Specification of a Component-based Domotic System to Support User-Defined Scenarios

Fady Hamoui[1], Marianne Huchard[2], Christelle Urtado[1], Sylvain Vauttier[1]

[1] Lgi2p / Ecole des Mines d'Alès
Nîmes – France
<First>.<Last>@ema.fr

[2] Lirmm, Umr 5506, Cnrs and Univ. Montpellier 2
Montpellier – France
huchard@lirmm.fr

Abstract

Many studies have been conducted in order to develop systems that respond to user goals in domotic environments. These systems generally offer predefined scenarios corresponding to general goals and enable users to select those they want to trigger. We claim that such behaviors cannot be hardwired: user scenario definition should be supported. In this article, we propose the specification of a component-based domotic system that tackles this issue. This system offers users high level Guis to define their own scenarios from functionalities of the devices detected in the environment. These scenarios are automatically implemented: components are generated from device descriptors, assembled and the resulting software is run.

1 Introduction

Domotic environments are composed of electrical / electronic devices controlled by a domotic system that uses software and communication technologies to have the devices satisfy user goals. Each device provides control services. Each service in turn offers functionalities. Some devices also emit events that reflect a change of their parameter values. A domotic system can thus be seen as both a set of distributed services and a system that manipulates these services to achieve user goals. Users may simply use existing functionalities or need to combine them in a more complex scenario. Domotic environments can be used in many different ways. As it is not possible to hardwire all possible user scenarios, users must be given the capability to define their own scenarios. Furthermore, scenario integration should be automatic and dynamic, in order not to interrupt system execution [7, 10] and scenario execution must rely on some technical solution that coordinates service executions. Service Component Architecture [18] is a good candidate. It provides a model for the composition of services. Software components are the best means to implement services. They expose interfaces that describe functionalities, each of which represents a service [5]. Components can be dynamically assembled to achieve dynamic service connection [5, 7, 16, 9] without disrupting system execution [6]. Such systems have been developed by industry or academics. They are generally included in a fixed or mobile housing of control that allows to act on home devices such as shutters and lights. Few of them support complex user goals: most are exclusively based on predefined scenarios. Our goal is to specify a self-configurable system that runs (combinations of) services available on nearby devices. This system should seamlessly integrate user-defined scenarios without disrupting its execution. Users should easily express their goal scenarios with a dedicated end-user language [4]. The remainder of this article is as follows. Section 2 lists qualities that we expect from a domotic system. Section 3 describes our domotic system from the user point of view: it shows how users can describe their own scenarios. Sections 4 and 5 further describe our system by respectively providing its meta-model (as its structural view) and its process-oriented two-phased description (as its dynamical view). Section 6 compares to existing proposals while Sect. 7 concludes and draws perspectives to this work.

2 Target qualities for domotic systems

Let us consider a domotic environment composed of shutter, radiator and clock devices. The clock, for example, provides a service to set or get time and an event that indicates time change and contains the new time. Users must have the capability to define the following

evening scenario: *at 07:00 PM, if the living-room temperature is below 17°C, the shutter should be closed and the radiator turned on at level 6*. In order to support such scenarios, domotic systems should have the following qualities [3, 2]. *Decentralization* makes the software spatial structure stick to the physical distribution of devices. It also increases software quality and availability by distributing the load on several units and reducing the impact of failures. *Ability to define goals* allows users to add custom scenarios at any time. *Dynamic evolution* makes the system reactive to changes without impacting service continuity. *Autonomicity* limits user intervention to scenario definition: technical steps that implement scenarios are under system responsibility.

3 User goal-oriented functions

To meet these requirements, our system is composed of software agents built from software components. Agents are autonomous and collaborative entities. They have a flexible internal structure that allows dynamic (re)configuration through the (re)assembly of components. We identified two types of agents: GUI agents and device control agents (DCAs). GUI agents are a software mediator between devices and users. They enable users to customize the domotic system and define their goals. DCAs are responsible for the detection of devices that are available in the environment and for the execution of user-defined scenarios through their ability to control devices. Users can explicit their goals using services provided by the available devices by either selecting a particular service or defining a complex scenario. To do so, GUI agents make graphical user interfaces that represent the domotic environment available to users. These GUIs are automatically generated from the descriptors of detected devices.

