

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

Fady Hamoui, Christelle Urtado, Sylvain Vauttier
LGI2P / Ecole des Mines d'Alès
Parc scientifique G. Besse
F30035 Nîmes - France

<First>.<Last>@ema.fr

Marianne Huchard
LIRMM, UMR 5506
CNRS and Univ. Montpellier 2
34095 Montpellier, France

huchard@lirmm.fr

Abstract

Many studies have been conducted in order to develop systems that respond to user requirements in domotic environments. These systems generally offer predefined scenarios corresponding to general requirements and enable users to select those he/she wants 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 automatically detects available devices in the environment and offers users high level GUIs to define their own scenarios from functionalities of the detected devices. These scenarios are then automatically implemented by the system: components are generated from device descriptors, assembled as prescribed by the scenarios and the resulting software is run by a rule execution engine.

1 Introduction

Domotic environments are composed of electrical / electronic devices connected to a network and controlled by a domotic system that uses software and telecommunication technologies to have the devices satisfy user requirements. Each device provides a set of services. Each of them in turn offers a set of functionalities. A domotic system can thus be seen as both a set of distributed services and a system that manipulates these services to achieve user requirements. Users may simply use existing functionalities or need to combine them in a more complex scenario.

The context of use of a domotic environment varies from a user to another. As it is not possible to hardwire all possible user scenarios, users must be given the capability to define their own scenarios. Further-

more, scenario integration in the system should be automatic and dynamic, in order not to interrupt system execution [7, 11] and scenario execution must rely on some technical solution that coordinates the execution of several services.

Service Component Architecture (SCA) [20] is a good candidate solution to this issue. It provides a model for the composition of services used to build applications based on a Service Oriented Architecture (SOA) [18]. Software components are the best means to achieve this. They expose interfaces that describe functionalities [21, 10]. Each interface represents a service [5]. Software components can be dynamically assembled to achieve dynamic service connection [5, 7, 17, 10]. This enables to assemble components during the execution of the system without disrupting it [6].

Such systems have been developed by industry or academics (see Sect. 6). The systems are generally included in a fixed or mobile housing of control that allows to act on home devices such as shutters and lights. Few systems support more complex users requirements, but are based on predefined scenarios and do not offer the capability to add new scenarios.

Our goal is to specify a self-configurable system [18] that handles the services of devices discovered in its environment. This system should also be autonomous to seamlessly integrate user-defined scenarios without disrupting its execution.

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 state-of-the-art proposals while Section 7 concludes and draws perspectives to this work.

2 Target qualities for domotic systems

Let us consider a domotic environment composed of the shutter, radiator and clock devices. All devices provide services to control them. Some also emit events that reflect a change of their parameter values. For instance, the clock provides a service to set or get time and an event that indicates time change and contains the new time. In this context, the user 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 [4, 3]:

- *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 requirements* allows users to add custom scenarios at any time.
- *Dynamic evolution* makes the system reactive to changes without impacting service continuity.
- *Autonomy* limits user intervention to scenario definition. Technical steps that are needed to implement the defined scenarios (component generation and system assembly and deployment) should be supported without human intervention.

3 User requirement-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 requirements. 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 requirements 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.

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. This GUI is automatically generated from the descriptors of detected devices.

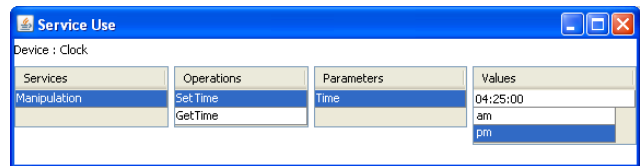


Figure 1. Service selection GUI

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 [1, 8] that combine various operations. ECA rules enable the coordination of services as they are active (their execution is automatically triggered by event occurrence), express alternatives (with their condition clause) but are declarative (easier to read) and still interpretable [13]. Users must successively define the three clauses of the new rule as illustrated by Fig. 2 for the *evening scenario* example.

