitous computing, general household appliances are being equipped with smart processors and network interfaces. Home appliances, such as TVs, DVDs, air-conditioners, refrigerators, ventilators, even lights and windows, are connected to LAN at home, comprising a *home network system* (HNS, for short).

A major challenge of the HNS lies in integrating features of *different* appliances via network [13]. The integration would implement various *value-added services* (called *HNS integrated services*), which provide a more comfortable and convenient living environment for home users. For instance, integrating an air-conditioner, a window (controller), a ventilator and thermometers would implement an HVAC (heating, ventilation and air-conditioning) service [4]. This integrated service achieves energy-saving air-conditioning by automatically controlling the ventilator and the window, according to the current temperature inside/outside the room.

Recently, several HNS protocols have been standardized (e.g., UPnP [15] for AV appliances, ECHONET [4] for white goods). However, these protocols only prescribe how each appliance communicates with others via network. How to integrate *features* of different appliances is beyond their scope. Thus, there are still many open issues in developing the HNS integrated services efficiently. Especially, methods in early stages of development, including *modeling*, *design* and *validation*, have not been widely studied. Lack of these methods may cause unreliable and low-quality service design, which forces service developers to directly implement services in an ad-hoc and time-consuming manner.

To cope with the problem, this paper presents a formal method for specifying and validating the HNS integrated services, which helps developers in early stages of service development. The proposed framework consists of two parts. The first part is a description method of the HNS and integrated services. Based on our previous work [10], we establish an object-oriented model of the HNS. In the model, each appliance (or environment) is represented by an object, consisting of properties and methods. For each appliance object, its properties characterize the current internal state of the object, while the methods represents features (i.e., APIs) that the object supports. Each method is modeled as a pair of pre and post conditions. HNS integrated services are constructed by combining the methods provided by different appliance objects. To specify integrated services, we propose a modeling language. This language allows sequential execution, branches and loop.

In the second part, we present a method that validates the integrated services with *symbolic model checking* [8]. Specifically, we develop a method for translating an integrated service specified in the proposed language into the well-known SMV (Symbolic Model Verifier) language. Once translated, the SMV tool automatically and exhaustively verifies the integrated service against any properties specified in CTL (Computational Tree Logic). Thus, we can effectively detect design flaws in integrated services.

Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)
0-7695-2465-6/05 $20.00 © 2005 IEEE

2. Home Network Systems

A *home network system* (*HNS*) consists of one or more *networked appliances* connected to a local area network at home. In general, each HNS appliance has a set of *device control interfaces* (i.e, *APIs*), by which the users or external software agents can control the appliance via the network. To handle API calls, an appliance is usually equipped with at least a processor, a storage and a network interface. The communication among the appliances is performed with an underlying *HNS protocol*. Several protocols are currently being standardized, such as UPnP [15], ECHONET [4], Jini [9] and HAVi [6]. The protocols prescribe a set of *network-level* agreements including; address setup, advertisement, message formats. In this paper, we discuss a generic framework independent of specific protocols. Hence, we assume that each appliance has a certain mechanism (e.g., middleware) to handle a protocol.

The main advantage of the HNS lies in the ability to flexibly integrate features of different appliances to provide value-added services. We refer to such services as *HNS integrated services*. For the sake of discussion, consider an example of an HNS which consists of the following networked appliances: an air-conditioner, a ventilator, a window, two thermometers and a smoke sensor. Also we assume that the air-conditioner operates either of the two modes: the cooling mode and the fan mode.

HVAC Service: HVAC achieves energy-saving air-conditioning of a room based on the set temperature. To simplify the discussion here, we focus on its cooling function. If the room is hotter than the set temperature, the HVAC service operates the air-conditioner in the cooling mode. To efficiently cool down the room, if the room temperature is hotter than the outside, the ventilator is also turned on to provide fresh outside air. In this case the ventilator will keep working until the room temperature reaches the outside temperature. If the room temperature is below the set temperature, on the other hand, HVAC has the air-conditioner operate in the fan mode.

Air-Cleaning Service: The Air-Cleaning Service, which involves the smoke sensor, is used to clean the air in the room. When the smoke sensor detects the smoke (caused by tobacco, cooking, or fire), the service automatically opens the window and turns on the ventilator. When the air is cleaned, the window and the ventilator are shut down.

The HNS integrated services are usually implemented as *software applications* that activate different HNS appliances in a pre-defined manner. These applications could be installed in a centralized *home server* [13], or could be distributed in appliances themselves [11]. Figure 1 shows an example HNS, where integrated services are deployed in the home server. Although several architectures for the appliance integration are currently being standardized (e.g., OSGi [13], DLNA [3] and DHF [7]), design and implementation of concrete integrated services (i.e., software applications) are completely up to individual service vendors.

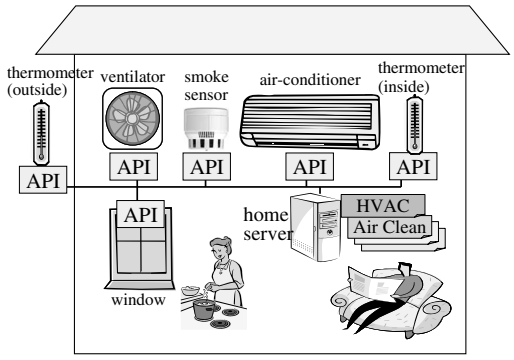

Figure 1. An Example Home Network System

3. The Object-Oriented Model for HNS

To assist a service developer efficiently in the early stages of development, it is essential to establish a *model* of HNS independent of the underlying protocols, platforms, or appliance implementations. In our model, both applications and the environment that the applications interact with are modeled as *objects*. The object of each appliance consists of *properties* and *methods*, while the object of the environment constitutes only of *properties*. The properties of an object are attributes that characterize the internal state of that object. Each property has a *type* determining allowable values of the property.

Definition 1 (Environment) An *environment* P_e is defined as a set of *environment properties*.

Definition 2 (Appliance) A *networked home appliance* d is defined as a 4-tuple $d = (P_d, M_d, Pre_d, Post_d)$, where

- $P_d = \{p_1, \cdots, p_n\}$ is a set of all *properties* of d.
- $M_d = \{m_1, \cdots, m_k\}$ is a set of all *methods* of d.
- Pre_d is a *pre-condition function* which maps each method $m_i \in M_d$ to a logic formula over P_d. m_i can be executed only when $Pre_d(m_i)$ is true.
- $Post_d$ is a *post-condition function* which maps each method $m_i \in M_d$ to a logic formula $[p_{i1} = a_{i1} \wedge p_{i2} = a_{i2} \wedge \cdots]$, where $p_{ij} \in P_d \cup P_e$, a_{ij} is either a constant value or a formula over of $P_d \cup P_e$. The type of a_{ij} must coincide with p_{ij}. If m_i is executed, the value of p_{ij} becomes equal to a_{ij} and the other properties that do not appear in the formula keep their values unchanged.

