Composition Management Interfaces for a Predictable Assembly

Xabier Aretxandieta[1], Goiuria Sagardui[1], and Franck Barbier[2]

[1] Mondragon Goi Eskola Politeknikoa, Informatika Saila, 20500 Mondragon, Spain
{xaretxandieta,gsagardui}@.eps.mondragon.edu,
www.eps.mondragon.edu
[2] PauWare Research Group, UPPA, BP 1155, 64031 Pau CEDEX, France
Franck.Barbier@FranckBarbier.com
http://www.PauWare.com

Abstract. Software system construction based on the reuse of software components has to be done with flexibility enough to control the desired behavior of the resulting assemblies. Applications created by component composition usually depend on a strict method of construction in which COTS or in-house components are only integrated with great difficulty. Actually, reliable assemblies result from being able to observe the inner workings of components and from getting an in-depth understanding of them. The need for fine-grained tailoring and adequate setups is also therefore essential. To enhance the usability, the interoperability and the runtime adaptability of components, composition management interfaces are proposed. They aim at preparing and guiding composition by exposing information about components' inners (states and transitions), which in turn allow for the making of rules that formalize appropriate composition conditions. Finally, state-based expressions for composition are built on a set of primitives discussed in the paper.

1 Introduction

Assembling software components is among others challenges [1] one of the key problems of Component-Based Software Engineering (CBSE). Nowadays, it is difficult to find compatible software components due to the diversity of sources of commercial components. Indeed, many of the COTS components currently on the market have not been developed with varied collaboration potentialities. As well as for in-house components, their compositionality must be anticipated, at design time especially. So, when designing components, one has to equip them with special interfaces and internal mechanisms to support runtime monitoring and mechanisms for adjusting composition [2]. This supposes that, at design time, such adjustment capabilities have been instrumented and transferred to the executable component for a component composition validation like in [3] but almost with different orientation. In this scope, this paper promotes composition primitives which are members of what we call "Composition Management Interface" (CMI). These primitives create a visibility on selected internal mechanisms of components in order to formalize composition in general. This paper

M. Lumpe and W. Vanderperren (Eds.): SC 2007, LNCS 4829, pp. 81–96, 2007.
© Springer-Verlag Berlin Heidelberg 2007

also discusses two different points of view on the use of these primitives. These two points of view are:

- the fact that a designer of a software system uses information gathered by the offered primitives to simulate component compositions in general. In this case, she/he carries out a verification/validation process. For instance, the resulting assembly may function while some QoS attributes may be deficient (e.g., performance);
- the fact that a designer of a software system calls the said primitives in its application, i.e. at runtime, to avoid composition failures. For instance, forcing a component to be in a given state before interacting with another one. Normally, this is an uncommon action since a component aims at, in general, encapsulating such internal features.

So, the necessity of controlling a system of collaborating components requires the assessment of the system's functional behavior and the possibility of managing abnormal situations, such as defects for instance. These often not only result from a single component deficiency but from the global collaboration itself. Since several collaborating components may come from different sources, composition may either fail or be unreliable if designers (COTS providers, in-house component developers) do not organize and offer composition monitoring/control capabilities for their products. In this paper, it is envisaged and proposed a notion of CMI which encompasses the ideas of component configurability, component observability and component controllability to support composition capabilities. This notion is based on the modeling of explicit states of components. Moreover, the consistent call sequences of primitives in CMI correspond to state-based design contracts that are both checked at model time and at runtime. Finally, assembly predictability in terms of an assembly's expected functional behavior and the analysis of unsatisfactory QoS properties (essentially failures) are first-class concerns resulting from the use of CMI. Instead of only having provided interfaces that document or instrument syntactical compositions (callable services [12]), there are described rules which provide a safer composition framework. For example, forcing a component for being into a given state is often a strict constraint which has to be satisfied before the component may begin collaborating with another component. Such a primary operation is typically a basic service of a CMI. As demonstrated in [21], the syntax of interfaces does not offer information enough for making a component execution adapt to unanticipated collaborations.

