Designing Dynamic Software Architecture for Home Service Robot Software*

Dongsun Kim[1] and Sooyong Park[2,**]

[1] Department of Computer Science, Sogang University, Shinsu-Dong, Mapo-Gu, Seoul, 121-742, Republic of Korea
darkrsw@sogang.ac.kr

[2] Department of Computer Science and Interdisciplinary Program of Integrated Biotechnology, Sogang University, Shinsu-Dong, Mapo-Gu, Seoul, 121-742, Republic of Korea
sypark@sogang.ac.kr

Abstract. Behavior, situations and environmental changes in embedded software, such as robot software, are hard to expect at software design time. To deal with dynamic behavior, situations and environmental changes at runtime, current software engineering practices are not adequate due to the hardness of software modification. An approach to resolve this problem could be making software really "*soft*" that enables runtime software modification. We developed a practical framework called SHAGE(Self-Healing, Adaptive, and Growing SoftwarE) to implement reconfigurable software in home service robots. SHAGE enables runtime reconfiguration of software architecture when a service robot encounters unexpected situations or new user requirements. This paper focuses on designing reconfigurable software architecture, so called, dynamic software architecture. We also conducted a case study on a home service robot to show applicability of the framework. The results of the study shows practicality and usefulness.

1 Introduction

This research issued from the intelligent service robot for the elderly project in Center for Intelligent Robotics (CIR) at KIST(Korea Institute of Science and Technology). This project was faced with 'how to satisfy and adapt changing requirements more faster at runtime'. Home service robots provides services such as 'delivering a newspaper', 'reading a book', 'making a cup of coffee', and so on. These services have its own quality attributes(e.g. speed, accuracy, safety, and etc). For example, the user gives a command to his/her robot to move with only a goal position. But situations can be diverse; a) when the user holds a party, there may be many visitors. They are unrecorded and moving objects for the robot. In this case, the robot needs to move more carefully, even though

* This research was performed for the Intelligent Robotics Development Program, one of the 21st Century Frontier R&D Programs funded by the Ministry of Commerce, Industry and Energy of Korea.

** To whom all correspondence should be addressed.

E. Sha et al. (Eds.): EUC 2006, LNCS 4096, pp. 437–448, 2006.

it may be getting slow. the robot's software architecture must be reconfigured to provide high safety a quality attribute, for example, a map-builder that uses vision, laser sensors, sonar sensors but takes a lot of time to build a map. b) when the user is home alone, there are only recorded objects. In this case, the robot can move more quickly. To satisfy this situation, the robot's software architecture must be reconfigured to provide high speed as a quality attribute, for example, a component that uses only laser sensors but takes less time to build a map.

In both cases a) and b), the robot must recognize situations of the environment and infer the user's requirements to adapt its software architecture. If there is no cost limitation, the robot can prepare all possible functionalities to satisfy all quality attributes. Unfortunately, robots have cost limitation because they are consumer products. Thus, a robot cannot have all possible software components as well as hardware components. In addition to cost limitation, robot developers cannot predict all possible interrelations between quality attributes and robot software architecture. This temporal limitation leads to new software architecture configurations for each new set of quality attributes after the robot sold.

In this paper, we focuses on dynamic software architecture that enables software architecture reconfiguration. To handle this we proposes slot-based two level software architecture working in SHAGE framework. In section 2, we explain SHAGE framework which is a basis of slot-based two level software architecture which is described in section 3. Also, we show the results of a case study conducted in a robot in section 4. Then, we draw conclusion in section 5.

2 Background

SHAGE[1]. Framework is developed to give a self-managed software capability to robot software in the project. The Framework consists of two parts which are separated by a dashed line as depicted in figure 1. The inner part is installed in each robot and consists of seven modules. The outer part provides repository services for robots to obtain new knowledge which describes 'how to adapt'. The target architecture(beneath the reconfigurator) represents the software architecture of the robot and the target of reconfiguration. This paper addresses how to design this software architecture.