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 user choose the one that suits their needs. In the *evening scenario* example, the event would be *at 07:00 PM* and would be obtained by comparing with the = operator the *07:00 PM* parameter value to the time provided by the *Time change* event.

Condition clause. The condition clause defines in which cases the rule will be triggered. It is not mandatory. A condition is a boolean function with at least two parameters, provided by either the user or measurement functions offered by sensor devices. Thus,

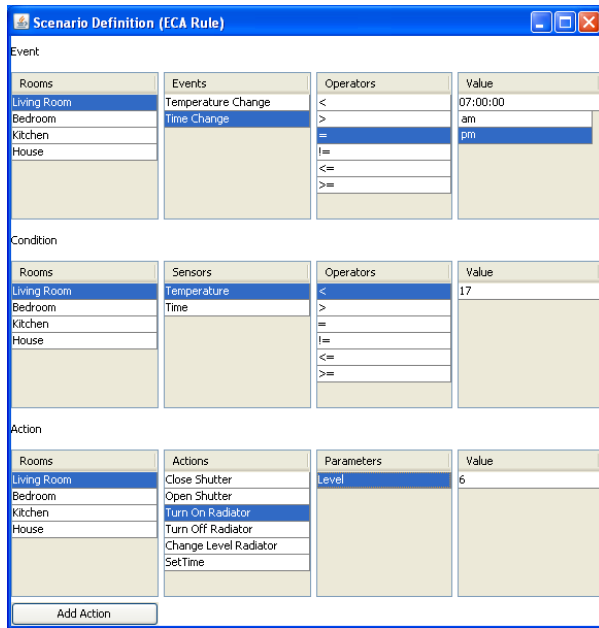


Figure 2. Scenario definition GUI

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 *evening scenario* example, the condition would be *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, \leq is a comparison operator and 17 °C is a parameter value provided by the user.

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. This process can be repeated as needed. In the *evening scenario* example, the actions would be *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 a scenario definition, a *coordination descriptor* is generated. It contains data relative to the ECA rule that corresponds to the scenario. This descriptor stored by the GUI agents and is assigned to a device control agent that implements the scenario. The system also contains predefined (or previously defined) scenarios users can choose to 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* provide events that come from sensor devices. They are used to detect changes in the environment and thus form the event clause of rules. *Action services* perform operations on actuator devices thus providing services to users. They are used in the action clause of rules. *Comparison services* are technical services that provide (a wide range of) comparison methods for all primitive types. These services are mostly predefined and participate in the definition of condition clauses. *Coordination services* enable scenario execution. As scenarios are defined by ECA rules, they integrate a rule execution engine.

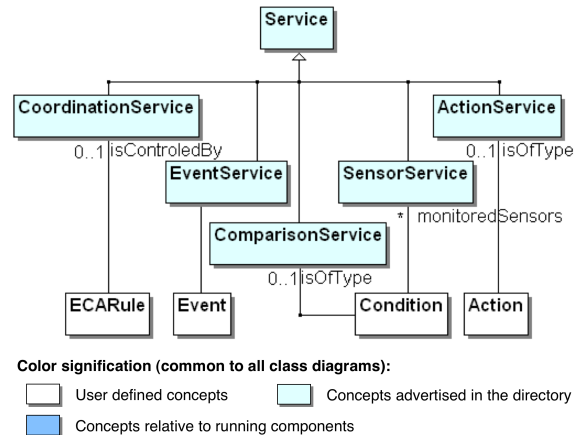


Figure 3. Service typology

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 the services* offered by devices of the environment (*Service* class). A service has a single type, represented by the *Interface* class. It

