Describing and Verifying Integrated Services of Home Network Systems

Pattara Leelaprute¹, Masahide Nakamura², Tatsuhiro Tsuchiya¹, Ken-ichi Matsumoto², Tohru Kikuno¹

¹Graduate School of Information Science and Technology, Osaka University, Japan
{pattara, t-tutiya, kikuno}@ist.osaka-u.ac.jp
²Graduate 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.
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 achieves energy-saving air-conditioning 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.

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 \( (P_d, M_d, Pre_d, Post_d) \), where

- \( P_d = \{ p_1, \ldots, p_n \} \) is a set of all properties of \( d \).
- \( M_d = \{ m_1, \ldots, m_k \} \) 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 over \( P_d \). \( a_{ij} \) is either a constant value or a formula over \( P_d \). 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).
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 $s = [c_1, c_2, \ldots, c_n]$, where $c_i$ is the current value of the property $p_i(e \in P_d)$. For a method $m \in M_d$, we say that $m$ is enabled under $s$ if $Pre_d(m)$ 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 $s' = [a_1, a_2, \ldots, a_n]$, as specified in $Post_d(m) = [p_{i_1} = a_{i_1} \land p_{i_2} = a_{i_2} \land \ldots]$. We assume that during the execution of $m$, no other methods $d$ 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 $D = \{d_1, d_2, \ldots, d_r\}$ be a set of appliances. Also, let $M = \bigcup_{d \in D} M_d$ 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.
- $D = \{d_1, d_2, \ldots, d_r\}$ 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.max, 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 env_name {
    # Environment definition
    ENVIRONMENT env_property1 [= init_val];
    env_property2 [= init_val];

  }

  APPLIANCE appliance_name1 {
    # Appliance definition
    APPLIANCE appliance_properties declaration
    type_name1 app_property1 [= init_val];
    type_name2 app_property2 [= init_val];

    METHOD # Method definition
    return_type1 method_name1([type_name formal_param]*){
      PRE pre_condition;
      POST post_condition;
      ENV env_name.property_name;
      ENV env_name.property_name;
      RETURN return_value;
    }

    return_type2 method_name2(...) {
    }
  }

  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 env_name.property_name;
    ENV env_name.property_name;
    RETURN return_value;
  }
```

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., ATemp + 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.

```
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;
  
  return_type2 method_name2(...) {
  }
}
```
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 |

```

- 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()*.

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:

```smv
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:

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

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-
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, \( \text{var}:=A.m1()+B.m2()+c; \) is translated into three statements below:

\[
\begin{align*}
& A.m1(); \\
& B.m2(); \\
& \text{var}:=\text{RET}(A.m1())+\text{RET}(B.m2())+c;
\end{align*}
\]

For the resultant statements, we put sequence numbers from \begin{dummy} \end{dummy} to \begin{dummy} \end{dummy}, as the unique labels. By convention, we assume that \begin{dummy} \end{dummy} and \begin{dummy} \end{dummy} are labeled by 0: and max: respectively. In the following, let \( \text{label}(stm) \) 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 \( L=\text{label}(stm) \). The first rule applies when \( stm \) is a method invocation \( m(). \) In DEFINE, pre/post conditions are described as they are. Moreover, a value of each property \( p \) is defined as \( \text{nextL}_p \), 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 \( \text{var} \). The last two rules apply to \( \text{if} \) and \( \text{while} \). 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 \( \text{pc} \) in \( smv \) as a program counter. First, we define a local variable \( \text{pc} \) in the \( svc \) module of \( smv \), as \( \text{VAR} \, \text{pc}:\{0..\text{max}\} \), and initialize as \( \text{ASSIGN} \, \text{init}(\text{pc}):=0; \). Suppose that \( \text{pc}=L \) when \( smv \) is currently simulating the \( L \)-th statement \( stm \) (i.e., \( \text{label}(stm)=L \)). Then, we want to update \( \text{pc} \) to an appropriate next value after \( stm \) is executed. For this, we present rules to compute the next value of \( \text{pc} \) 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 \( m() \) \( \text{preL}(\text{pc}) \) (see Table 1) must hold. In this case, the next value of \( \text{pc} \) 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 \( \text{exit}() \) \( \text{pc} \) 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:\( \text{ASSIGN} \, \text{next}(\text{pc}):=\text{case} \ldots \text{esac}. \)

**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 \( \text{nextL}_p:=a; \), it means that \( p \) is updated by the \( L \)-th statement, which is a method invocation \( m() \). Thus, \( p \) can be updated when \( \text{pc}=L \) 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:

\[
\text{ASSIGN} \, \text{next}(p):=\text{case} \ldots \text{esac};
\]

For each property \( p \) of \( svc \), we model the behavior of \( \text{end}() \) pseudo function. By definition, \( \text{end}() \) 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.

\[
\text{VAR} \, \text{end}:[0,1]; \quad \text{--Boolean for \text{end}() function} \\
\text{ASSIGN} \, \text{next}(\text{end}):=\text{case} \ldots \text{esac};
\]

\[
\text{VAR} \, \text{end}:[0,1]; \quad \text{--non-determinism in \text{L}th statement}
\]
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 eventually be 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


Appendix

A. System Description for the Example HNS