Seven modules in the inner part of the framework are the Monitor, the Architecture Broker, the Component Broker, the Decision Maker, the Learner, the Reconfigurator, and the Internal Repositories. The monitor is responsible for observing the current situation of the environment(this is what observer does) and evaluating the result of adaptation that the framework does(this is what evaluator does). The architecture broker searches candidate architectures based on architecture reconfiguration strategies and composes candidate component compositions for the selected architecture by using concrete component retrieved by the component broker. The component broker finds concrete components which will be arranged into an architecture and retrieves the components from

[1] Formerly it was AlchemistJ as described in [1].

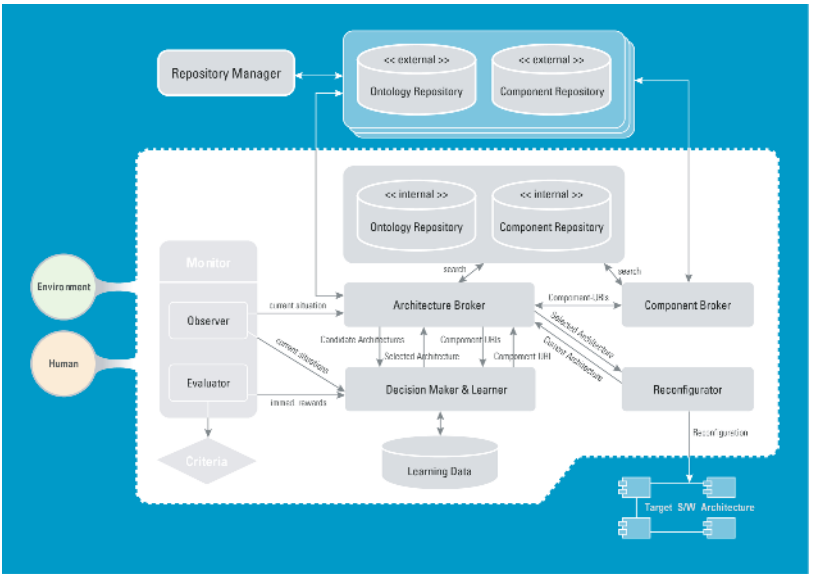

Fig. 1. SHAGE Framework consists of two parts; the inner part has seven modules: the Monitor(not in our scope yet), the Architecture Broker, the Component Broker, the Decision Maker, the Learner, the Reconfigurator, and (internal) Repositories. The outer part consists of repository servers that provide repository services.

repositories. The decision maker determines an best-so-far architecture from a set of candidate architectures that the architecture broker found and an best-so-far component composition from candidate component compositions that the architecture broker composed. The learner accumulates rewards evaluated by the evaluator in the monitor and the decision maker uses these accumulated rewards to choose the best-so-far architecture and the best-so-far component composition. The learning data is a knowledge repository for the learner. The reconfigurator manages and reconfigures the software architecture of the robot based on the best-so-far architecture and the best-so-far component composition which are selected by the decision maker. The internal repositories consists of the ontology repository and the component repository. The ontology repository contains architecture reconfiguration strategies that describes functionalities the robot must have and component ontologies that describes characteristics of a component. The component repository contains components implemented to be used in the robot software architecture.

The outer part of the framework is a set of servers containing external repositories. Each server has an ontology repository and a component repository as the inner part has. Internal repositories in the robot requests new ontologies and components when the robot cannot adapt its behavior to the current situation properly. Also External repositories broadcasts new knowledge to update robots' internal repositories globally. The repository manager is installed in each server and has tools for addition and removal of ontologies and components. From the