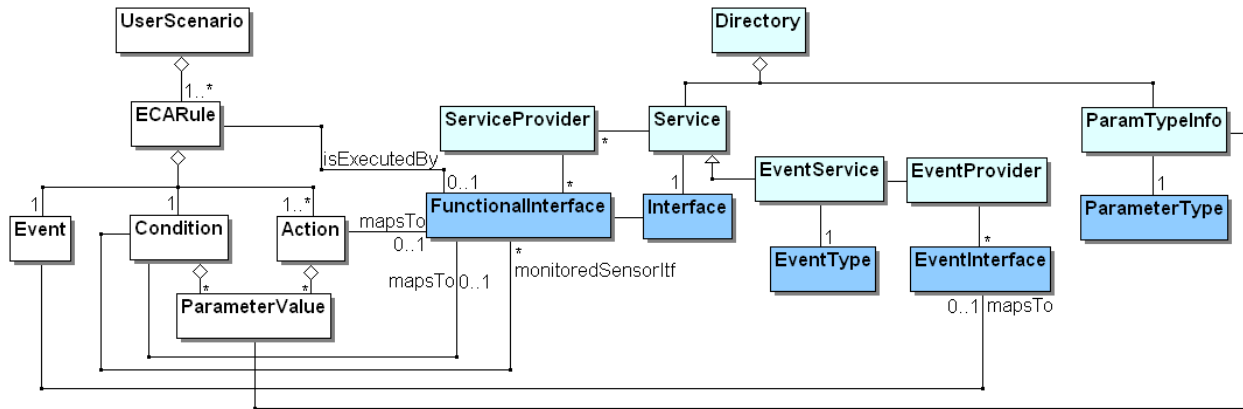


Figure 4. Service directory and user scenarios

can provided by several service providers. Each service provider is bound to concrete component interfaces (often represented as a *lollipop* in component models, as shown on Fig. 8), represented by the *FunctionalInterface* class.

- *information on the 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* in component models, as shown on Fig. 8), represented by the *EventInterface* class.
- *information on the parameter types* that are encountered in operations offered by devices of the environment. 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 rule clause elements refer to the corresponding service advertisements (as shown back up in Fig. 3) and are further mapped to corresponding concrete component interfaces (events to event interfaces, conditions and actions to functional interfaces) when the providers of the needed services have been found/chosen. 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 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 (see Fig. 5). The meta-model proposed here for component assembly (see Fig. 6) inspires from our previous work on self-assembling components [10, 9].

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 requirements 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, *coord-*