We show in [4] that COTS components benefit from having tangible states [9] which are accessible in management-based interfaces. From the point of view of component acquisition, this creates a support for evaluation, selection and adoption. From the viewpoint of component use and integration, in this paper, it is explained and illustrated through an example, an appropriate set of state-based composition primitives. In [5], we describe a software library and explain how to use this library in order to organize and implement the inner workings of components based on complex states and complex state dependencies. We

also apply it to Enterprise JavaBeans^{TM}(EJBs), a widespread and well-known technological component framework.

In this paper, the notion of component is more open and not limited to EJBs. A case study in the field of Ambient Intelligence is presented. It is especially show how to construct components that encapsulate sensor and actuator operations. Since such components have states and state-based dependencies, it is explained how to master composition contract disruptions by means of state-based actions offered in CMI.

In section 2, we review the general kinds of composition, in which only the provided and required interfaces of components are used for expressing collaborations. In section 3, we define the notion of a State-Based Contract (SBC) and we state in further details the composition primitives which are grounded on component states. Section 3 also exposes what constitutes CMI and how they may be put into practice with an example. Finally, we conclude in section 4.

2 Composition Types

"Components are for composition" [7]. Our vision of components resides in the understanding of the component paradigm, not as an architectural abstraction related to an ADL but as an implementation as stated in [15]. To that extent, in [13] and in [14], they offer composition languages to create a composition in a correct way. In these two contributions, formal languages are provided to statically verify and validate the correctness of composition.

Two general-purpose means of composition are here considered: vertical and horizontal. Vertical composition (Fig.1a) refers to a whole-part relationship, in which a composite governs all of its compounds. Horizontal composition (Fig.1b) is defined as the cooperation of a set of distributed/non distributed components which do not create a bigger component. In a vertical composition, the assembly is itself a component, while within a horizontal composition the resulting

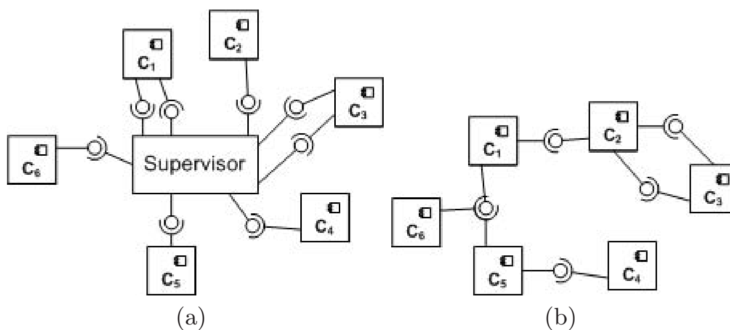

Fig. 1. a) Vertical composition, the assembly named "Supervisor" is the only way for clients to request services. b) Horizontal composition, one or more components may play the role of entry when clients request services.

assembly is simply a functional collaboration or scenario derived from the requirements. This is composition by interaction where the created message circuit corresponds to a composition topology with emergent properties. In vertical compositions, component states are shared between composites and compounds. In horizontal compositions, no coercive assumption can be made since composition conditions are more open than those within the context of vertical composition. Vertical composition indeed imposes in most cases coincident lifetimes of components [20].

Realizing horizontal component composition is either based on wrappers or glue code. This is the notion of exogenous composition [6]. This technique isolates components from their close environment, so that composition is accomplished through intermediate adapters: wrappers, glue code, containers, etc. If such brokers partially or totally hide components from their expected component clients, this creates a mixing of vertical and horizontal composition, as shown in Fig.2. The counterpart of exogenous composition is endogenous composition. This refers to the direct composition of components without any intermediary "broker" or "proxy" components where the composition primitives reside internally in the codification of the component. With this second type of composition, one may notice that technological component models (such as Enterprise JavaBeans for instance) enable the setup of deployment parameters for components in order to clearly show the linking of required interfaces with provided interfaces. This creates a direct and effective composition whose nature is safe[1] but limited in scope.