3.1 Service selection

Using the dedicated GUI, users can select a device to display its provided services and select one. Each service in turn offers a set of parameterized operations. Users select such an operation and provide adequate parameter values. The system then invokes the required functionality. For example, the user can select the *clock* to view its provided services. A single service is available that provides the *set time* and *get time* operations. The choice of the *set time* operation enables the user to specify the new time as shown on the simple GUI of Fig. 1.

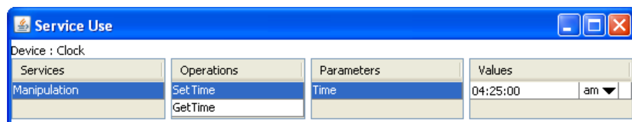

Figure 1. Service selection GUI

3.2 User scenario definition

Using the dedicated GUI, users can define new complex scenarios. A scenario is defined by several Event / Condition / Action (ECA) rules [12] that combine various operations. ECA rules enable the coordination of services as they are active (their execution is automatically triggered), express alternatives (with their condition clause) but are declarative (easier to read) and still interpretable [12]. Users must successively define the three clauses of the new rule as illustrated by Fig. 2 for the *evening scenario* example.

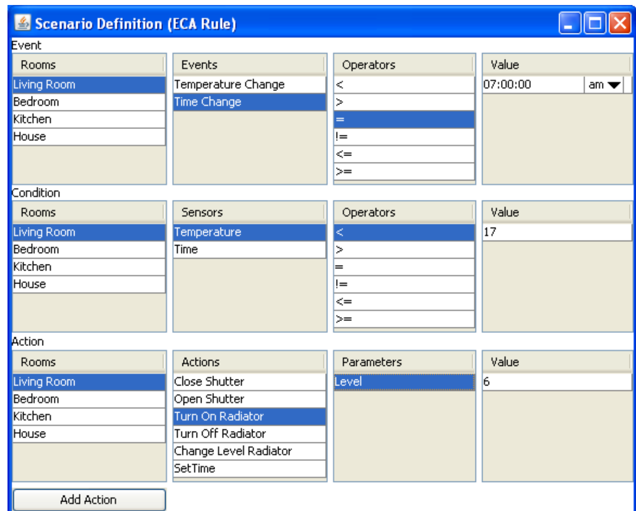

Figure 2. Scenario definition GUI

Event clause. An event is a pre-condition for triggering action executions. The GUI displays a list of all available events (the sum of all events that can be emitted by all detected devices) among which users choose the one that suits their needs. In the example, the event is *at 07:00 PM* and is obtained by comparing with the = operator the *07:00 PM* parameter value to the time provided by the *Time change* event of the living room.

Condition clause. The condition clause defines in which cases the rule will be triggered. A condition is a boolean function with at least two parameters, provided by either the user or measurement functions offered by sensor devices. Thus, the GUI displays all measure services available. The user chooses such a

service and a comparison operator. He then provides a value to compare to. In the example, the condition is *if the temperature in the living room is below 17°C* where *the temperature of the living-room* is provided as a service by a sensor device, < is a comparison operator and *17°C* is a user-provided parameter value.

Action clause. The action clause contains one or more service operations that perform actions on devices. The user selects a device, chooses an available operation and, if needed, specifies values for its parameters. In the example, the actions are *the shutter should be closed* (no parameter) and *the radiator should be turned on at level 6* (*6* is a parameter value).

At the end of scenario definition, a *coordination descriptor* is generated that contains data relative to the ECA rule. It is stored by the GUI agent and assigned to the DCA that will implement it. The system also contains predefined (or previously defined) scenarios users can execute directly.

4 Meta model of the component-based domotic system

This section aims to describe more precisely the proposed system by providing its meta-model. For readability's sake, the meta-model representation is divided into three views. The first view presents our service typology and the correspondence between services and ECA rule clauses. The second view shows what scenarios are and how services that compose a scenario are advertised in the service directory. The last view is devoted to showing how the agents that compose the system are made from software components and component connections.