A property $p \in P_d$ (or a method $m \in M_d$) of an appliance d is denoted by $d.p$ (or $d.m$, respectively).

HNS = (P_e, D, W_e), where D = {AirConditioner, Thermometer_inside, Thermometer_outside, SmokeSensor, Window, Ventilation}

Appliance	Property Name	Property Type Name	Property Type	Initial Value
AirConditioner	Power	tPower	{ON,OFF}	OFF
	TempSetting	tAC_Temp	{18..28} (℃)	24
	ModeSetting	tAC_mode	{COOLING,FAN}	COOLING
Thermometer_inside	Power	tPower	{ON,OFF}	OFF
	CurrentTemp	tTemp	{15..40} (℃)	*
Thermometer_outside	Power	tPower	{ON,OFF}	OFF
	CurrentTemp	tTemp	{15..40} (℃)	*
SmokeSensor	Power	tPower	{ON,OFF}	OFF
	CurrentSmoke	tSmoke	{0,1}	0
Window	Power	tPower	{ON,OFF}	OFF
	WindowStatus	tWindowStatus	{OPEN,CLOSE}	CLOSE
Ventilator	Power	tPower	{ON,OFF}	OFF
Environment	Temp_in	tTemp	{15..40} (℃)	*
	Temp_out	tTemp	{15..40} (℃)	*
	Smoke	tSmoke	{0,1}	0

(a) Appliance/Environment Properties

Appliance	Method	Pre-Condition	Post-Condition	Env-Read	Env-Write
AirConditioner	ON()		Power='ON'		
	OFF()		Power='OFF'		
	setTemperature(tAC_Temp temp)	Power='ON'	TempSetting=temp		Temp_in
	setMode(tAC_Mode mode)	Power='ON'	ModeSetting=mode		Temp_in
Thermometer_inside	ON()		Power='ON'		
	OFF()		Power='OFF'		
	measureTemp()	Power='ON'	CurrentTemp=Temp_in	Temp_in	
Thermometer_outside	ON()		Power='ON'		
	OFF()		Power='OFF'		
	measureTemp()	Power='ON'	CurrentTemp=Temp_out	Temp_out	
SmokeSensor	ON()		Power='ON'		
	OFF()		Power='OFF'		
	detectSmoke()	Power='ON'	CurrentSmoke=Smoke	Smoke	
Window	ON()		Power='ON'		
	OFF()		Power='OFF'		
	OPEN()	Power='ON'	WindowStatus='OPEN'		Temp_in
	CLOSE()	Power='ON'	WindowStatus='CLOSE'		Temp_in
Ventilator	ON()		Power='ON'		Temp_in
	OFF()		Power='OFF'		Temp_in

(b) Appliance Methods

Figure 2. The Model of the Example HNS

The methods of an appliance represent the APIs (i.e., device control interfaces) that refer or update the properties of the appliance. Since the internal implementation of the APIs is usually encapsulated in the appliance, we characterize each method simply by a pair of a *pre-condition* and a *post-condition.*

For each appliance object, the current values of the properties represent the *state* of the object. A method execution causes a *state transition* of the object, which changes the current state to the *next* state according to the pre/post conditions.

Definition 3 (Appliance Semantics) A *state* s of an appliance $d = (P_d, M_d, Pre_d, Post_d)$ is defined as $s = [c_1, c_2, \cdots, c_n]$, where c_i is the *current value* of the property $p_i (\in P_d)$. For a method $m \in M_d$, we say that m is *enabled* under s iff $Pre_d(m)$ is true for the current values represented in s. When m is enabled under s, m can be executed. If m is executed, the state s is changed to the *next state* $s' = [a_1, a_2, \cdots, a_n]$, as specified in $Post_d(m) = [p_{i1} = a_{i1} \wedge p_{i2} = a_{i2} \wedge \cdots]$. We assume that during the execution of m, no other methods of d can be executed.

Intuitively, a pre-condition $Pre_d(m)$ models a *guard* of the method m. A post-condition can be regarded as an *assertion* that must be satisfied after the method is executed.

Appliances can interact with the environment through the execution of their methods. In our model, sensing the environment is represented by the post-condition of a method. That is, the environment properties can be read in the post-condition.

Executing appliance methods may also have some effects on the environment. To specify which methods can change which environment properties, we introduce the following definition.

Definition 4 (Environment Write Function) Let $D = \{d_1, d_2, \cdots, d_r\}$ be a set of all appliances. Also, let $M = \cup_{d_i \in D} M_{d_i}$ be a set of methods of all appliances. The effects on the environments caused by method executions are specified by the *environment write function*. The *environment write function* $W_e : M \rightarrow 2^{P_e}$ maps each method $m \in M$ to a set of environment properties that are written by m.

Consequently, an HNS is defined by a set of appliance objects and an environment object.

Definition 5 (Home Network System) A home network system is defined as $HNS = (P_e, D, W_e)$, where

- P_e is the environment.
- $D = \{d_1, d_2, ..., d_r\}$ is a set of appliances.
- W_e is the environment write function.

Figure 2 shows a model of the example HNS presented in Section 2. Figure 2(a) lists properties and their types for each appliance or environment object. For instance, `AirConditioner` has three properties: `Power`, `TempSetting` and `ModeSetting`. The type of `Power` is {`ON`,`OFF`}, saying that the value of `Power` is either `'ON'` or `'OFF'`. For convenience, each type has a name prefixed by `t`. Figure 2(b) lists appliance methods. For each method, its pre/post conditions and the environment properties to read/write are shown. For example, take the method `AirConditioner.setMode(tAC_Mode mode)`. Its pre-condition states that the method can be executed only when `Power='ON'` holds for the air-conditioner. The method has a formal parameter `mode`, to which some value of type `tAC_Mode` is assigned when this method is invoked (e.g., `AirConditioner.setMode('COOLING');`). When the method executed, property `ModeSetting` is updated to the value of `mode`, according to its post-condition (`ModeSetting=mode`). Invocation of the method writes an environment property `Temp_in`, meaning that setting the air-con mode can change the temperature inside the room. In Figure 2, `*` denotes a "*don't care*". Although formal parameters are not defined in our model, they are used simply for notational convenience; a method with formal parameters represents a family of methods.

4. The Proposed HNS Description Language

In this section, we propose a language for representing integrated services based on the model described in the previous section. Our language consists of two parts: (a) system description for the HNS and (b) service description for the integrated services.

This description language is designed with the assumption that the platform that executes the integrated service can read the values of the properties of any appliances.