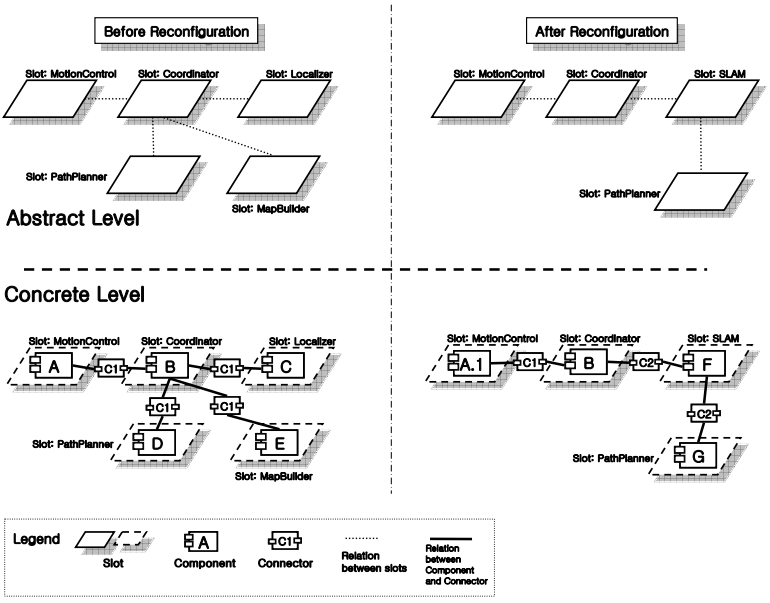

Fig. 2. The Slot-based Two Level Software Architecture Approach consists of the abstract level and the concrete level. In this approach, first, slots are reconfigured in the abstract level and then, components and connectors are placed in the concrete level.

next section, this paper proposes an architectural style to support for designing dynamic software architecture of robot systems.

3 Slot-Based Two Level Software Architecture

3.1 Overview

Dynamic software architecture enables runtime changes of software[2,3]. The slot-based two level software architecture approach provides two level adaptation mechanism; the abstract level and the concrete level(shown in figure 2). At the abstract level, there are only *slots*. A slot represents an abstract component that describes services. A service describes functionality that a slot should provide. A service does not indicates a specific method or a concrete component but describes what messages a slot can be requested and what results a slot returns.

At the concrete level, each slot is filled by one concrete component. A concrete component is an executable code, for example .class files in Java or .so files in C++, implemented by predefined component implementation rules(will be explained from section 3.2 to section 3.4). Every component in the framework must implement common interfaces which can process messages such as 'start', 'stop', and 'suspend' and specific interfaces which realize services on a slots and processes component-specific messages that the component can receive and give. These specific interfaces are described by the component description language

shown in figure 3. The component description language is a XML-based language and describes required interfaces that contains messages that the component can request to other components and provided interfaces that can process messages other components requested.

When the architecture broker requests reconfiguration of the robot software architecture, the reconfigurator re-organizes the architecture based on the selected architecture reconfiguration strategy(will be explained in section 3.4) and places components based on the components the decision maker selected and then selects and places connectors between components based on the component descriptions. In detail, the mismatch indicator(will be explained in section 3.3) in the reconfigurator measures mismatches between components and finds suitable connectors. For example, two components have been deployed in two different machines, the reconfigurator selects a remote connector which enables remote communication such as RPC or RMI. After all components are connected, the reconfigurator sends a 'start' messages to every new component in the architecture and reports that reconfiguration is done to the architecture broker.

From section 3.2 to 3.4, this paper proposes an approach to design adaptable components and connectors to be used in dynamic software architecture and to author architecture reconfiguration strategies for runtime reconfiguration of the architecture.

3.2 Designing Adaptable Software Components

Constructing dynamic software architecture begins with designing adaptable software components. SHAGE Framework offers templates and guidelines for designing and implementing adaptable software components. To design software components in SHAGE Framework, first, a designer must identify services of a component. There are two types of services for each component; provided and required services.

A provided service represents what the software component provides to other software components and a required service represents what the software component needs to execute its functionalities. These services of a component is described by a component description language. SHAGE Framework also provides a language for describing a component as shown in figure 3. Each service is described by 'name', 'type' and 'msg' fields. The 'name' field represents a unique service name in a component to be used in implementation. The 'type' field describes classification of a service to check compatibility with other services. Classification is determined by service ontologies in the ontology repository(shown in figure 1). These ontologies depicts generalization relations between services; from abstract services to specific services, for example, from 'pathplanning' to 'pathplanning.laserbased', to 'pathplanning.laserbased.gradient'. The 'msg' fields represents requests that a service can push to other services in case of a required service or requests that a service can receive from other services in case of a provided service. The 'msg' fields consist of argument fields and a response field. Argument fields describe data which is needed to process a