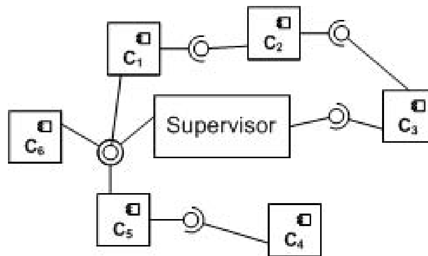

Fig. 2. Hybrid composition mixing the models in Fig.1a and in Fig.1b

In all cases, component composition stumbles over unknown or unpredictable execution contexts if key states of components have not been modeled. For instance, the reliable integration of a component requiring a database connection imposes, as a minimal requirement, the identification of two macroscopic states for a connection object: open and closed. Additional parallel states may also exist for states that embody other stable and well-identified contexts: Busy, Idle, Listening ... So, a connection object can be shared by more than one client, or

[1] The underlying composition model of EJBs predefines the numbers of composition forms and thus creates reliable compositions.

it may be unshared (only one client is served at a given time), or no client at all are served, etc. In this example, forcing the state of this component to be "unshared" or guarantying such a context, permits safer specialized collaborations which are often required if one expects to combine this component with others. Moreover, this decreases the risk of failures. For COTS components especially, they have not been specifically prepared for working together; the possibility of expressing composition pre-conditions as sketched above, is therefore useful.

3 State-Based Contracts

The notion of state-based contract relies on the implementation of the body (i.e. its inside) of a software component by means of a UML 2 state machine diagram [8] (Fig.3). For a system in which several components are assembled, there will be so many UML 2 state machine diagrams as existing individual components in the system.

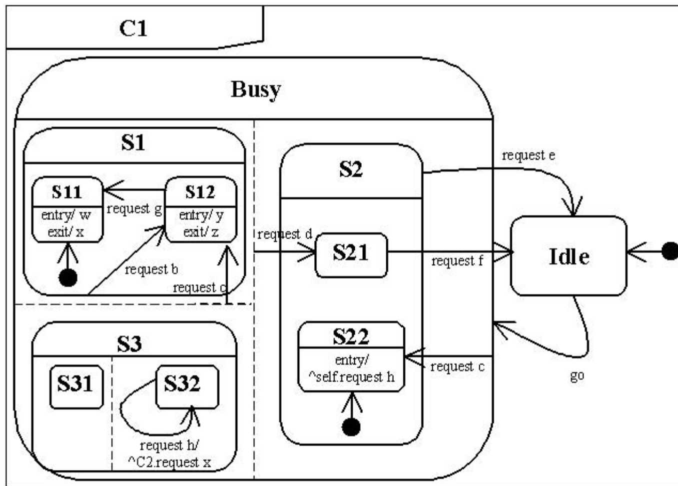

Fig. 3. C1 component specification

This way of working allows the creation of instances of components in two different abstraction levels. The high abstraction level is the level in which the behavior of a software component is modeled in terms of states and transitions. As for the low abstraction level, the component itself is the one that is implemented. The exercised design method leads to having events labeling transitions as services in the provided interfaces of the C1 component (Fig.4).

Composition capability limitation. The expression `^C2.request x` in C1 (Fig.3) corresponds to a composition or a required service of the component. More precisely, it amounts to the sending of an event to another component

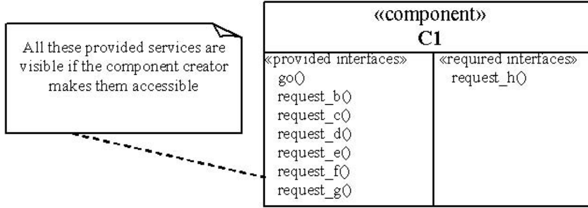

Fig. 4. Component C1 (Fig.3) provided services

instance, in this case C2. This C2 component by definition owns in its state machine a transition labeled by the request x event. Such an approach (i.e. expression ^`C2.request x` inside C1) makes component composition immutable at runtime. It creates a high coupling and contradicts the black-box nature of components especially for COTS components whose future compositions are undefined. This kind of composition is rigid and looses the sense of independent component execution. It also makes difficult component composition prediction activity. In our opinion such composition style should be avoided. Furthermore, strong mechanisms to create a composition and a prediction-enabled technology which aims not only at operating at runtime but also managing it are needed (see section 3.1 and 3.2). In order to create a more flexible and open composition support, the possibility to express state based contracts is offered. For instance, a state-based contract has the following shape:

$$C2.exeTrans(requestx) \tag{1}$$

This expression represents that inside a component C1 it is defined such an expression to send an event (request x for this example) to C2, that has the capability to assume this request.