4.1. System Description

The system description part is used to describe the HNS model defined in Section 3. An HNS is described in the following format. A sentence begins with # is a comment, and capital words (e.g., SYSTEM, TYPEDEF) denote keywords.

```
SYSTEM HNS_name {
  TYPEDEF      # Type definition
    type_name1  type1;
    type_name2  type2;
    :
  # Environment definition
  ENVIRONMENT env_name {
!!  PROPERTY   # Environment properties declaration
     type_name1  env_property1 [:= init_val];
     type_name2  env_property2 [:= init_val];
     :
  }
  # Appliance definition
  APPLIANCE appliance_name1 {
!!  PROPERTY   # Appliance properties declaration
     type_name1  app_property1 [:= init_val];
     type_name2  app_property2 [:= init_val];
     :
    METHOD     # Method definition
     :
  }
  APPLIANCE appliance_name2 {
     :
  }
  :
}
```

The system is named at the top nesting level following SYSTEM keyword. The body of the system consists of three sections: TYPEDEF section, ENVIRONMENT section, and APPLIANCE section.

Type Definition The TYPEDEF section declares types commonly used in the system. The proposed language supports three types: *Boolean* (i.e., {true,false}), *integer*, or *enumeration*. An integer type is specified by upper and lower bounds, e.g., {1..5}. An enumeration type is defined by enumerating concrete elements, e.g., {ON,OFF}. Every type can be named by an identifier, e.g., tPower {ON,OFF}. The scope of the type name covers the entire system description.

Environment Definition The ENVIRONMENT section define an environment object. In our HNS model, the environment consists of only environment properties. For each property, an identifier and its type declared in the TYPEDEF section are specified. Optionally, one can specify the initial value of the property with the assignment operator (:=).

Appliance Definition All appliances deployed in the HNS are declared in multiple APPLIANCE sections (blocks), each of which defines an appliance object. An APPLIANCE block comprises of definitions of properties and methods of the appliance. The appliance properties are specified in the same way as in the ENVIRONMENT section. Each method is described in a METHOD subsection in the following format:

```
METHOD     # method definition
 return_type1 method_name1([type_name formal_param]*){
   PRE         pre_condition;
   POST        post_condition;
   ENV_R       env_name.property_name;
   ENV_W       env_name.property_name;
   RETURN      return_value;
 }
 return_type2 method_name2(...) {
   :
 }
 :
```

A method is associated with a name and its return type; for notational convenience, we allow methods to return a value. A read of the return value of a method represents a subsequent execution of that method and a read of an appliance property of interest. The return type must be a type that is declared in the TYPEDEF or void. A method can have one or more formal parameters (separated by ','), each of which is declared by type and name. The body of the method contains pre/post conditions, environment read/write, and the return value. To specify a logical formula for the pre(or post) condition, the properties, the formal parameters, and ordinary unary and binary operators (+, -, =, <, >, <=, >=, !=, !, &, |) can be used. ENV_R (or ENV_W) enumerates environment properties read (or written, respectively) by the method. RETURN specifies the value to be returned, which can be specified by a property or an expression (e.g., AC_Temp + 1).

Appendix A shows the system description for the example HNS presented in Figure 2.

4.2. Service Description

To provide sufficient expressive power required to describe complex integrated services, the service description part supports: (a) WHILE and IF statements, (b) local variables for storing temporal data, and (c) pseudo functions to abstract events. The service description adopts the following format.

```
DEPLOYED_SYSTEM HNS_name;

SERVICE service_name1([type_name formal_param]*){
   # Local variable declaration
   VAR  type_name local_var1 [:=initial_value];
        type_name local_var2 [:=initial_value];
   :
   # Appliance declaration
   APPLIANCE appliance1, appliance2,..;

   CONTENT             # service content
    statement1;
    statement2;
    :
```

```
}
SERVICE service_name2(type_name formal_para_name){
       :
}
```

The keyword DEPLOYED_SYSTEM is used to specify the HNS where the services are deployed. It *imports* a system description designated by HNS_name. The integrated services are defined by one or more SERVICE blocks. Similarly to the appliance method, a service can have formal parameters. An actual value is passed to each parameter when a user agent invokes the service.

Variable and Appliance Declarations A service can use local variables declared in the VAR subsection. The declaration is done in the same way as properties in the system description. Allowable types are either Boolean, integer, or enumeration. The scope of the variables is only within the service. The APPLIANCE subsection designates appliance objects used in the service. These appliances must be the ones defined in the imported system description.

Service Content The CONTENT subsection describes the body of the service, which consists of one or more *statements*. Basically, the statements are sequentially executed one-by-one from top to bottom, as in the ordinary procedural programming language. The syntax of the statements is given in BNF as follows.

```
statement::
        invocation_statement
      | assignment_statement
      | if_statement
      | while_statement
      | { compound_statement }
invocation_statement:: invocation ;
invocation::
        appliance_name.method_name(arg_list)
      | pseudo_function
arg_list:: expression | expression, arg_list
pseudo_function:: end() | exit()
assignment_statement:: local_var := expression
if_statement::
        if (Boolean_expression) statement
      | if (Boolean_expression) statement else statement
while_statement::
        while (Boolean_expression) statement
        compound_statement:: statement
      | compound_statement statement
```

• An invocation_statement refers to a statement that invokes an appliance method (or a pseudo function, discussed later). The method invocation is executable when its pre-condition is satisfied. Invocation of an appliance method is a primitive construct of an integrated service. Invocation of an appliance method can take *actual parameters* as a list of *expressions*. Due to limited space, we omit the syntax of expression. In the proposed language, an expression is constructed by operators given in Section 4.1. As for *operands*, local variables, appliance properties, constants, and invocation can be used.

• A pseudo_function is a meta function to model an abstract *event* in the service. Two types of functions are supported: end() and exit(). end() returns true (1) when *the user terminates the service* (e.g., with a termination signal from the user agent). exit() models a system call by which *the system terminates the service*. These allow the developer to specify explicitly when or by whom the service is terminated. The pseudo functions are always executable.

• An assignment_statement sets a local variable to the evaluated value of an expression of the same type. Note that no appliance property can come at the left-hand side, since the properties cannot be updated directly, and should be written via appliance methods.

• An if_statement specifies a conditional branch. If Boolean_expression is evaluated to be true, then the first statement is executed next. If not, the second statement (in ELSE-clause) is chosen to execute. The Boolean_expression is an expression whose result takes true (1) or false (0).

• A while_statement specifies a loop. The statement is repeatedly executed while the associated Boolean_expression is true.

• A compound_statement represents a *block* of multiple statements.

Appendix B represents an example design of the HVAC and the Air-Cleaning services introduced in Section 2. The service description assumes that the services are deployed in the HNS described in Appendix A.

5. Translating Service Description for Symbolic Model Checking

The service description written in the proposed language is rigorous enough to be amenable to formal verification. In this section we show the method for model checking the integrated services with the SMV model checker .