message. An argument field consists of a name field of the argument and a type field of the argument that describes syntactical and semantical type(e.g. double, float, array of double, robot pose, global map, etc). A response field is provided when the message has return values.

```
<?xml version="1.0" encoding="euc-kr"?>
<Component>
    <name>Navigator.Mapbuilder:LaserbasedMapbuilder</name>
    <description>Laser sensor-based mapbuilder</description>
    <thread value="false"/>
    <language value="CPP"/>
    <deployment value="MainSBC"/>
    <location URI="navigator.mapbuilder.laserbasedmapbuilder.LaserbasedMapbuilder"/>
    <provided-interfaces>
          <service type="Algorithmic.MapBuilding.LaserbasedMapBuilding"
    name="MapBuilder">
                  <msg name='ReadMap'>
                        <reponse>
                              <arg name='Map'
    type='Primitive.double[500][500]'/>
                        </reponse>
                  </msg>
                  <msg name='UpdateMap'>
                        <arg name='dRobotPos' type='Primitive.double[3]'/>
                        <reponse>
                              <arg name='Map'
    type='Primitive.double[500][500]'/>
                        </reponse>
                  </msg>
          </service>
    </provided-interfaces>
    <required-interfaces>
    </required-interfaces>
</Component>
```

Fig. 3. An example of a component description. The component description language is a XML-based language and describes services of components, an implementation language, a deployment location, the location of executable code, and so on.

While the component description language illustrates external part of a component, component implementation templates provides guidelines for inner structure of a component. As depicted in figure 4, inner structure of a component consists of provided/required interfaces, a service manager, and component-specific modules. A required interface encapsulates a message using data from a component-specific module through the service manage of a component and sends the message to a required port of a connector(see section 3.3). A provide interface receives an encapsulated message and elicits data from the message and calls the service manager to execute the functionality that the message indicates. A service manager has two roles; one is to select an appropriate required interface and send data to the required interface when a component-specific module requests a service in other components, and the other is to select an appropriate component-specific module and send data to the component-specific module. A set of component-specific modules is a group of objects to implement application-specific functionalities. With this structure, a component interacts with other components through connectors.

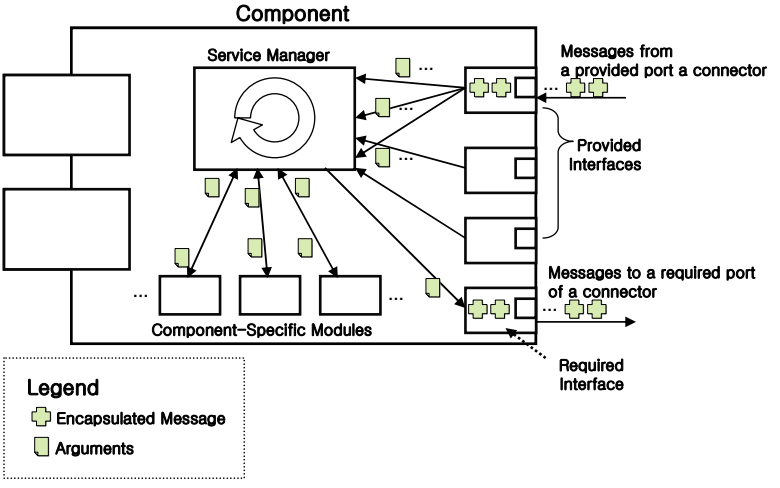

Fig. 4. A component consists of provided/required interfaces, a service manager, and component-specific modules

3.3 Designing Connectors

A connector is responsible for relaying messages between components and transforming data embedded in the messages for components to do their functionalities correctly. The project of CIR(see section 1) consists of over thirty laboratories at universities and companies. During the project we realized that most developers in laboratories had diverse backgrounds such as mechanical engineering, electrical engineering, and so on(few was computer science). This fact caused a lots of different representations for one notion. They used different data formats for maps, a robot pose, etc. For example, to represent a robot pose, one group of developers uses a column-major double array that length is three while the other group uses a row-major integer array that has same length. Another problem was that they would not unify representations because their components are not dedicated to the project and are used to other projects. Moreover, each group claimed their representation was most appropriate for the project and some groups complained that they had no time to redesign all components to follow new unified formats. To handle this situation, SHAGE framework provides guideline for implementing connectors.