4.1 Service typology

We have identified five service categories (see Fig. 3). *Sensor services* provide measures that come from sensor devices. They are used to provide measures in the condition clause of rules. *Event services* are used in event clauses: they provide events emitted by sensor devices and allow to detect changes in the environment. *Action services* are used in the action clauses: they perform operations on actuator devices thus providing services to users. *Comparison services* are used in condition clauses: they are (mostly predefined) technical services that provide comparison methods for all primitive types. *Coordination services* enable scenario execution. As scenarios are defined by ECA rules, they integrate a rule execution engine.

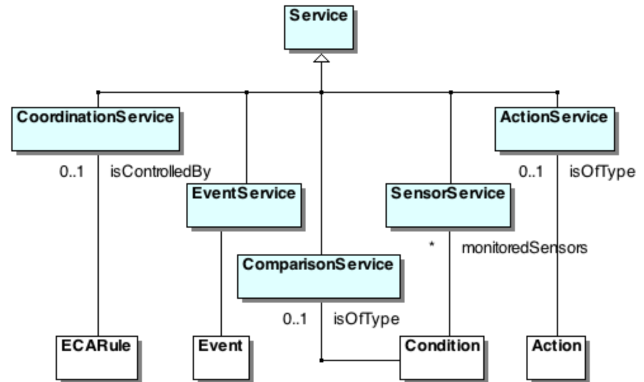

Figure 3. Service typology

4.2 Service directory and user scenarios

Agents have access to a service directory that enables inter-agent cooperation without hard-coding the underlying dependencies (decoupling). This directory (see Fig. 4) contains information on:
- *services* offered by devices of the environment (*Service* class). A service has a single type, represented by the *Interface* class. It can be provided by several service providers. Each service provider is bound to concrete component interfaces (often represented as a *lollipop* as shown on Fig. 6), represented by the *FunctionalInterface* class.
- *events* emitted by devices of the environment. They are represented by the *EventService* class as a specialization of the *Service* class. As for general services, an event is of a certain event type and can be provided by several event providers. Each event provider is bound to concrete event interfaces (sometimes represented as a *triangle* as shown on Fig. 6), represented by the *EventInterface* class.
- *parameter types* encountered in operations offered by devices. They are represented by the *ParamTypeInfo* class. Primitive types (numbers, strings, dates, times, etc.) are instances of the *ParameterType* class.
User scenarios (see Fig. 4) are defined by one or more ECA rules that are composed of an event, a condition and of one or more actions. These clause elements refer to the corresponding service advertisements (see Fig. 3) and are further mapped to corresponding concrete component interfaces (events to event interfaces, conditions and actions to functional interfaces) when service providers have been found / chosen for each necessary service. Parameter values defined by users during rule condition or action definition are mapped to parameter type information from the directory. When the condition clause is built from measure services (as

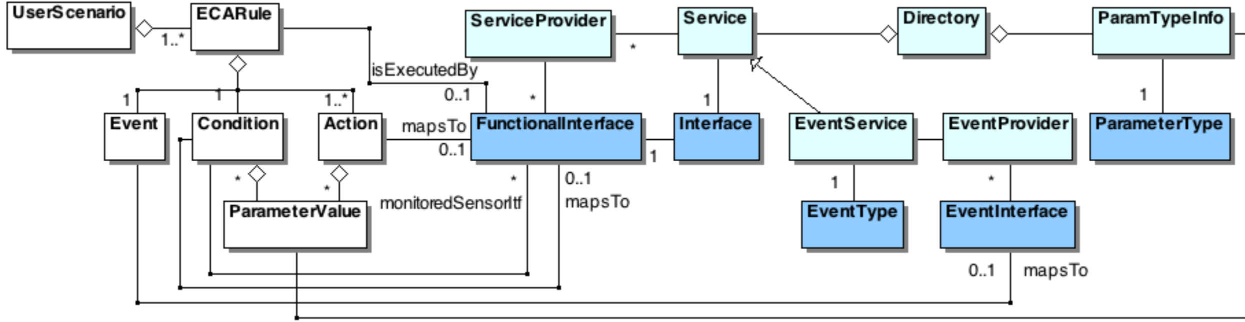

Figure 4. Service directory and user scenarios

results of some sensor service execution) the condition clause of the rule is linked to the functional interface that corresponds to this sensor service.