5.1. Symbolic Model Checking and SMV

Model checking is the process of exploring a finite state space to determine whether or not a given property holds [2]. The major problem of model checking is that the state spaces arising from practical problems are often extremely large. A promising approach to this problem is the use of *symbolic* representation of the state space; that is, Boolean functions are used to represent the state space, instead of explicit adjacency-lists. *SMV* is a software tool that implements symbolic model checking [8]. To use SMV to verify a system, the system needs to be described as an SMV program in the SMV language. An SMV program is divided into one or more modules, each of which specifies a finite state machine. The interaction between concurrently executing modules is specified in the main module. In particular, if concurrently executing modules are defined using the process keyword, these modules use an interleaving semantics, in which only one process transitions at each step.

Each module contains variable declarations to determine its state space and descriptions of the initial state and transition relation of the machine. Variable declarations are preceded by the keyword VAR. The type associated with a variable can be Boolean or an enumerated type. The transition relation is described by a collection of parallel assignments to the next version of the variables. Assignments of initial values and next values to the variables are preceded by the keyword ASSIGN. Initial states are assigned by specifying the initial values of the variables using the expression `init(x)`, where `x` is a variable. The expression `next(x)` is used to refer to the variable `x` in the next state

The correctness property to be verified is specified in an SMV program as a formula in CTL (Computational Tree Logic), under the keyword SPEC. CTL is a well-known temporal logic. SMV verifies whether all possible initial states satisfy the property or not.

5.2. Compiling Service Description

This subsection presents a method for *compiling* an integrated service described in the proposed language into an SMV program. Our language basically describes each service as a sequential program, where statements are executed sequentially one-by-one according to given control flows. On the other hand, the SMV describes the system based on the transition relation only. Hence, the key is how to represent the original control flow in the SMV program. To achieve the compilation, we take a similar idea discussed in Chapter 2 of [2].

To represent the behavior of integrated services by an SMV program, it is necessary to determine the semantics of concurrent execution of multiple services. We make the following assumptions.

- Multiple services can be executed concurrently and in an interleaving manner.
- A statement with one or no method invocation is executed in an atomic step.
- A statement involving n method invocations is executed as consecutive n atomic actions, each of which corresponds to the execution of one of the n methods. The order of method invocation is from left to right.

Also we assume that a service description Ser and the corresponding system description Sys are given as inputs. The output is an SMV program smv that simulates Ser.

Step 0: Preliminary We define a main module in smv. For every appliance (or environment) property $A.p$ in Sys, we define $A.p$ as a variable `A_p` in VAR section of smv, with the appropriate type. Also, we assign the initial value to `A_p` in ASSIGN section of smv. For every environment property ep, we define its next value as an arbitrary value of its type, since the environment changes arbitrarily so that the system cannot predict it. (This solution is also adopted in [1].) For instance, for the system given in Figure 2, the main module would be:

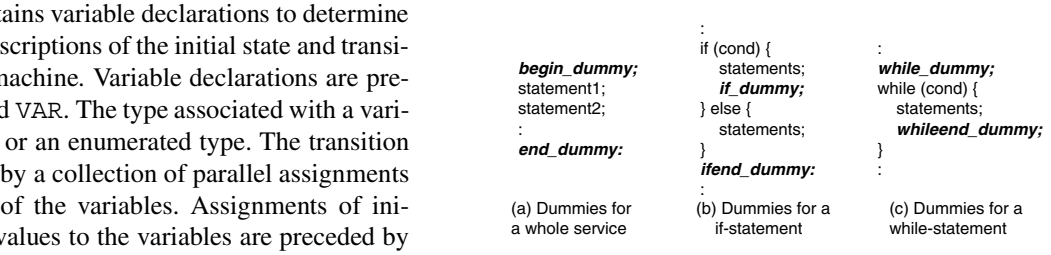

Figure 3. Inserting Dummy Statements

```
MODULE main
VAR
   AC_pwr:{ON,OFF};      --AC. Power (Appliance property)
   :
   ENV_T_out:{15,..,40}; --Temp outside (Env. property)

ASSIGN
   init(AC_pwr):=OFF;
   :
   next(ENV_T_out):={15,..,40};  --Changes arbitrarily
   :
```

For each SERVICE svc declared in Ser, we perform from Step 1 to Step 5.

Step 1: Making a Module for the Service We define svc as a module `svc` in smv. The formal parameters of svc are defined as variable in VAR section with appropriate type. Also, we assign the initial value by using an arbitrary value of their type, since we cannot predict which value the user will choose when executing the service. Then, the formal parameters of svc and all properties referred in svc are specified as the formal parameters of `svc`. After that, `svc` is included as `process` in the main module. Also, we declare local variables of svc in VAR section as they are. For instance, HVAC service could be:

```
MODULE main
VAR
   :
   user_temp:{18,..,28};     --Formal parameter of HVAC
   HVAC1: process HVAC(user_temp,AC_pwr,...,ENV_T_out);
ASSIGN
   :
  init(user_temp) := {18,..,28}; --Selected arbitrarily

MODULE HVAC(usr_temp,AC_pwr,...,ENV_T_out)
VAR
   Ti_temp: {15,..,40};          --Local variable
  :
ASSIGN
  init(Ti_temp) := 25;
  :
```

Step 2: Labeling Statements For every statement in CONTENT of svc, we attach a *unique label* to achieve a consistent control flow in smv. First, we insert *dummy statements* in svc, as shown in Figure 3. These dummies have no effect, but for specifying special points of control. Next, we *extract* each method invocation from ex-

Labeled Statement	DEFINE
L: $m_i()$; # method *where* PRE: *formula*; POST: $p_{i1} = a_{i1}$ & $p_{i2} = a_{i2}$ & …;	pre*L* := *formula*; post*L* := ($p_{i1} = a_{i1}$ & $p_{i2} = a_{i2}$ & …); next*L*_p_{i1} := a_{i1}; next*L*_p_{i2} := a_{i2}; :
L: var:=*exp*; # assign	next*L*_var:=*exp*;
L: if (*bool_exp*) { # if	if*L*_true := *bool_exp*; if*L*_false:= !(*bool_exp*);
L: while (*bool_exp*) { # while	while*L*_true := *bool_exp*; while*L*_false:= !(*bool_exp*);

Table 1. Condition Conversion Rules

Labeled Statement		ASSIGN next(pc) : =	
L:	m();	(pc = *L*) & pre*L*	: *L*+1 ;
L:	if (*cond*) {	(pc = *L*) & if*L*_true	: *L*+1;
		(pc = *L*) & if*L*_false	: *label*(***if_dummy***)+1;
L:	***if_dummy***;	(pc = *L*)	: *label*(***ifend_dummy***);
L:	while (*cond*) {	(pc = *L*) & while*L*_true	: *L*+1;
		(pc = *L*) & while*L*_false:	*label*(***whileend_dummy***)+1;
L:	***whileend_dummy*** ;	(pc = *L*)	: *label*(***while_dummy***);
L:	exit();	(pc = *L*)	: *label*(***end_dummy***);
L:	***end_dummy*** ;	(pc = *L*)	: *label*(***begin_dummy***);
L:	*any_other_statement* ;	(pc = *L*)	: *L*+1;

Table 2. Program Counter Update Rules

pressions. For each method invocation $m()$ in an expression e, we replace $m()$ by the return value of $m()$. Then, we add $m()$; immediately before e. For instance, `var:=A.m1()+B.m2()+c;` is translated into three statements below:

```
A.m1();
B.m2();
var: = RET(A.m1()) + RET(B.m2()) + c;
```

For the resultant statements, we put sequence numbers from `begin_dummy;` to `end_dummy;`, as the unique labels. By convention, we assume that `begin_dummy;` and `end_dummy` are labeled by `0:` and `max:`, respectively. In the following, let $label(stm)$ denote the label (number) of a statement stm.