In the framework, each connector is implemented based on the template depicted in figure 5. A connector consists of a provided port, a required port, and transformation filters. A required port receives messages from required interface in a component and interprets the messages to prepare transformation. A provided port encapsulates messages and sends the messages to a provided port in a component. Transformation filters are composed by the mismatch indicator in the reconfigurator. The mismatch indicator measures mismatches between two components by comparing descriptions of the components(see figure 3 and section 3.2) when software architecture is reconfigured and new components

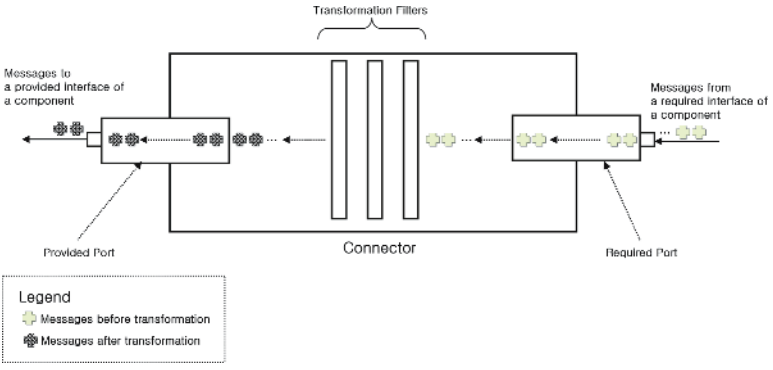

Fig. 5. A connector consists of provided/required ports and transformation filters for resolving mismatches between two components

are introduced in the architecture at runtime. Based on the measured mismatches, the mismatch indicator composes transformation filters by using pre-implemented filters in the component repository(see figure 1). A pre-implemented filter is a predefined conversion rule implemented by developers. We investigated conversion rules of data such as maps, a robot pose, and so on by interviewing developers and classified them. Eventually, mismatches in the software architecture at runtime could be resolved without modifying existing code by using connectors.

3.4 Authoring Architecture Reconfiguration Strategies

To satisfy user requirements(see section 1), software architecture of a robot should be reconfigured at runtime without suspension. Architecture reconfiguration strategies enables runtime reconfiguration of software architecture on SHAGE framework. Strategies are described by a XML-based language shown in figure 6. A strategies illustrates how the current software architecture should be reconfigured at runtime. A strategy only indicates abstract level reconfiguration, i.e. reconfiguration of slots shown in figure 2. Once abstract level architecture is reconfigured, the reconfigurator automatically places components into slots and selects connectors to link components.

Each strategy consists of 'description', 'profile', and 'configuration' fields. The 'description' field is a natural language description of a strategy. The 'profile' field describes minimum constraints that the current software architecture should have before reconfiguration to prevent failure of reconfiguration(e.g. existence of slots to prevent replacement slots). The 'configuration' field describes how slots are reconfigured. In the field, Three command are possible, i.e. 'replace', 'add', and 'remove'. To request slot-replacement or addition , services of slots should be described. These services are corresponding to the services in component descriptions described section 3.2. If a new slot is introduced in software architecture by slot-replacement or addition, the component broker in the framework(see figure 1) searches a set of candidate components that can process indicated

services from the component repository. Then, the decision maker selects the most appropriate component in the set of candidate components based on learning data the learner accumulated. With the selected component, the reconfigurator reconfigures the current software architecture to a new software architecture as described in the strategy.