4.3 Component-based software agents

In our proposal, agents are built from components. The meta-model proposed here for component assembly (see Fig. 5) inspires from our previous work on self-assembling components [9, 8]. The two agent categories we have identified in our system (GUI agents and DCAs) specialize general agents made from components (*Agent* abstract class). Each agent has access to a service directory. GUI agents enable users to define their goals in the form of scenarios (stored as their *scenarioList*). DCAs detect services and events provided by available devices and execute scenarios. To do so, DCAs are composed of four types of components that mostly follow the typology of services provided in Sect. 4.1. *Sensor components* retrieve measures provided by sensor devices. *Action components* perform actions on actuator devices. *Comparison components* execute comparison services on primitive parameter types. Finally, *coordination components* control and coordinate the three previous component types to execute a scenario. These components all are generated by DCAs from device descriptors and built-in information on data types. Software components export their requirements and provisions through interfaces, represented by the *ConnectableInterface* class. Two components interact through the assembly of two interfaces, one provided by a component and the other required by the second component. Components export both functional interfaces and event interfaces (modeled as specializations of the *ConnectableInterface* class). Whatever its direction, the type of a functional interface is defined by an interface (*Interface* class) that can be compared to those of Java. These interfaces group operation declarations (modeled by the *Operation* class) each of which involves any number of input parameter types and at most an output type. Sensor components export one or more provided event interfaces. Whatever its direction, an event interface is typed by an event type. An event interface is a channel through which events are emitted when the value measured by some sensor changes. The event contains the new value. Coordination components export one or more required functional interfaces and an event interface.

5 Domotic system dynamics

The dynamics of the domotic system can schematically be decomposed into two phases.

Self-configuration phase. This phase consists in the detection of available devices to set up the system and maintains accurate information on devices. Each device is described by a descriptor which contains information on services and events they provide. DCAs download these descriptors and extract the information needed to generate sensor and action components. Then, they advertise information on the services and events provided by each component into the directory. This self-configuration phase executes autonomically at system startup and re-executes periodically to detect device or service addition or removal.

Self-assembly phase. This phase translates user scenario definitions into operational component assemblies that implement the scenarios. After a scenario is defined, the corresponding coordination descriptor is sent to a DCA for it to parameterize the rule execution engine of a corresponding coordination component. Then, the coordination component is assembled to the declared sensor, action and comparison components. Once the assembly achieved, the scenario is activated. The coordination component then listens to events, is able to retrieve values from its sensor compo-

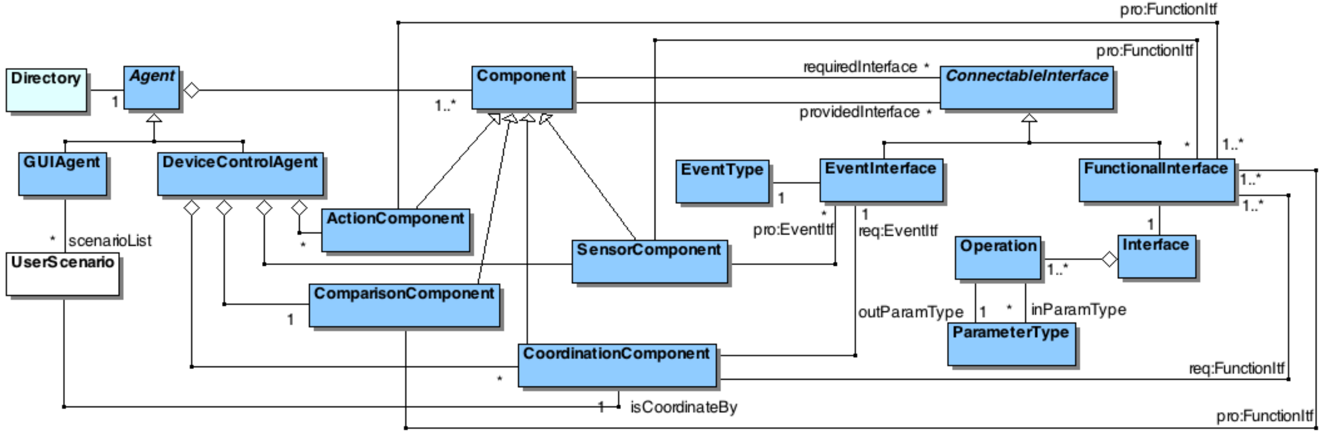

Figure 5. Agent and component typology and component external description

nents, compute the value of the condition with its comparison component and execute the prescribed actions thanks to its action components. The coordination descriptor of the *evening scenario* contains the information necessary for the DCA to parameterize the coordination component and assemble it to the *Clock*, *Radiator*, *Shutter* and comparator components. The descriptor and resulting assembly are presented in Fig. 6.