Step 3: Extracting Conditions as Macros For each stm of the labeled statements, we extract conditions from stm, and convert them into `DEFINE` macros of smv. The rules for the conversion are shown in Table 1, supposing that $L = label(stm)$. The first rule applies when stm is a method invocation $m_i()$. In `DEFINE`, pre/post conditions are described as they are. Moreover, a value of each property p is defined as `nextL_p`, which will be used to specify the next value of p in Step 5. The second rule applies to assignment statements, which specifies the next value of local variable var. The last two rules apply to `if` and `while`. The rules generate two macros that represent two cases where the condition is true and false.

Step 4: Deploying a Program Counter To simulate the control flow of svc, we introduce a variable `pc` in smv as a *program counter*. First, we define a local variable `pc` in the `svc` module of smv, as `VAR pc:{0..max};`, and initialize as `ASSIGN init(pc):=0;`. Suppose that `pc = L` when smv is currently simulating the L-th statement stm (i.e., $label(stm) = L$). Then, we want to update `pc` to an appropriate *next* value after stm is executed. For this, we present rules to compute the next value of `pc` in Table 2.

The first rule applies when stm is a method invocation $m()$. In order for $m()$ to be executed, the pre-condition of $m()$ `pre`L (see Table 1) must hold. In this case, the next value of `pc` will be L+1. When stm is an if-statement (or while-statement), the next value depends on the condition as shown in the second (or fourth) rule. If the condition is false, then stm is set to the label of a dummy statement, so that the execution *jumps* to an appropriate control point (see Figure 3). Similarly, a jump also occurs when stm is a certain dummy statement. When stm is `exit()`, `pc` is set to the last statement, which simulates the end of service. We apply the rules to all the statements, and resultant SMV code would be: `ASSIGN next(pc):= case .... esac;`.

Step 5: Specifying the Transition Relation When a method invocation $m()$ occurs, properties specified in the post condition of $m()$ are updated to the next values, which defines a *transition relation* of the system. Hence, for each property p, we need to identify *when* p is updated by *which statement*. Fortunately, we can easily specify the transition relation by consulting `DEFINE` section generated in Step 3. For a property p, if `DEFINE` contains `next`L`_p:=a;`, it means that p is updated by the L-th statement, which is a method invocation $m()$. Thus, p can be updated when `pc=L` and the pre-condition of $m()$ becomes true. Note that p could be updated in other statements as well. As a result, the next value of p can be defined as:

```
ASSIGN next(p):= case
  (pc=L) & preL :nextL_p; --updated by L-th statement
    :                     --updated by other statements
   1: p;                  --stay unchanged.
  esac;
```

Similarly, we can define the next value of a local variable var, which could be updated by an assignment statement.

```
ASSIGN next(var):= case
  (pc=L)    :nextL_var; --updated by L-th statement
    :                   --updated by other statements
   1: var;              --stay unchanged.
  esac;
```

We apply the same procedures to all the properties and local variables defined in svc. Finally, we model the behavior of `end()` pseudo function. By definition, `end()` returns true when the user wants to terminate the service. However, the system cannot predict when the termination occurs. So, we abstract the behavior by using non-determinism.

```
VAR end:{0,1};      --Boolean for end() function
ASSIGN next(end):= case
  (pc=L)    :{0,1}; --non-determinism in L-th statement
```

```
 1: end;          --stay unchanged.
esac;
```

6. Verifying Integrated Services

Using the proposed language, the HVAC and Air-Cleaning services in Section 2 can be described (see Appendix A and B for the system and service descriptions). The description is compiled into an SMV program, comprising about 580 lines of code. Due to the page limitation, we show the compiled HVAC service only in Appendix C.

Then, based on the informal descriptions (high-level requirements) presented in Section 2, we derived the following CTL formulas as desirable properties for the services.

Properties for HVAC Service:

• If the service is started, then the air-conditioner will be eventually turned on.

```
P1: SPEC AG(pc=1 -> AF(AC_pwr=on ))
```

• If the room temperature exceeds the set temperature, then the mode of the air-conditioner will be set to the cooling mode. On the other hand, if the temperature is below the set temperature, then the mode of the air-conditioner will be set to the fan mode. (`Ti_temp` represents the temperature inside the room, and `user_temp` is a formal parameter that contains the temperature set by a user.)

```
P2: SPEC AG((Ti_temp > user_temp)
                    -> AF(AC_setMode=cooling))
P3: SPEC AG(!(Ti_temp > user_temp)
                    -> AF(AC_setMode=fan))
```

• Once the ventilator is turned on, it will keep operating until the inside temperature reaches the outside temperature. (`To_temp` represents the temperature outside the room.)

```
P4: SPEC AG(Ventilation_pwr=on ->A[Ventilation_pwr
    =on U !(Ti_temp > To_temp)])
```

Properties for Air-Cleaning Service:

• Whenever the sensor device senses smoke, the ventilator will be turned on and the window will be opened.

```
P5: SPEC AG(SmokeSensor_currentSmoke=1
                      -> AF(Ventilation_pwr=on))
P6: SPEC AG(SmokeSensor_currentSmoke=1
                      -> AF(Window_status=open))
```

• If the window is opened, then the window will keep open until the sensor no longer senses smoke. Similarly, once the ventilator is turned on, it will keep working until no smoke is detected.

```
P7: SPEC AG(Window_status=open -> A[Window_status
        =open U SmokeSensor_currentSmoke=0])
P8: SPEC AG(Ventilation_pwr=on -> A[Ventilation_pwr
        =on U SmokeSensor_currentSmoke=0])
```

We have verified these properties against individual services using a model checker SMV [14]. As a result, it was automatically proven that the HVAC service successfully conformed to P1 through P4 in all possible reachable states. Also, all properties from P5 through P8 became true for the Air-Cleaning service. The time taken for the verification was 1.2 sec. per each property on the average, with the PC with PentiumM 1.0Ghz (Memory 760MB, WinXP Pro). As a result, it was confirmed quite efficiently that our design of the individual services had no critical design flaw.