3.5 Related Work on Dynamic Software Architecture

Hillman's work[4,5] proposed an open framework for dynamic reconfiguration which supports component addition, removal, and replacement. This work focused only on a framework architecture and reconfiguration scripts to support reconfiguration. There was no concerns on designing components and connector and constructing software architectures. Hadas[6,7] framework provides facilities to dynamically change inner structure of a component. Hadas offers a good methodology for designing adaptable components which contains the way to reconfigure methods and data of a component by using metamethods and metainvocation mechanism. But this framework did not concerned architecture-based reconfiguration so that there was no way to design connectors and software architectural styles. Oreizy's work[8,9,2] is a famous research for architecture-based reconfiguration which provides the C2 architectural style as a reconfigurable software architecture style(which contains ways to design components and connectors), xADL as a architectural description language[2], and ArchStudio as a framework to support architecture-based adaptation of software at runtime. But C2 architectural style has restrictions such that a component can have at most two connections, upper and bottom and all connectors should be bus style connectors.

```
<?xml version="1.0" ?>
<reconfiguarationdescription name="http://sembots.icu.ac.kr/reconf#ToVisionbasedLocalization">
  <description>
    Change the current robot architecture into vision based Localizer
  </description>
  <profile>
    <required slotName="http://sembots.icu.ac.kr/service#Localizer"
              action="http://sembots.icu.ac.kr/action#Replace"/>
    <required slotName="http://sembots.icu.ac.kr/service#MapBuilder"
              action="http://sembots.icu.ac.kr/action#Remove"/>
  </profile>
  <configuration>
    <script>
      <Replace slotName="http://sembots.icu.ac.kr/service#Localizer">
        <services>
          <service name="http://sembots.icu.ac.kr/service#VisionbasedLocalization"/>
          <service name='http://sembots.icu.ac.kr/service#VisionbasedMapBuilding'/>
        </services>
      </Replace>
      <Remove slotName="http://sembots.icu.ac.kr/service#MapBuilder"/>
    </script>
  </configuration>
</reconfiguarationdescription>
```

Fig. 6. Architecture reconfiguration strategies describes abstract level reconfiguration. A strategy describes replacement, addition, and removal of slots.

[2] xADL describes a snapshot of software architecture, not a architecture reconfiguration language.

4 Experiment

The experiment was designed as follows: 1. initially the user of the robot needs 'more faster navigation', so the robot is configured mainly to use faster sensors(say, laser sensors). 2. while moving, the robot is stuck by a table because the bottom of the table is empty but the current sensors can detect only knee height objects. Then, the user asks the robot to move 'more carefully'. 3. The robot tries to reconfigure its software architecture to detect objects that the current sensors cannot detect.

We implemented a few components and configured the initial software architecture of the robot for navigation as shown in figure 7.(a). Each component can be executed independently. 'MotionControl' component controls wheels and 'Localizer' measures the current position of the robot based on encoder data which means how many degrees the wheels rotated. 'MapBuilder' makes a map around the robot based on sensor data from laser sensors. 'PathPlanner' plans a path from the current position to a goal position based on data from 'Localizer' and 'MapBuilder'. 'Coordinator' gets the goal position from the user of the robot and relays data between components. Based on these components the robot can moves without collisions except tables.

We designed a room as an experimental environment. we placed two tables; one was covered by a tablecloth and the other was not. The robot was placed at the same line on which the two tables were placed. In other words, "robot - table with a cloth - table without a cloth" on the same line in sequence. After configuring the initial architecture of the robot, The user put a goal position to 'Coordinator' and requested 'more faster maneuver'. The goal position was between two tables and the robot verified that the current architecture was suitable for the user's requirement. The robot decided the initial architecture is enough because laser sensors were very fast and precise. Then, the robot started to move and its laser sensors could detect a tablecloth, so the robot could move without collisions. This was not an abnormal situation, so the robot did not need adaptation. Then, we put the other goal position over the second table. In this case, the robot could not detect the table and rushed to the table. The user requested the robot to stop and to realize the situation and to adapt it.

When the robot was requested to adapt its behavior, the monitor in the framework detected the current situation. This situation was passed to the architecture broker. The architecture broker began the adaptation process. The situation was the current architecture could not detect all object in the room so the robot needed to find other functionalities to detect some other objects which could not be observable with the current architecture. The framework tried to apply various architectural configurations. First, the framework selected a strategy that adds a slot which have a localization service and a mapping service(figure 7.(b)). After abstract level reconfiguration, the reconfigurator successfully placed the SLAM(Simultaneous Localization And Mapping) component and appropriate connectors. A connector named 'D' represents two components around the connector are deployed in the same SBC while a connector named 'R' represents they are deployed in two different SBCs. In this case the SLAM