Our rationale is to externalize (or to avoid at modeling time) the business rules that require composition from the inside of components. Namely, to create a lower coupling, the ^`C2.request x` composition expression must be removed from the behavior specification of C1. Following such a strategy, components gain autonomous capabilities. This component is a very flexible and useful component that the system designer can configure for a component composition. Now that composition it is externalized (out from the inners of the component) the system designer configures the component to assume the defined business rules of the system. The system designer materializes such business rules by expressions (state-based contract expressions) corresponding to a composition. In this case ^`C2.request x` is replaced by `C2.exeTrans(request x)` that it is externally inserted by the designer and it is based on the availability of predefined primitives which make up a fixed composition management interface (Fig.5).

So, more generally, state-based contracts correspond to an expression of a business rule (or more simply a requirement of the global application to be built) using the primitives in the prefabricated Composition Management Interface. So for each defined business rule in the system there will exist state-based expressions

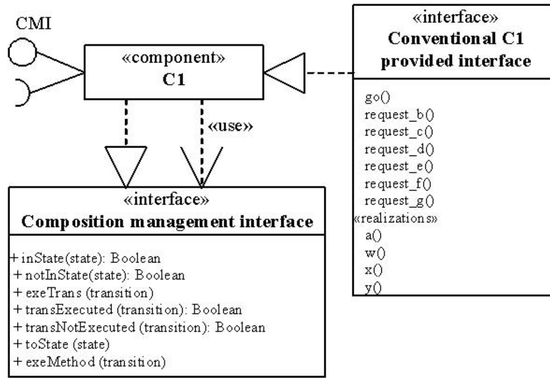

Fig. 5. C1 component realizes CMI interface services, but also uses this interface for the realization of the requested internal actions for a component composition. C1 defines its conventional interface for being used with CMI.

inside the components. And why? Because the components themselves do not cover all the system requirements directly and it is necessary an operation among components to cover those requirements. We are referring to composition.

The CMI just introduces behavior constraints into the component in order to adapt its behavior to the required and defined global system behavior. The component can run as expected. But due to the insertion of state based contract expressions hypothetical malfunction of the component can occur. In this case the responsible for the malfunctioning of the system is the system designer. From a logical viewpoint, a state-based contract can be evaluated at runtime: it holds or leads to a composition failure (see Section 3.2).

For such an intention, components may then be equipped with enhanced composition capabilities as depicted in Fig.5. In this figure, primitives are specified and executed by the CMI that are used to formalize the composition between components in state-base contract formalization.

The composition primitives are defined separately from the component implementation but together with it. The composition primitive's container has deployment descriptors that capture requester intentions for composition that are externally attached to the component. In our case we have created an XML (Fig.6) schema with all the necessary information to establish composition.

Through this XML source, we want to demonstrate that the inside of C1 is not again polluted by composition code (black box nature of the component) but compositions rules exist and occur externally.

The component diagram in Fig.5 imposes for C1 an implementation of the offered CMI. In this case C1 component uses the CMI to execute the business rules by means of a state based contract expression. C2 uses the CMI to execute inside the components the state based expressions from other components.

The way of working is the following: the system designer based on the state machines design of the individual components defines in the XML file the

```xml
<?xml version="1.0" encoding="utf-8" ?>
<!-- Composition Setup file -->
- <composition>
  <!-- All the composed objects are defined here -->
 - <assemblies>
  - <assembly>
      <name />
      <ip />
      <port />
    </assembly>
    ...
  </assemblies>
  <!-- All the events the actions and events that can have -->
  <!-- associated composition primitives are defined here -->
 - <events>
  - <event>
      <name />
      <state />
      <primitive />
      <remoteObject />
      <remoteAction />
      <remoteState />
    </event>
    ...
  </events>
</composition>
```

Fig. 6. Composition specification XML schema

composition primitives that corresponds to a defined business rule. When an event occurs inside a component Cx the XML file is looked up in order to find some defined primitive for such event. In the case that primitives are defined for such event a primitive launcher function executes the related action with the necessary information. Once the primitive launcher executes the composition requirement the requested component performs the requested action (Fig.7).