Interestingly however, the properties P4 and P8 became false when we ran the two services *in parallel*. The counter-example of P8 told that the ventilation is turned off although there is still smoke. This is because the HVAC turns off the ventilation, when the room temperature becomes lower than the outside temperature. The finding was observed due to a functional conflict among the services, which is well known as the feature interaction problem [5]. Developing a more systematic approach to detecting feature interaction using the model checking will be left to our future research.

7. Conclusion

This paper presented a formal framework to verify and validate integrated services of home network systems. Although not presented in the paper, we actually have described various services using the proposed language, such as the seven services presented in [10] and more including a security service and a power-saving service. Thus, we believe that the proposed language is expressive enough to describe practical services. Also, the proposed language enables a compact modeling independent of the underlying HNS protocols or specific platform. Hence, it could be helpful for developers to conduct a platform-independent modeling (PIM) of the MDA [12]. It is also interesting to develop a method to convert the proposed language into platform-specific workflow languages such as BPEL4WS.

Acknowledgments

This research was partially supported by Grant-in-Aid for Young Scientists (B)15700058, 2005 and the 21st Century Center of Excellence Program "New Information Technologies for Building a Networked Symbiotic Environment" of JSPS.

References

[1] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin and J. Reese, "Model checking large software specifications", *IEEE Transactions on Software Engineering*, 24(7): pp. 498-520, July 1998.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking,* MIT Press, 1999.

[3] Digital Living Network Alliance- http://www.dlna.org/

[4] ECHONET Consortium- http://www.echonet.gr.jp/

[5] "Feature Interaction in Telecommunications and Software Systems", Vol. I-VIII, IOS Press, 1992-2005.

[6] HAVi - http://www.havi.org/

[7] ITU-T Recommendation J.190, "Architecture of Media HomeNet that supports cable-based services", 2002.

[8] K. L. McMillan, "Symbolic Model Checking", Kluwer Academic Publishers, 1993.

[9] Jini - http://www.jini.org/

[10] M. Nakamura, H. Igaki, K. Matsumoto, "Feature Interactions in Integrated Services of Networked Home Appliance", *Proc. of Int'l. Conf. on Feature Interactions in Telecommunication Networks and Distributed Systems (ICFI'05)*, pp. 236-251, Jun. 2005.

[11] M. Nakamura, H. Igaki, H. Tamada and K. Matsumoto, "Implementing integrated services of networked home appliances using service oriented architecture", *Proc. of 2nd International Conference on Service Oriented Computing (ICSOC2004)*, pp.269-278, Nov. 2004.

[12] Object Management Group, "OMG Model Driven Architecture", http://www.omg.org/mda/

[13] OSGi Appliance, "The OSGi Service Platform", http://osgi.org.

[14] "The SMV System", http://www.cs.cmu.edu/~modelcheck/smv.html

[15] UPnP Forum - http://www.upnp.org/

Appendix

A. System Description for the Example HNS

```
SYSTEM my_home {

  TYPEDEF
    tPower       {ON,OFF} # power for all appliances
    tAC_Temp     {18..28};     # for AirConditioner
    tAC_Mode     {COOLING,FAN} # for AirConditioner
    tWindowStatus {OPEN,CLOSE};
    tTemp        {15..40};    # Temperature
    tSmoke       {0,1}        # Smoke

  ENVIRONEMT inside {   # Environment inside the room
!!  PROPERTY tTemp Temp_in;
             tTemp Temp_out;
             tSmoke Smoke;
  }

  APPLIANCE AirConditioner {
    PROPERTY
      tPower   power:=OFF;
      tAC_Temp TempSetting := 24;
        #Temperature Setting for air conditioner
      tAC_Mode ModeSetting := COOLING;
        #Mode Setting for air conditioner
    METHOD
      void ON() {
        PRE     true;
        POST    power=ON;}
      void OFF() {
        PRE     true;
        POST    power=OFF; }
      void setTemperature(tAC_Temp temp) {
        PRE     power=ON;
        POST    TempSetting=temp;
        ENV_W   env.Temp_in;}
      void setMode(tAC_Mode mode) {
        PRE     power =ON;
        POST    ModeSetting = mode;
        ENV_W   env.Temp_in;}
    }

  APPLIANCE Thermometer_inside {
    PROPERTY
      tPower power:=OFF;
      tTemp  CurrentTemp;
    METHOD
      void ON() {
        PRE     true;
        POST    power=ON;}
      void OFF() {
        PRE     true;
        POST    power=OFF;}
      tTemperature measureTemp() {
        PRE     power=ON
        POST    CurrentTemp=env.Temp_in;
        ENV_R   env.Temp_in;
        RETURN  CurrentTemp;}
  }

  APPLIANCE Thermometer_outside {
      :
  }

  APPLIANCE SmokeSensor {
    PROPERTY
      tPower power:=OFF;
      tSmoke CurrentSmoke:= 0;
    METHOD
      void ON() {
        PRE     true;
        POST    power=ON;}
      void OFF() {
        PRE     true;
        POST    power=OFF;}
      tSmoke deteckSmoke() {
        PRE     power=ON;
        POST    CurrentSmoke=env.Smoke;
        ENV_R   env.Smoke;
        RETURN  CurrentSmoke;}
  }

  APPLIANCE Ventilation {
    PROPERTY
      tPower power:=OFF;
    METHOD
      void ON() {
        PRE     true;
        POST    power=ON;
        ENV_W   env.Temp_in;}
      void OFF() {
        PRE     true;
        POST    power=OFF;
        ENV_W   env.Temp_in;}
  }

  APPLIANCE Window {
    PROPERTY
      tPower power := OFF;
      tWindowStatus WindowStatus := close;
    METHOD
      void ON() {
        PRE      true;
        POST     power=ON;}
      void OFF() {
        PRE     true;
        POST    power=OFF;}
      void OPEN() {
        PRE     power=ON;
        POST    WindowStatus=OPEN;
        ENV_W   env.Temp_in;}
      void CLOSE() {
        PRE     power=ON;
        POST    WindowStatus=CLOSE
        ENV_W   env.Temp_in;}
  }
}
```