```plaintext
SYSTEM my_home {  
  TYPEDEF  
  tPower [ON,OFF] # power for all appliances  
  tAC_Temp [18..28] # for AirConditioner  
  tSmoke [0,1] # Smoke  
  tTemp [18..28] # Temperature  
  tWindowStatus [OPEN,CLOSE];  
ENVIRONEMT inside { # Environment inside the room  
  PROPERTY Temp_in;  
  tPower power;
}  
APPLIANCE AirConditioner {  
  PROPERTY  
  tPower power;  
  tAC_Temp TempSetting := 24;  
  # Temperature Setting for air conditioner  
  # for repeatedly running  
  # Mode Setting for air conditioner  
  METHOD  
  void ON() {  
    void OFF();  
  }  
  POST power=ON;  
  POST power=OFF;  
  POST WindowStatus=OPEN;  
  POST WindowStatus=CLOSE;  
}  
APPLIANCE Thermometer_outside {  
  PROPERTY  
  tTemp CurrentTemp;  
  METHOD  
  void ON() {  
    void OFF();  
  }  
  POST power=ON;  
  POST power=OFF;  
  POST WindowStatus=OPEN;  
  POST WindowStatus=CLOSE;  
}  
APPLIANCE Thermometer_inside {  
  PROPERTY  
  tTemp CurrentTemp;  
  METHOD  
  void ON() {  
    void OFF();  
  }  
  POST power=ON;  
  POST power=OFF;  
  POST WindowStatus=OPEN;  
  POST WindowStatus=CLOSE;  
}  
APPLIANCE SmokeSensor {  
  PROPERTY  
  tSmoke Smoke;  
  METHOD  
  void ON() {  
    void OFF();  
  }  
  POST power=ON;  
  POST power=OFF;  
  RETURN CurrentSmoke;  
}  
APPLIANCE Ventilation {  
  PROPERTY  
  tPower power;  
  METHOD  
  void ON() {  
    void OFF();  
  }  
  POST power=ON;  
  POST power=OFF;  
  POST WindowStatus=OPEN;  
  POST WindowStatus=CLOSE;  
}  
APPLIANCE HVAC(tAC_Temp user_temp) {  
  # Temperature Setting for air conditioner  
  # Local variable  
  METHOD  
  void ON() {  
    void OFF();  
  }  
  POST power=ON;  
  POST power=OFF;  
  RETURN CurrentSmoke;  
}  
APPLIANCE Window {  
  PROPERTY  
  tPower power;  
  METHOD  
  void ON() {  
    void OFF();  
  }  
  POST power=ON;  
  POST power=OFF;  
  RETURN CurrentSmoke;  
}  
APPLIANCE Air_Cleaning() {  
  # Local variable  
  METHOD  
  RETURN CurrentTemp;  
}  
APPLIANCE HVAC(tAC_Temp user_temp) {
  # Temperature Setting for air conditioner
  # Local variable
  METHOD
  RETURN CurrentTemp;
}  
APPLIANCE SmokeSensor {
  PROPERTY
  tSmoke Smoke;
  METHOD
  RETURN CurrentSmoke;
}  
APPLIANCE Ventilation {
  PROPERTY
  tPower power;
  METHOD
  RETURN CurrentSmoke;
}  
APPLIANCE HVAC(tAC_Temp user_temp) {
  # Temperature Setting for air conditioner
  # Local variable
  METHOD
  RETURN CurrentTemp;
}
```

B. Service Description of HVAC and Air-Cleaning