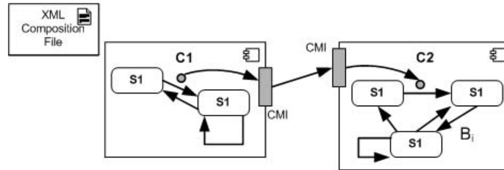

Fig. 7. CMI works for creating monitoring and composition facilities

Finally the prediction activity is defined. [19] Defines prediction as "the ability to effectively predict properties from the system". Our composition mechanism ensures that the design is analyzable, due to the state machine construction and the CMI facilities) and the resulting system will therefore be predictable. In our case we will execute all of the models and component coded using PauWare (statecharts execution engine [16]). Activity into the components will be simulated. And then for each activated business rule the states of all of the models will be studied. The combination of all of the component states will be the information used against the system requirements. If the result of the composition is not satisfactory the CMI can be used to define some failure actions (see section 3.2).

3.1 Composition Primitives

Next the composition primitives are described to define business rules among components. These components will create an assembly through the CMI. For the description of the composition primitives they are divided into two categories, action coordination and action enforced.

Action Coordination. Here are presented a set of composition primitives to establish some functional dependency among several components. These composition primitives enables the designer to, establish such as synchronization, more generally orchestration or a coordination dependency among components. This orchestration it is transferred to conditional actions-dependency used as guards or it is transferred to the action-accomplishment. For such intention the following primitives have been defined:

inState, ***notInState:*** These composition management functions are used to create a composition rule to specify a guard expression in the execution of some given action or a given event inside the component. To create coordinated actions, even synchronized actions between components, the specification of these actions have to be grounded on *inState* and *notInState*. These two primitives act as a Boolean guard (a pre-condition in fact) for the execution of some behavior activity inside the component. This activity can be the execution of an action related to a transition, or the entry/exit actions to be done when arriving in a given state or do activity during the presence in a state.

These primitives can play different roles to define a composition rule. "If"-sense expression or "While"-sense expression. With the first meaning the rule created using the primitive is that if in a given C_x component an e_x event causes an internal transition, this will be completed if another C_y component resides in the specified S_y state. With the second meaning a given component accomplishes some actions while another component resides in an specified state (Fig.10).

exeTrans: This primitive controls the interaction of many components. It sends a signal to another component to execute some desired action. The meaning of the rule is related to an event sending action that a component sends to another one or just a notification for an occurrence. This action, because of the nature of the codification of the component, is the execution of a method. The sending action by the petitioner will be served depending on the state of the receiver component. When possible, the sending transition execution action will be completed and the rule will be fulfilled otherwise the petition of the rule-completion will be ignored.

transExecuted, ***transNotExecuted:*** These primitives are used for detecting if some given transition has been occurred in a component. The utility of this rule resides on the creation of an action dependency among components involved in an assembly. It is used as the *inState* and *notInState* rules but in this case *transExecuted* and *transNotExecuted* is related to a transition-dependency.

Action enforced. This category amounts to a kind of duty-based composition. These are a set of primitives for a designer to try to correct the behavior of an undesired situation. The aim of these primitives is the initialization of the component or the definition of a new stable situation. These primitives are used for:

- The requirement for a component for being in a concrete state
- The requirement for a component to force the execution of a concrete method (in this case a transition).

These two rules are defined to force a component to act in a desired way. The *toState* and the *exeMethod* rules are proposed for this situation. These rules should be under the strict control of the constructor of the component because of the dangerousness of them.

toState: This primitive forces the component to change its current state to the specified one. By the use of this primitive the designer can specify the obligation for a component to be in a concrete and known state in order to ensure the component collaboration. The meaning of this primitive is that "it is mandatory for a component being in a known and concrete state or situation in order to guarantee the safeness and correctness of the system".

In Fig.9 an example is defined that uses the *toState* primitive. In this case the presence detector component uses the alarm component to activate the alarm, if the detected and identified person it is a non trustworthy person.

exeMethod: This primitive almost forces the component to execute a method (i.e. a transition) in any case although theoretically it is not possible to give a response from its current state. In this case this method is related to an execution of a transition that is not programmed. Like the previous mentioned primitive (i.e. *toState*) it is focused on getting the component into shape.

A very important issue in these two primitives is that both of them should be clearly specified by the constructor of the component, to guarantee the safeness of the component and the extensibility of the safeness and the stability of the system. It must also guarantee the controlled means of execution of the rules, to avoid anarchical executions or security faults, where every state and every part of the component is accessible. The use of these primitives can be secured by the use of the invariant state contract.