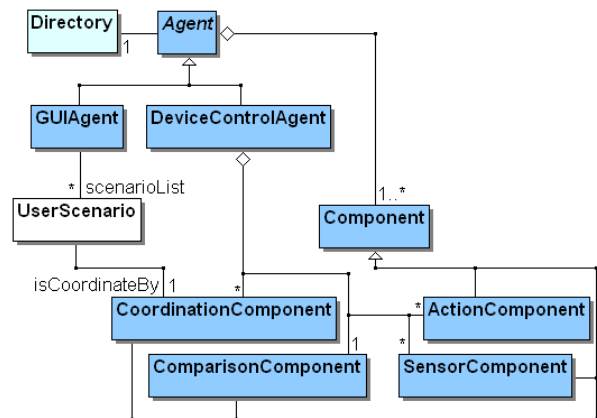


Figure 5. Component-based software agents

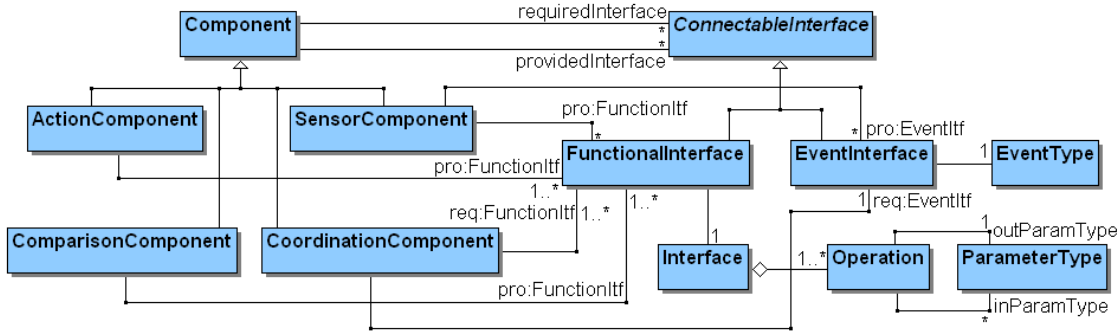


Figure 6. Component typology and external description

dination 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 programming languages such as 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 measured value. For example, an event can be emitted when the temperature of a room changes or when the TV turns on. 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.

5.1 Self-configuration phase

The self-configuration 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 contain 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.

5.2 Self-assembly phase

The self-assembly 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 the its sensor components, 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* is shown on Fig. 7. It contains the information necessary for the DCA to parameterize the coordination component and assemble it to the *Clock*, *Radiator*, *Shutter* and comparator components. The resulting assembly is presented in Fig. 8.

```
rule:
  event: LivingRoom.Clock.TimeChange(time=07:00 PM)
  condition: LivingRoom.Radiator.Temperature()
  action: LivingRoom.Shutter(id=*) .Close() ;
          LivingRoom.Radiator.TurnOn(level=6)
```

Figure 7. Evening scenario rule descriptor

6 State of the art

6.1 Component models

Sensor Beans (SB) [14] and SOFA [19] are two component models close to ours. They both propose a typology of interfaces and oppose to our approach that relies on a component typology. Despite this, there are correspondences. Our action components have interfaces that correspond to *Service* interfaces (SB) and *CSProcCall* interfaces (SOFA) while our sensor components have interfaces that correspond to *Event* and *Producer/Consumer* interfaces (SB) and *EventPassing* and *DataStream* interfaces (SOFA). Our proposal is thus comparable as for the syntactic richness of component interfaces but further adds semantics thanks to the component typology.

6.2 Domotic systems

We classify existing domotic systems into three categories and compare existing systems and the proposal of this paper according to a series of aforementioned criteria (see Table 1). The first category, we call *Pre-defined Scenario Systems* (PSSs), contains centralized systems based on predefined scenarios. In [3, 4, 11], 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. [3, 4] provide a little more general architecture: new components are generated as bridges, to enable communication between various technologies and protocols. In our proposal, the components that can be generated on the fly are not dedicated to satisfying technical purposes but to meeting new user requirements (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-requirements. Moreover, they are not change-resistant as any unforeseen change requires the intervention of an expert user (the system designer). The second category, we call *Service Control Systems* (SCSS), contains systems [2, 12, 15, 22] that 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 (see Sect. 3.1) provided in our system as one of the two means the user has to interact with the domotic system. The user interacts with the system through this GUI to trigger service executions but cannot define complex scenarios. Among them, [22] nonetheless allows to define simple

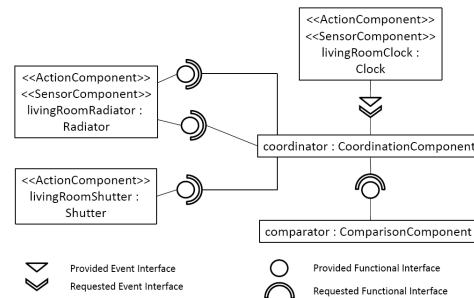


Figure 8. Evening scenario component assembly

scenarios as service sequences. The third category, we call *Scenario Definition Systems* (SDSS), 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, [16] provides users with a GUI allowing them to define and execute their own scenarios. However, scenarios seem restricted to sequences of service calls: they do not propose conditional executions as ECA rules do. Moreover, services can be composed so as the result of a service is used as a parameter of another. Such parameters must then be hard-wired into services: there is no possibility for users to dynamically define 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 requirements. 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 components. The system is under development using the OSGi and UPnP standards¹. 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] J. J. Alferes, F. Banti, and A. Brogi. An Event-Condition-Action logic programming language. In

¹<http://www.osgi.org> and <http://www.upnp.org>.