```plaintext
SERVICE HVAC(tAC_Temp user_temp) {  
  # Temperature Setting for air conditioner  
  # Local variable  
  APPLIANCE  
  AirConditioner, Thermometer_inside, Thermometer_outside,  
  Ventilation;  
  CONTENT  
  WHILE (END()==0) {  
    AirConditioner.setMode('COOLING');  
    IF (Ti_temp < To_temp) {  
      AirConditioner.setTemperature(Ti_temp);  
      Ventilation.ON();  
      Ti_temp := Thermometer_inside.measureTemp();  
      To_temp := Thermometer_outside.measureTemp();  
      Ventilation.OFF();  
    }  
  }  
  ELSE {  
    AirConditioner.OFF();  
    SmokeSensor.OFF();  
    Window.OPEN();  
    Window.ON();  
    AirConditioner.setMode('FAN');  
  }  
  EXIT();  
}  
SERVICE Air_Cleaning() {  
  VAR  
  tSmoke Smoke_status;  
  # Local variable  
  APPLIANCE  
  SmokeSensor, Ventilation, Window;  
  CONTENT  
  WHILE (END()==0) {  
    SmokeSensor.detectSmoke();  
    IF (Smoke_status==1) {  
      SmokeSensor.setMode('FAN');  
      Ventilation.ON();  
      Ventilation.OFF();  
      Window.OFF();  
      SmokeSensor.OFF();  
    }  
  }  
  EXIT();  
}  
```
C. SMV Program for HVAC Service

MODULE HVAC

VAR

user_temp: {18, 24, 28}; --formal parameter
Ventilation_pwr: {on, off}; --Power
AC_set: {18, 24, 28}; --Temp. Setting
AC_pwr: {on, off}; --Power
TM_outside_currentTemp:{15, 25, 40}; --Current Temp.
TM_inside_currentTemp: {15, 25, 40}; --Current Temp.

ASSIGN

init(TM_inside_pwr):= off;
init(TM_outside_pwr):= off;
init(Ventilation_pwr):= off;
init(Ti_pwr):= on;
init(To_pwr):= on;
init(VN_pwr):= on;
init(Ti_currentTemp):= 25;
init(To_currentTemp):= 25;
init(TM_inside_currentTemp):= 25;
init(TM_outside_currentTemp):= 25;


### HVAC Service ###

--- HVAC Service ---

MODULE HVAC(user_temp, AC_pwr, AC_set, AC_mode, Ti_pwr, To_pwr, VN_pwr, VN_set, VN_mode, TM_inside_pwr, TM_inside_currentTemp, TM_outside_pwr, TM_outside_currentTemp, Ventilation_pwr, ENV_T_in, ENV_T_out)

VAR

Ti_tmp, To_pwr, To_temp, VN_pwr, ENV_T_in, ENV_T_out)

MODULE HVAC(user_temp, AC_pwr, AC_set, AC_mode, Ti_pwr, To_pwr, VN_pwr, VN_set, VN_mode, TM_inside_pwr, TM_inside_currentTemp, TM_outside_pwr, TM_outside_currentTemp, Ventilation_pwr, ENV_T_in, ENV_T_out)

VAR

Ti_tmp, To_pwr, To_temp, VN_pwr, ENV_T_in, ENV_T_out)

MODULE HVAC(user_temp, AC_pwr, AC_set, AC_mode, Ti_pwr, To_pwr, VN_pwr, VN_set, VN_mode, TM_inside_pwr, TM_inside_currentTemp, TM_outside_pwr, TM_outside_currentTemp, Ventilation_pwr, ENV_T_in, ENV_T_out)

VAR

Ti_tmp, To_pwr, To_temp, VN_pwr, ENV_T_in, ENV_T_out)

MODULE HVAC(user_temp, AC_pwr, AC_set, AC_mode, Ti_pwr, To_pwr, VN_pwr, VN_set, VN_mode, TM_inside_pwr, TM_inside_currentTemp, TM_outside_pwr, TM_outside_currentTemp, Ventilation_pwr, ENV_T_in, ENV_T_out)

VAR

Ti_tmp, To_pwr, To_temp, VN_pwr, ENV_T_in, ENV_T_out)

MODULE HVAC(user_temp, AC_pwr, AC_set, AC_mode, Ti_pwr, To_pwr, VN_pwr, VN_set, VN_mode, TM_inside_pwr, TM_inside_currentTemp, TM_outside_pwr, TM_outside_currentTemp, Ventilation_pwr, ENV_T_in, ENV_T_out)

VAR

Ti_tmp, To_pwr, To_temp, VN_pwr, ENV_T_in, ENV_T_out)

MODULE HVAC(user_temp, AC_pwr, AC_set, AC_mode, Ti_pwr, To_pwr, VN_pwr, VN_set, VN_mode, TM_inside_pwr, TM_inside_currentTemp, TM_outside_pwr, TM_outside_currentTemp, Ventilation_pwr, ENV_T_in, ENV_T_out)

VAR

Ti_tmp, To_pwr, To_temp, VN_pwr, ENV_T_in, ENV_T_out)

MODULE HVAC(user_temp, AC_pwr, AC_set, AC_mode, Ti_pwr, To_pwr, VN_pwr, VN_set, VN_mode, TM_inside_pwr, TM_inside_currentTemp, TM_outside_pwr, TM_outside_currentTemp, Ventilation_pwr, ENV_T_in, ENV_T_out)

VAR

Ti_tmp, To_pwr, To_temp, VN_pwr, ENV_T_in, ENV_T_out)

MODULE HVAC(user_temp, AC_pwr, AC_set, AC_mode, Ti_pwr, To_pwr, VN_pwr, VN_set, VN_mode, TM_inside_pwr, TM_inside_currentTemp, TM_outside_pwr, TM_outside_currentTemp, Ventilation_pwr, ENV_T_in, ENV_T_out)

VAR

Ti_tmp, To_pwr, To_temp, VN_pwr, ENV_T_in, ENV_T_out)

MODULE HVAC(user_temp, AC_pwr, AC_set, AC_mode, Ti_pwr, To_pwr, VN_pwr, VN_set, VN_mode, TM_inside_pwr, TM_inside_currentTemp, TM_outside_pwr, TM_outside_currentTemp, Ventilation_pwr, ENV_T_in, ENV_T_out)