3.2 Composition Exception Capture

Individual behavior differs from group behavior as has been stated above. In all cases where composition primitives referred to business rules are used, a global behavior is defined. Using the CMI, an exception capture mechanism is used in parallel to govern these global behaviors. In a compound, due to the individual evolution of the composed components, some coordination faults can occur. The component can run correctly, as the vendor or the developer of the component has promised. It can have the correct or expected behavior, but because of the introduced composition primitives an error can be produced. The reason for using

this exception capture is the possible non completion of a composition primitive. In these cases, the non compliance of the business rule should be captured to give the desired response to the abnormal situation. Formally speaking, this is not an error, since it is an uncoordinated action or a bad global state combination result of the composition primitive execution. In this case an unstable status of the system can be produced. Therefore instructions/actions should be proposed by the designer to act in consequence.

This abnormal situation must be captured by the system designer that has introduced the composition primitives. This is the part of the contract that the designer of the system establishes in order to guarantee the desired behavior of the whole system. Similar to [10] if the component does not accomplish with the defined global action the exception capture mechanism is where the composition primitives can be corrected. So for every defined composition rule, a composition exception capture mechanism must be defined to ensure the consistency of the system. This does not guarantee that the anomaly that originated the error is the last executed instruction. The latter is generated due to the occurrence of a coordination fault or cooperation faults. So this can be located where a composition rule has been used. In this case although something not desired has occurred, it is captured and the system is analyzed by trying to predict where the coordination fault has occurred.

3.3 Case Study

In order to put into practice the previous ideas, an ambient intelligence application (domotics system) is proposed and developed in the field of embedded systems. In this domotic system case study built on the top of PauWare [16], the identified collaborating components are:

- presence detector: this component detects the presence of somebody and tracks this presence in the house;
- preferences selector: a preferences manager in which the preferences of the states of the rooms are described in order to control them (sensors and actuators);
- illumination manager: direct actuator on the illumination of a room;
- alarm: this component detects an intrusion in the house and alerts the external security company.

A syntactical composition of all of these business components leads to the UML2 Component Diagram in Fig.8.

As can be seen in Fig.8 no one of the component connectors (provided and required interfaces assembly) have any name. This represents the specification of the composition necessities of the components that later will be created using the CMI. Here only the relations between the components are specified.

For each use case scenario (i.e. business rules) where there are several components, compositions primitives are defined. For example when a person enters the house, she/he is recognized by the Presence Detector (PD) as being a secure or non secure person. Next, preferences for the identified person in the house

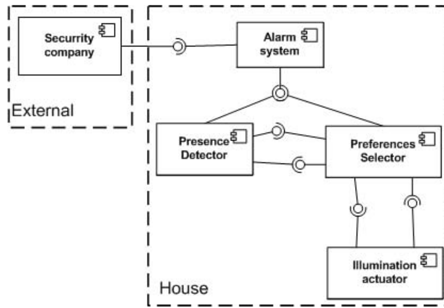

Fig. 8. Component diagram for the domotics system

are loaded and executed. This is the specification of the relationship between the two components in Fig.8. Then this business rule must be formalized in the XML composition specification file to declare explicitly which are the primitives for covering the business rule.

Once the preferences are loaded the illumination manager is the actuator that places the desired illumination level for a room taking the control over the lights and the blinds to graduate the amount of illumination desired.

In the next example (Fig.9) the utility of the CMI is demonstrated for the *exeTtrans* and *toState* primitives.

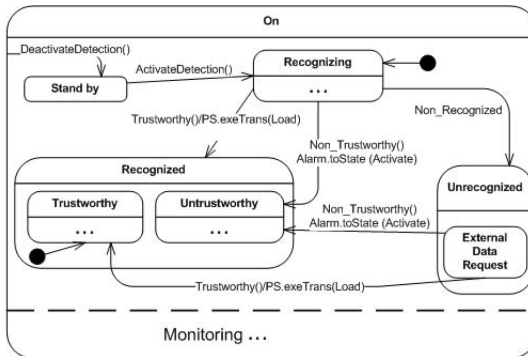

Fig. 9. Use of the *exeTrans* and *toState* primitives for the presence detector (PD) component

In Fig.9 the PD component functionality is explained. This component identifies the person who has entered the house. If the identified person is a trustworthy person (biometric information capture), the PD notifies preferences selector (PS) to load the preferences of that person. The *exeTrans* primitive specified by the signature `PS.exeTrans(Load)` it is used to send a signal to the PS component trying to execute the load method.