B. Service Description of HVAC and Air-Cleaning

```
DEPLOYED_SYSTEM my_home;

SERVICE HVAC(tAC_Temp user_temp) {

  VAR
    tTemperature Ti_temp,To_temp; # Local variable
  APPLIANCE
    AirConditioner, Themometer_inside, Thermometer_outside,
    Ventilation;

  CONTENT
    WHILE (END()=0) {     # For repeatedly running
      Thermometer_inside.ON();
      Thermometer_outside.ON();
      Ti_temp := Thermometer_inside.measureTemp();
      To_temp := Thermometer_outside.measureTemp();
      AirConditioner.ON();
      AirConditioner.setTemperature(user_temp)

      WHILE (Ti_temp > user_temp) {
        AirConditioner.setMode('COOLING');
        IF (Ti_temp > To_temp)  {
          WHILE (Ti_temp > To_temp) {
            Ventilation.ON() ;
            Ti_temp := Thermometer_inside.measureTemp();
            To_temp := Thermometer_outside.measureTemp();
          }
          Ventilation.OFF() ;
        }
      }
      AirConditioner.setMode('FAN');
    }
    Thermometer_inside.OFF();
    Thermometer_outside.OFF();
    AirConditioner.OFF();
}


SERVICE Air_Cleaning() {
  VAR
    tSmoke Smoke_status;    # Local variable
  APPLIANCE
    SmokeSensor, Ventilation, Window;

  CONTENT
    WHILE (END()=0)  {      # For repeatedly running
      SmokeSensor.ON();
      Smoke_status := SmokeSensor.detectSmoke();
      WHILE (Smoke_status=0) {
        IF (END()=0)  {
        Smoke_status := SmokeSensor.detectSmoke();
        }
        ELSE {
         SmokeSensor.OFF();
         EXIT()             # Quit the service
        }
      }
      WHILE (Smoke_status=1) {
        Window.ON();
        Window.OPEN();
        Ventilation.ON();
        Smoke_status := SmokeSensor.detectSmoke();
      }
      Window.CLOSE();
      Window.OFF();
      Ventilation.OFF();
    }
    SmokeSensor.OFF();
}
```