6 State of the art

Component models. Sensor Beans (SB) [13] and SOFA's [17] component models are close to ours. They both propose a typology of interfaces and oppose to our approach that relies on a component typology. Our action components nonetheless have interfaces that correspond to *Service* (SB) and *CSProcCall* interfaces (SOFA) while our sensor components have interfaces that correspond to *Event* and *Producer/Consumer* (SB) and *EventPassing* and *DataStream* interfaces (SOFA). Our proposal is thus comparable as for the syntactic richness of interfaces but further adds semantics through a component typology.

Domotic systems. Existing domotic systems fit into three categories. *Predefined Scenario Systems* contain centralized systems based on predefined scenarios. In [2, 3, 10], the only capability offered to users is to choose the scenarios they want to execute. The implementation of a scenario generally consists in assembling existing components. [2, 3] provide a slightly more general architecture: new components are generated as bridges, to enable interoperability between various technologies. In our proposal, the components that are generated are not dedicated to satisfying technical purposes but to meeting new user goals (they encompass some semantics on the system). To conclude, to our opinion, predefined scenarios are not sufficient to cover all possible situations and meet all user-goals: they are not change-resistant as any unforeseen change requires the intervention of an expert user. *Service Control Systems* [1, 11, 14, 19] allow users to control available services. They automatically detect devices in their environment and build a user GUI that lists the services provided by the detected devices. This capability is very close to the service selection GUI provided in our system. The user interacts with the system through this GUI to trigger service executions but cannot define complex scenarios. Among them, [19] nonetheless allows to define simple scenarios as service sequences. *Scenario Definition Systems* enables users to define their own scenarios. [7] offers a tool to define scenarios that is designer-oriented and does not allow runtime scenario definition. Similarly to our proposal, [15] provides users with a GUI for scenario definition and execution. However, scenarios seem restricted to sequences of service calls: they do not propose conditional executions as ECA rules do. Moreover, there is no possibility for users to dynamically define service parameters in their scenario scripts.

7 Conclusion and perspectives

In this paper, we described the specification of a domotic system that enables users to define their own goals. The system consists of a set of component-based agents. It automatically detects services and events offered by available devices and includes them in a GUI that represents the environment. Users can use a simple service implemented by a generated component or define a scenario represented by a new (automatically produced) component assembly formed by generated

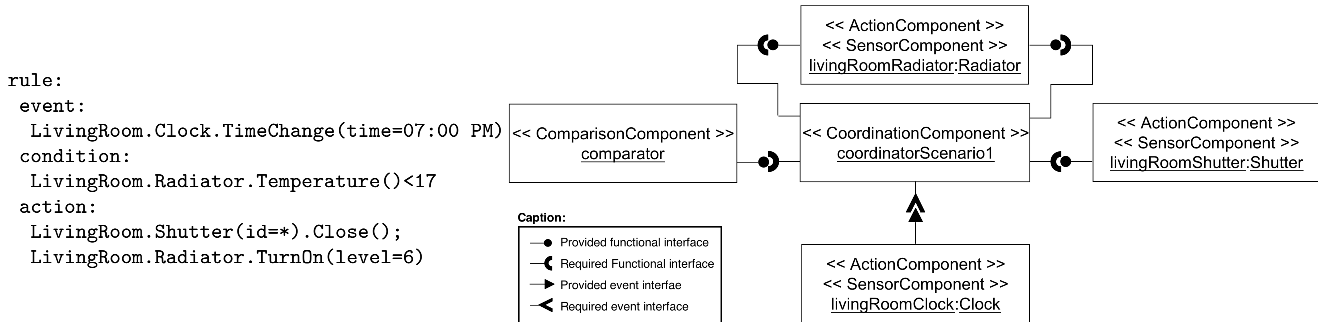

Figure 6. Evening scenario rule descriptor and component assembly

components. The system is under development using the OSGi and UPnP standards[1]. In the future, it will be enhanced with new component types that implement scenario conflict management, fault tolerance policies, automatic service adaptation, etc.

