Abstract

Service-Oriented Computing (SOC) is a generic computing paradigm that can be applied to expressing computing of all kinds of applications, just like the object-oriented and imperative paradigms. SOC is also known as SOA (Service-Oriented Architecture) emphasizes the software architecture of using loosely coupled services to compose applications, where services are published by service brokers in their service registries and repositories and are available for remote access on internet. So far, SOC research and applications have been largely limited to software development in electronic and Web based applications, because of its successful Web services standards and implementations, such as WSDL, SOAP, XML.

Service-oriented robotics software development extends SOC from its traditional fields into a new domain, which was considered not feasible a couple of years ago, because of the efficiency issues in terms of computing and communication. This report presents the concepts, principles, and methods in SOC and SOC-based robotics software development, applies these concepts, principles, and methods in the design of a distributed robotics application.

Keywords: Service-oriented computing, distributed computing, event-driven computing, distributed robotics computing.

1. Introduction

The Service-Oriented Computing (SOC) and Service-Oriented Architecture (SOA) as a paradigm refer to the set of concepts, principles, and methods that represent computing in three parallel processes: service development, service publication, and application composition using services that have been published. The main SOC implementation

---

1 The project is sponsored by Intel's Robotics Challenge Initiative, 2007.
today is Web services, which are supported by a set of Web-based technologies [2][6][16], including

- Standard data representation in XML (eXtensible Markup Language), as well as language and protocol definition in XML;
- Standard interface definition in WSDL (Web Services Description Language);
- Standard binding/communication protocol, such as SOAP (Simple Object Access Protocol) and MIME (Multipurpose Internet Mail Extension);
- Ontology definition languages such as RDF (Resource Description Framework) and OWL (Web Ontology Language);
- Service registry and repository mechanisms, such as UDDI (Universal Description Discovery and Integration) and ebXML (Electronic Business using eXtensible Markup Language);
- Application composition languages such BPEL (Business Process Execution Language) and WS-CDL (Web Services Choreography Description Language).

As Object-Oriented Computing (OOC) is based on imperative computing, SOC is based on OOC. In fact, current SOC development requires that each service is defined as a class and the instance of a service is a remote object connected through a proxy.

As shown in Figure 1, the development processes in OOC and in SOC are very different. Typically, an OOC application is developed by the same team in the same language, while an SOC application is developed by using pre-developed services developed by independent service providers. To find required services, the application builder looks up the service directories and repositories. If a service cannot be found, the application can publish the requirement or develop the service in house. Service providers can develop services based on their own requirement analysis or look up the requirement published in the directories.

All major computer companies, including BEA, Google, HP, IBM, Intel, Microsoft, Oracle, SAP, Sun Microsystems, etc [7][5][8][9][11], as well as the government agencies, such as U.S. Department of Education [20] and U.S. Department of Defense [15], have adopted and supported the new computing paradigm, its standard technologies, and features. It results in booming implementation and deployment of major software projects in SOC. McKendrick listed ten companies where major SOC projects were deployed in 2006 [10].

SOC research and applications have been largely limited to the electronic and Web based applications. There are few studies on SOC-based robotics research and applications. Chen applied SOC concepts to develop recomposable embedded systems and robotics applications and implemented a prototype of the system in Fall 2005 [3]. Microsoft released the SOC-based Robotics Studio and VPL (Visual Programming Language) in June 2006 [4], which marks a milestone in SOC and in Robotics. Many robot manufacturers have moved their programming platform to VPL, including Coroware, iRobot, Kuka, LEGO NXT Mindstorm, Parallax, Robosoft, Robotics Connection, WhiteboxRobotics, etc [12].
Service-oriented robotics computing is important not only in application software development but also in education. SOC-based robotics programming is easier than traditional robotics programming and is fun. Sponsored by the U.S. Department of Education, a SOC-based robotics-computing curriculum is being developed for high schools in the United States. Three pilot runs of the course have been taught to high school students and teachers in Spring, Summer, and Fall 2007 [20][21]. More detail about the curriculum can be found at the link: http://asusrl.eas.asu.edu/highschool/soc/

The rest of the report is organized as follows. Section 2 introduces the concepts, principles, and research issues in SOC-based robotics software development. Section 3 presents a case study in Robotics Studio and VPL that illustrates the design of event-driven and service-oriented concepts and principles. Section 4 presents the design of an event-driven and service-oriented robotics application based on off-the-shelf computer components. Section 5 concludes the report.