	System	Decentra- lization	Agents	Compo- nents	Meta- data	User scenarios	Dynamic evolution	Component generation	Auto- matic	Services
PSSs	ANSO [3]	no	no	yes	no	no	yes	yes	yes	yes
	Homega [4]	no	no	yes	yes	no	yes	no	no	yes
	MaDcAr [11]	no	no	yes	no	no	yes	no	yes	no
SCSS	CINEMA [2]	no	no	no	no	no	no	no	no	yes
	Sencha [12]	no	no	yes	no	no	yes	no	yes	yes
	PHS [15]	no	no	yes	no	no	yes	no	yes	yes
	MASML [22]	yes	yes	yes	no	no	yes	yes	yes	yes
SDSS	Wcomp [7]	no	no	yes	no	no	no	yes	yes	no
	HNS [16]	yes	no	no	no	yes	yes	no	yes	yes
	Our prop.	yes	yes	yes	yes	yes	yes	yes	yes	yes

Table 1. System comparison

- 10th *Europ. Conf. on JELIA*, Liverpool, UK, LNAI, 4160:29-42, Sept. 2006. Springer.
- [2] S. Berger, H. Schulzrinne, S. Sidiroglou, and X. Wu. Ubiquitous computing in home networks. *IEEE Communications Magazine*, 41(11):128-135, Nov. 2003.
- [3] A. Bottaro, A. Gerodolle, and P. Lalanda. Pervasive service composition in the home network. In *21st Int'l Conf. on AINA*, Niagara Falls, Canada, 596-603, May 2007. IEEE.
- [4] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, and C. Marin. A dynamic-SOA home control gateway. In *IEEE Int'l Conf. on SCC*, Chicago, USA, 463-470, Sept. 2006.
- [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'l Conf. on WI-IAT*, Hong Kong, 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'l Wkshp on RSP*, Los Alamitos, USA, 119-125, 2006.
- [8] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, and al. The HiPAC project: combining active databases and timing constraints. *ACM SIGMOD Records*, 17(1):51-70, March 1988.
- [9] 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.
- [10] 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.
- [11] G. Grondin, N. Bouraqadi, and L. Vercoouter. MaDcAr: An abstract model for dynamic and automatic (re-)assembling of component-based applications. In *9th Int'l Symposium on CBSE*, Västerås, Sweden, LNCS, 4063:360-367, June 2006. Springer.
- [12] H. Ishikawa, Y. Ogata, K. Adachi, and T. Nakajima. Building smart appliance integration middleware on the OSGi framework. In *7th IEEE Int'l Symposium on ISORC*, Vienna, Austria, 139-146, May 2004.
- [13] 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.
- [14] C. Marin and M. Desertot. Sensor bean: a component platform for sensor-based services. In *3rd Int'l Wkshp on MPAC*, New York, USA, 1-8, 2005. ACM.
- [15] K. Matsuura, T. Hara, A. Watanabe, and T. Nakajima. A new architecture for home computing. In *IEEE Wkshp on WSTFES*, Washington, USA, 2003.
- [16] M. Nakamura, H. Igaki, H. Tamada, and K. ichi Matsumoto. Implementing integrated services of networked home appliances using service oriented architecture. In *2nd Int'l Conf. on SOC*, New York, USA, 269-278, 2004. ACM.
- [17] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing special section. *Communications of the ACM*, 46(10):24-28, Oct. 2003.
- [18] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer. Service-oriented computing: A research roadmap. In *SOC Dagstuhl Seminar Proc. 05462*, IBFI, Dagstuhl, Germany, 2006.
- [19] F. Plásil, D. Bálek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. *Int'l Conf. on CDS*, 43-52, 1998.
- [20] SCA Consortium. Building systems using a service oriented architecture, 2005.
- [21] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM, 2002.
- [22] 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.