C. SMV Program for HVAC Service

```
MODULE main
VAR
-- #### Appliance Property ####
-- Thermometer inside
  TM_inside_pwr:{on,off};           --Power
  TM_inside_currentTemp:{15,25,40}; --Current Temp.
-- Thermometer outside
  TM_outside_pwr:{on,off};          --Power
  TM_outside_currentTemp:{15,25,40};--Current Temp.
-- AirConditioner
  AC_pwr: {on,off};             --Power
  AC_set: {18,24,28};           --Temp. Setting
  AC_mode: {cooling,fan};       --Mode Setting
-- Ventilation
  Ventilation_pwr: {on,off};    --Power
--#### Environment Property ####
  ENV_T_in: {15,25,40};         --Temp. inside
  ENV_T_out: {15,25,40};        --Temp. outside

  user_temp: {18,24,28};        --formal parameter

HVAC1: process HVAC(user_temp,AC_pwr,AC_set,AC_mode,
TM_inside_pwr,TM_inside_currentTemp,TM_outside_pwr,
TM_outside_currentTemp,Ventilation_pwr,
ENV_T_in,ENV_T_out);

ASSIGN
  init(TM_inside_pwr):= off;
  init(TM_inside_currentTemp):= 25;
  init(TM_outside_pwr):= off;
  init(TM_outside_currentTemp):= 25;
  init(AC_pwr):=off;
  init(AC_set):=24;
  init(AC_mode):=cooling;
  init(Ventilation_pwr):= off;
  init(user_temp):= {18,24,28};
  init(ENV_T_in):= 25;
  init(ENV_T_out):= 25;
  next(ENV_T_in):= {15,25,40}; --Changes arbitrarily
  next(ENV_T_out):= {15,25,40};--Changes arbitrarily

-----------------------------------------------

--####  HVAC Service ####

MODULE HVAC(user_temp,AC_pwr,AC_set,AC_mode,Ti_pwr,
Ti_tmp,To_pwr,To_tmp,VN_pwr,ENV_T_in,ENV_T_out)

VAR
 pc:{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,25,26,27};
 active: {0,1};
 end:{0,1};
 Ti_temp: {15,25,40};           -- Local variable
 To_temp: {15,25,40};           -- Local variable


DEFINE
-- #pc=0 BEGIN PROGRAM
        begin_service :=1;

-- #pc=1 WHILE (END()=0)
        while1_cond     := (end=0) ;
        while1_cond_ng := !(end=0) ;

-- #pc=2 TM_inside.ON()
        pre2_Ti_ON      := 1;
        post2_Ti_ON     := (Ti_pwr=on);
        post2_Ti_ON_Ti_pwr      := on;

-- #pc=3 TM_outside.ON()
        pre3_To_ON      := 1;
        post3_To_ON     := (To_pwr=on);
        post3_To_ON_To_pwr      := on;

-- #pc=4 TM_inside.measureTemp()
        pre4_Ti_MT      := (Ti_pwr=on);
        post4_Ti_MT     := (Ti_tmp=ENV_T_in);
        post4_Ti_MT_Ti_tmp:= (ENV_T_in);

-- #pc=5 Ti_temp:= Ti_CurrentTemp -- Assignment
        pre5_ASM        := 1;
        post5_ASM       := 1;
        post5_ASM_Ti_temp := (Ti_tmp);

-- #pc=6 TM_outside.measureTemp()
        pre6_To_MT      := (To_pwr=on);
        post6_To_MT     := (To_tmp=ENV_T_out);
        post6_To_MT_Tp_tmp:= (ENV_T_out);

-- #pc=7 To_temp:= To_CurrentTemp -- Assignment
        pre7_ASM        := 1;
        post7_ASM       := 1;
        post7_ASM_To_temp := (To_tmp);

-- #pc=8 AC.ON()
        pre8_AC_ON      := 1;
        post8_AC_ON     := (AC_pwr=on);
        post8_AC_ON_AC_pwr := on;

-- #pc=9 AC.setTemperature(user_temp)
        pre9_AC_ST      := (AC_pwr=on);
        post9_AC_ST     := (AC_set=user_temp);
        post9_AC_ST_AC_tmp:= user_temp;

-- #pc=10 WHILE (Ti_temp > user_temp)
        while10_cond    := (Ti_temp > user_temp);
        while10_cond_ng:= !(Ti_temp > user_temp);

-- #pc=11 AC.setMode('cooling')
        pre11_AC_SM     := (AC_pwr=on);
        post11_AC_SM    := (AC_mode=cooling);
        post11_AC_SM_AC_mode := cooling;

-- #pc=12 IF (Ti_temp > To_temp)
        if12_cond       := (Ti_temp > To_temp);
        if12_cond_ng    := !(Ti_temp > To_temp);

-- #pc=13 WHILE (Ti_temp > To_temp)
        while13_cond    := (Ti_temp > To_temp);
        while13_cond_ng := !(Ti_temp > To_temp);

-- #pc=14 Ventilation.ON()
        pre14_VN_ON     := 1;
        post14_VN_ON    := (VN_pwr=on);
        post14_VN_ON_VN_pwr     := on;

-- #pc=15 TM_inside.measureTemp()
        pre15_Ti_MT     := (Ti_pwr=on);
        post15_Ti_MT    := (Ti_tmp=ENV_T_in);
        post15_Ti_MT_Ti_tmp:= (ENV_T_in);

-- #pc=16 Ti_temp:= Ti_CurrentTemp -- Assignment
        pre16_ASM       := 1;
        post16_ASM      := 1;
        post16_ASM_Ti_temp := (Ti_tmp);

-- #pc=17 TM_outside.measureTemp()
        pre17_To_MT     := (To_pwr=on);
        post17_To_MT    := (To_tmp=ENV_T_out);
        post17_To_MT_To_tmp:= (ENV_T_out);

-- #pc=18 To_temp:= To_CurrentTemp -- Assignment
        pre18_ASM       := 1;
        post18_ASM      := 1;
        post18_ASM_To_temp := (To_tmp);

-- #pc=19 WHILE dummy2
        while19_dummy   := 1;

-- #pc=20 Ventilation.OFF()
        pre20_VN_OFF    := 1;
        post20_VN_OFF   := (VN_pwr=off);
        post20_VN_OFF_VN_pwr := off;

-- #pc=21 WHILE dummy
        while21_dummy   :=1;

-- #pc=22 AC.setMode('fan')
        pre22_AC_SM     := (AC_pwr=on);
        post22_AC_SM    := (AC_mode=fan);
        post22_AC_SM_AC_mode:= fan;

-- #pc=23 WHILE dummy1
        while23_dummy   :=1;

-- #pc=24 TM_inside.OFF()
        pre24_Ti_OFF    := 1;
        post24_Ti_OFF   := (Ti_pwr=off);
        post24_Ti_OFF_To_pwr:= off;

-- #pc=25 TM_outside.OFF()
        pre25_To_OFF    := 1;
        post25_To_OFF   := (To_pwr=off);
        post25_To_OFF_To_pwr:= off;

-- #pc=26 AC.OFF()
        pre26_AC_OFF    := 1;
        post26_AC_OFF   := (AC_pwr=off);
        post26_AC_OFF_AC_pwr := off;

-- #pc=27 ENDING PROCESS
        end_service     := 1;

ASSIGN
    init(pc) := 0;
    init(active) :=0;
    init(end) := 0;
    init(Ti_temp) := 25;
    init(To_temp) := 25;

-- #### Transition Relations ####

    next(user_temp) := user_temp;
    next(Ti_pwr):=case
      (pc = 2) & pre2_Ti_ON : post2_Ti_ON_Ti_pwr;
      (pc = 24) & pre24_Ti_OFF : post24_Ti_OFF_To_pwr;
      1 : Ti_pwr;
      esac;

    next(Ti_tmp):= case
      (pc = 4) & pre4_Ti_MT : post4_Ti_MT_Ti_tmp;
      (pc = 15) & pre15_Ti_MT : post15_Ti_MT_Ti_tmp;
      1 : Ti_tmp;
      esac;

    next(Ti_temp):= case
      (pc = 5) & pre5_ASM : post5_ASM_Ti_temp;
      (pc = 16) & pre16_ASM : post16_ASM_Ti_temp;
      1 : Ti_temp;
      esac;

    next(To_pwr):= case
      (pc = 3) & pre3_To_ON : post3_To_ON_To_pwr ;
      (pc = 25) & pre25_To_OFF : post25_To_OFF_To_pwr ;
      1 : To_pwr;
      esac;

    next(To_tmp):= case
      (pc = 6) & pre6_To_MT : post6_To_MT_Tp_tmp;
      (pc = 17) & pre17_To_MT : post17_To_MT_To_tmp;
      1 : To_tmp;
      esac;

    next(To_temp):= case
      (pc = 7) & pre7_ASM : post7_ASM_To_temp;
      (pc = 18) & pre18_ASM : post18_ASM_To_temp;
      1 : To_temp;
      esac;

    next(AC_pwr):= case
      (pc = 8) & pre8_AC_ON : post8_AC_ON_AC_pwr;
      (pc = 26) & pre26_AC_OFF : post26_AC_OFF_AC_pwr;
      1 : AC_pwr;
      esac;

    next(AC_set):= case
      (pc = 9) & pre9_AC_ST : post9_AC_ST_AC_tmp;
      1 : AC_set;
      esac;

    next(AC_mode):= case
      (pc = 11) & pre11_AC_SM : post11_AC_SM_AC_mode;
      (pc = 22) & pre22_AC_SM : post22_AC_SM_AC_mode;
       1 : AC_mode;
       esac;

    next(VN_pwr):= case
      (pc = 14) & pre14_VN_ON : post14_VN_ON_VN_pwr;
      (pc = 20) & pre20_VN_OFF : post20_VN_OFF_VN_pwr;
       1 : VN_pwr;
       esac;

-- ###### Program Counter ######

    next(pc):= case
      (pc=0) : 1;
      (pc=1) & while1_cond            : 2;
      (pc=1) & while1_cond_ng         : 24;
      (pc=2) & pre2_Ti_ON             : 3;
      (pc=3) & pre3_To_ON             : 4;
      (pc=4) & pre4_Ti_MT             : 5;
      (pc=5) & pre5_ASM               : 6;
      (pc=6) & pre6_To_MT             : 7;
      (pc=7) & pre7_ASM               : 8;
      (pc=8) & pre8_AC_ON             : 9;
      (pc=9) & pre9_AC_ST             : 10;
      (pc=10) & while10_cond          : 11;
      (pc=10) & while10_cond_ng       : 22;
      (pc=11) & pre11_AC_SM           : 12;
      (pc=12) & if12_cond             : 13;
      (pc=12) & if12_cond_ng          : 21;
      (pc=13) & while13_cond          : 14;
      (pc=13) & while13_cond_ng       : 20;
      (pc=14) & pre14_VN_ON           : 15;
      (pc=15) & pre15_Ti_MT           : 16;
      (pc=16) & pre16_ASM             : 17;
      (pc=17) & pre17_To_MT           : 18;
      (pc=18) & pre18_ASM             : 19;
      (pc=19) & while19_dummy         : 13;
      (pc=20) & pre20_VN_OFF          : 21;
      (pc=21) & while21_dummy         : 10;
      (pc=22) & pre22_AC_SM           : 23;
      (pc=23) & while23_dummy         : 1;
      (pc=24) & pre24_Ti_OFF          : 25;
      (pc=25) & pre25_To_OFF          : 26;
      (pc=26) & pre26_AC_OFF          : 27;
      (pc=27) & end_service           : 0;
       1 : pc;
       esac;
    next(end):=case
       (pc=0) : {0,1};
       1 : end;
       esac;
    next(active):=case
       (pc=27) : 0;
       (pc=0) : 1;
       1 : active;
       esac;

 FAIRNESS running
 FAIRNESS (end=0 & pc=1)
```