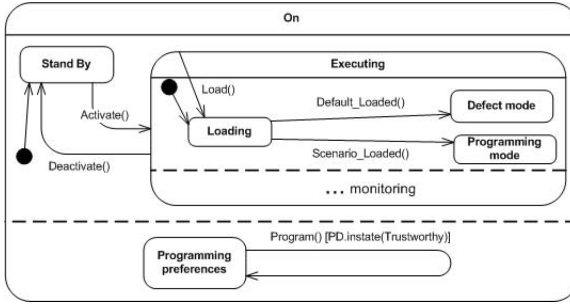

Fig. 10. Use of *inState* for the preferences selector (PS) component

If the detected person is a non trustworthy person, the PD component uses the alarm composed component to activate the sound of the alarm whatever this component is doing (`Alarm.toState(Activate)`).

In Fig.10 example the PS component and the *inState* primitive are shown. In this case the preferences for the actuators, only can be programmed by an identified and trustworthy person. By the use of Set Preferences [`PD.inState (Trustworthy)`] an scenario can be specified and programmed by users if the PD component resides in the trustworthy state (Fig.10).

It has been shown briefly the functioning of the CMI interface with the above exposition. In an ambient intelligence environment many components collaborate together in order to fulfill system requirements. Those components have been reconfigured in order to adapt their functionality to the desired one through the CMI. Their reconfiguration has been done respecting the nature of the components themselves as black box elements. Finally a prediction of the functionality has been done and conclusions can be obtained.

4 Conclusions

Due to the increasing complexity of systems, which results from the deferred necessary component adaptation to environmental constraints, composition has to be carried out through the latter, even for the dynamic (not static like in [17]) fixing of parameters of the involved components for a correct composition. Composition based on wrappers or glue code is nowadays no longer sufficient. SBC provides the system integrator with the necessary tools in order to introduce a component into a system in a controlled way. The CMI provides flexibility enough to create a component composition due to the common interface and in a predictable way because the SBC externalizes the internal visibility of the components.

The SBC's aim is to tackle composition difficulties due to interface incompatibilities and composition predictability. SBC proposes a unique and common interface that allows component composition based on defined composition primitive's actions. These primitives are also used to check properties of the components.

If a composition among components is carried out the SBC has an advantage over the mere component linkage technique. The SBC allows introducing norms of behavior for the components aside from the individual component behavior. This way the composition among components is more flexible. The component is an autonomous element which behavior's is refined to correspond to the desired one. The SBC defines a desired global behavior strategy that each component must comply in order to act together with other components trying to cover system requirements. Like this, components choreography can be laid down for a compound. It is important to notice that such approach does not use the source code of the components for establishing composition. The composition management functions offered by the CMI are used to adapt the component to its close environment.

Two key aspects are related to the CMI. The first one corresponds to the inclusion of the syntactic part of the component creation. However, the CMI must be added to the components construction in order to be composed (interoperability). The second one corresponds to the semantic part. The required information from the COTS supplier or in-house component developer should be based on the information of the components behavior, mainly the state attribute that depicts behavior. Thus, by extension, state machine diagrams define the behavior of the whole component. This information neither reveals the essence for example of the COTS (more problematic than in-house component), nor any crucial information about the component. It only describes the functionality of the component on a high level of abstraction to be used at design time. The description of the component must at least make reference to the previously mentioned information to avoid misinterpretations about the behavior of the component. If this information is not provided, it is very difficult to select an appropriate component for the specific necessities. The construction based on State Machines depicts control over the behavior.

It is important to realize that the inclusion of these composition primitives into the components depicts their nature of required services. The provided "services" of the components must be configured in such a way that those "services" are usable for the rest of the components involved into a composition. Because of that, the meaning of this "service" configuration for possible component collaboration and action coordination among components is understood as components required "services" or action dependency. Obviously the components provided "services" are the ones that are used to shape the behavior interface and its composition primitives.