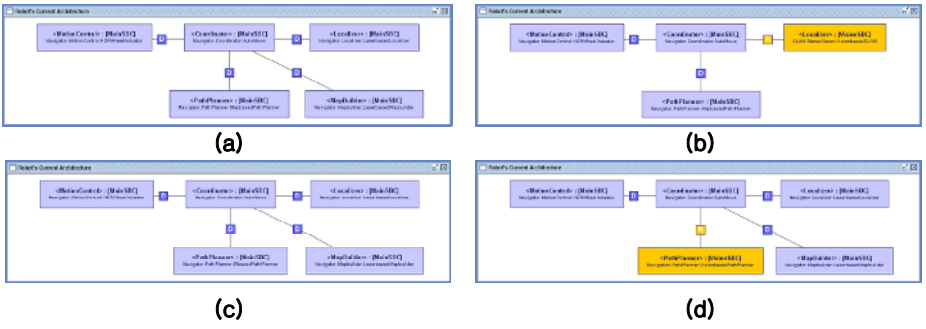

Fig. 7. Various software architectures of the robot during reconfiguration

component was deployed in 'Vision SBC', so that the connector named 'R' was placed. In detail, the SLAM component used a different format to represent a robot pose as explained in section 3.3, the mismatch indicator selected a transformation filter that convert two different format for a robot pose in 'Coordinator' and the SLAM component. The reconfigured architecture worked, but it did not satisfy user's requirements because the SLAM component also could not detect the table. In this manner the framework also tried to apply another strategy shown in figure 7.(c). But the framework realized(and learned) these was not able to solve the situation, and finally selected a strategy including a slot which had a vision-based path planning service. Then, the reconfigurator placed a path planner component that implements a vision-based path planning service(figure 7.(d)). During (a)-(d) the user did not interrupt and put more inputs.

This experiment shows the framework enables the robot to reconfigure robot software architecture successfully by applying the dynamic software architecture approach this paper proposed.

5 Conclusions

We proposed the slot-based two level software architecture as a dynamic software architecture approach for embedded systems especially in robot domain. This approach offers a method to design adaptable components and connectors and to author architecture reconfiguration strategy.

In order to verify operations of the framework, we have implemented an instance of self-adaptive robot software with 'infortainment robot' and examined the ability of adaptation of the robot from the experiment. The robot has adapted its behavior to user's feedback and the current situation.

As mentioned in section 3.3 the project consists of lots of laboratories and they have made a number of software components. These components are made based on different assumptions and platforms. Also these have different interfaces, data types, granularity, and scopes. These differences may cause lots of mismatches when the framework reconfigures the robot's software architecture. In order to

improve the mismatch indicator, we are classifying types of mismatches in robot domain and connector technologies for solving mismatches at run-time.

References

1. Kim, D., Park, S.: Alchemistj: A framework for self-adaptive software. In: The 2005 IFIP International Conference on Embedded And Ubiquitous Computing (EUC'2005), LNCS3824. (2005) 98–109
2. Oreizy, P., Taylor, R.N.: On the role of software architectures in runtime system reconfiguration. In: IEE Proceedings - Software Engineering. (1998)
3. Allen, R.J., Douence, R., Garlan, D.: Specifying dynamism in software architectures. In: Proceedings of the Workshop on Foundations of Component-Based Software Engineering. (1997)
4. Hillman, J., Warren, I.: Meta-adaptation in autonomic systems. In: 10th IEEE International Workshop on Future Trends of Distributed Computing Systems. (2004)
5. Hillman, J., Warren, I.: An open framework for dynamic reconfiguration. In: 26th International Conference on Software Engineering. (2004)
6. Ben-Shaul, I., Cohen, A., Holder, O., Lavva, B.: Hadas: A network-centric framework for interoperability programming. In: Proceedings of the Second IFCIS International Conference on Cooperative Information Systems. (1997)
7. Ben-Shaul, I., Holder, O., Lavva, B.: Dynamic adaptation and deployment of distributed components in hadas. IEEE Transaction on Software Engineering **27(9)** (2001) 769–787
8. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proceedings of the 20th international conference on Software engineering. (1998)
9. Oreizy, P.: Issues in the runtime modification of software architectures. Technical report, Department of Information and Computer Science, University of California, Irvine (1996)