References

[1] S. Berger, H. Schulzrinne, S. Sidiroglou, and X. Wu. Ubiquitous computing in home networks. *IEEE Communications Magazine*, 41(11):128–135, Nov. 2003.

[2] A. Bottaro, A. Gerodolle, and P. Lalanda. Pervasive service composition in the home network. In *21st Int. Conf. on AINA*, Niagara Falls, Canada, pp 596–603, May 2007. IEEE.

[3] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, and C. Marin. A dynamic-SOA home control gateway. In *IEEE Int. Conf. on SCC*, Chicago, USA, pp 463–470, Sept. 2006.

[4] M. Burnett, S. K. Chekka, and R. Pandey. FAR: An end-user language to support cottage e-services. In *Proc. IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, Stresa, Italy, pp 195–202, May 2001.

[5] H. Cervantes and R. S. Hall. Automating service dependency management in a service-oriented component model. In *6th Wkshp on CBSE*, Portland, USA, May 2003.

[6] Y. Charif-Djebbar and N. Sabouret. Dynamic service composition and selection through an agent interaction protocol. In *IEEE/WIC/ACM Int. Conf. on WI-IAT*, Hong Kong, pp 105–108, Dec. 2006.

[7] D. Cheung-Foo-Wo, J.-Y. Tigli, S. Lavirotte, and M. Riveill. Wcomp: a multi-design approach for prototyping applications using heterogeneous resources. In *IEEE Int. Wkshp on RSP*, Chania, Crete, pp 119–125, 2006.

[8] N. Desnos, M. Huchard, G. Tremblay, C. Urtado, and S. Vauttier. Search-based many-to-one component substitution. *Journal of Software Maintenance and Evolution, Wiley*, 20(5):321–344, Sept./Oct. 2008.

[9] N. Desnos, S. Vauttier, C. Urtado, and M. Huchard. Automating the building of software component architectures. In *3rd Europ. Wkshp on EWSA*, Nantes, France, LNCS, 4344:228–235, Sept. 2006. Springer.

[10] G. Grondin, N. Bouraqadi, and L. Vercouter. MaDcAr: An abstract model for dynamic and automatic (re-)assembling of component-based applications. In *9th Int. Symp. on CBSE*, Västerås, Sweden, LNCS, 4063:360–367, June 2006. Springer.

[11] H. Ishikawa, Y. Ogata, K. Adachi, and T. Nakajima. Building smart appliance integration middleware on the OSGi framework. In *7th IEEE Int. Symp. on ISORC*, Vienna, Austria, pp 139–146, May 2004.

[12] J.-Y. Jung, J. Park, S.-K. Han, and K. Lee. An ECA-based framework for decentralized coordination of ubiquitous web services. *Information & Soft. Tech.*, 49(11-12):1141–1161, Nov. 2007.

[13] C. Marin and M. Desertot. Sensor bean: a component platform for sensor-based services. In *3rd Int. Wkshp on MPAC*, New York, USA, pp 1–8, 2005. ACM.

[14] K. Matsuura, T. Hara, A. Watanabe, and T. Nakajima. A new architecture for home computing. In *IEEE Wkshp on WSTFES*, Washington, USA, pp 71–74, May 2003.

[15] M. Nakamura, H. Igaki, H. Tamada, and K. ichi Matsumoto. Implementing integrated services of networked home appliances using service oriented architecture. In *2nd Int. Conf. on SOC*, New York, USA, pp 269–278, Nov. 2004. ACM.

[16] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing special section. *Communications of the ACM*, 46(10):24-28, Oct. 2003.

[17] F. Plásil, D. Bálek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. *Int. Conf. on CDS*, pp 43–52, 1998.

[18] SCA Consortium. Building systems using a service oriented architecture. Whitepaper available from www-128.ibm.com/developerworks/library/specification/ws-sca/ [Last checked 2009-04-30], 2005.

[19] C.-L. Wu, C.-F. Liao, and L.-C. Fu. Service-oriented smart-home architecture based on OSGi and mobile-agent technology. *IEEE Trans. on SMC, Part C*, 37(2):193–205, 2007.

[1] http://www.osgi.org and http://www.upnp.org.