The CMI must be ready to use the afore mentioned information in order to be used by the system integrator to define a global behavior with predictable consequences. This allows the insertion of composition rules to the individual components to preserve the global behavior. As a consequence the SBC has the advantage of a predictable composition. This is because the provided rules monitor the desired and undesired states in an effective way and they use this information in early steps of the system creation.

This interface also has monitoring capabilities so that the system constructor may analyze the properties of the system (for COTS for example often the documentation associated is not the most adequate). This feature reinforces the prediction capabilities of the CMI for the system's behavior.

From the CBSE viewpoint the proposal of this paper corresponds with the reality of the component concerned from other mature engineering. In this paper the component has been presented as an autonomous element which execution is not related to other components avoiding high component coupling. This way the component operating capacities are higher and can be used in many heterogeneous applications. It also has been presented the component as an executable unit independent [18] from its close environment. Consequently for component composition the system designer must use the CMI to adapt the component to all requirements.

References

1. Crnkovic, I., Larsson, M.: Building Reliable Component-Based Software Systems. Artech House publisher (2002) ISBN1-58053-327-2
2. Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M.: Towards an Engineering Approach to Component Adaptation. In: Reussner, R., Stafford, J.A., Szyperski, C.A. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 193–215. Springer, Heidelberg (2006)
3. Speck, A., Pulvermuller, E., Jerger, M., Franczyk, B.: Component composition validation. Int. J. Appl. Math. Comput. Sci. 12(4), 581–589 (2002)
4. Barbier, F.: Web-Based COTS Component Evaluation, proceedings of The 3rd International Conference on COTS-Based Software Systems. In: Kazman, R., Port, D. (eds.) ICCBSS 2004. LNCS, vol. 2959, pp. 2–4. Springer, Heidelberg (2004)
5. Barbier, F.: An Enhanced Composition Model for Conversational Enterprise JavaBeans. In: Gorton, I., Heineman, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, Springer, Heidelberg (2006)
6. Lau, K.-K., Elizondo, P.V., ZhengWang,: Exogenous Connectors for Software Components. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 90–106. Springer, Heidelberg (2005)
7. Szyperski, C., Gruntz, D., Murer, S.: Component Software - Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, Reading (2002)
8. OMG Unified Modeling Language: Superstructure, version 2.0, Final Adopted specification, ptc/03-08-02 (2007), http://www.omg.org/uml/
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Element of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
10. Meyer, B.: Object-Oriented Software Construction, 1st edn. Prentice-Hall, Englewood Cliffs (1988) ISBN: 0-13-629031-0
11. Wallnau, K.C., PECT: Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC). Technical Report CMY/SEI-, TR-009 ESC-TR-2003-009 (2003)
12. Zhang, S., Goddard, S.: xSADL: An Architecture Description Language to Specify Component-Based Systems. ITCC (2) 443–448 (2005)

13. Sofware Compositon Group (2007), http://www.iam.unibe.ch/~scg/
14. Assmann, U.: Invasive Software Composition. Springer, Heidelberg (2003) ISBN:3-540-44385-1
15. Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: Volume II: Technical Concepts of Component-Based Software Engineering, 2nd ed (CMU/SEI-2000-TR-008, ADA379930). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University (2000)
16. (2007), http://www.PauWare.com
17. Inverardi, P., Wolf, A., Yankelevich, D.: Static checking of system behaviors using derived component assumptions. ACM Transactions on Software Engineering and Methodology 9(3), 239–272 (2000)
18. Nierstrasz, O., Tsichritzis, D.: Object-Oriented Software Composition. Prentice Hall, Englewood Cliffs (1995) ISBN: 0-13-220674-9
19. Hamlet, D.: Defining "Predictable Assembly". In: Gorton, I., Heineman, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 3–540. Springer, Heidelberg (2006)
20. Barbier, F., Aretxandieta, X.: State-based Composition in UML 2, submitted to IJSEKE
21. Sagardui, G., Aretxandieta, X.: Leire Etxeberria Components Behaviour Specification and Validation with Abstract State Machines. In: IBIMA 2005 Conference on Theory and Practice of Software Engineering (TPSE), El Cairo, December 13-15, 2005 (2005)