2. Development of Service-Oriented Robotics Applications

It has been perceived that SOC applications are less efficient than OOC applications because of the additional layer of standard interface, which makes it possible for language- and platform-independent communication and remote invocation implemented in Web services. Embedded systems and robotics applications often need real-time performance, and thus, SOC was considered not a suitable paradigm for developing embedded systems and robotics applications.

The facts are, SOC does not have to be implemented over the Web, and thus, the standard interface could be simpler and faster. The remote services could be migrated
into a local machine to reduce the communication cost [7]. On the other hand, the benefits of applying SOC in embedded systems and robotics applications are significant, particularly, for the following reasons [3]:

- Embedded systems / Robots have limited capacity to carry programs that handle all possible situations;
- Unforeseeable environmental situations can occur;
- Faults can occur and on-site repair is often not available;
- Users want to modify the system (requirements) without stopping the system;
- An SOC application can be made independent of devices that the application communicates with, and thus, the same application can be applied to drive different drives.

Figure 2 shows an example of SOC-based robotics application development. The application consists of a Remote Collaboration Center (RCC) and an onboard computer that is attached to the robot/vehicle. Instead of developing components specifically for a given application, the requirements of the components are published in a service broker. The service providers develop the services and publish them in the broker. Then the application builder looks up the available services and migrates the services into the application [7]. Migration can save the time for the remote invocations. Depending on timing and flexibility requirements, the services can run on the onboard computer or on the RCC. Services running on RCC can be replaced without stopping the robot. The robot’s mission can be modified if the services on the RCC is modified or replaced. This is how recomposable computing is achieved in SOC.

![Diagram of SOC-based robotics application development](image)

Figure 2. SOC-based robotics application development

For the same service requirement and service interface, different implementations for different devices can be developed, as show in Figure 3. Thus, the same application can be used to different devices. Assume the application is to traversing a map or a maze, the same application can be used to drive a different vehicle, as long as the services are rebound to the services for the new vehicle. This concept will be elaborated in section 4.
3. Case Study: Service-Oriented Robotics Application

Academic research in SOC-based robotics development has not been well received because of the concerns mentioned in section 2. However, the situation turns around when Microsoft released the SOC-based Robotics Studio in June 2006. Many robotics companies have moved their software development platforms to Robotics Studio [12]. Figure 4 shows the Microsoft development environment consisting of Robotics Studio, its composition language VPL, the .Net framework, and the .Net programming languages.

Robotics Studio provides a runtime environment for creating, hosting, managing, and connecting services within a node of an application and across the network. The result is a distributed application model, where nodes are peers rather than “clients and servers”. The Robotics Studio runtime consists of two parts. The first part is the Concurrency and Coordination Runtime (CCR), which enables coordination of messages without manual threading, object locks, semaphores, etc. The CCR applies the even-driven model for message passing and provides an execution context for services, including a set of high-level primitives for synchronizing messages.

The second part is the Decentralized System Services (DSS), which provides a service-hosting environment and a set of basic services, facilitating tasks such as debugging, logging, monitoring, security, discovery, and data persistence.
VPL offers a set of basic programming activities, such as variable declaration, initialization, assignment, if-then-else, switch, composite activities (services), and the links between the activities. A loop is constructed by a feedback link and a merge construct. The join construct is used to synchronize multiple activities and combine multiple data items.

Figure 5 shows the VPL code that allows a robot to traverse a maze and find its way out of the maze. The main diagram (code) of this application consists of three types of basic activities: Data, Variable, and If. It also contains three services:

- **SimulatedSonarSensor**: A built-in service that can detect and measure the distance to any obstacle placed in the simulation environment.
- **GenericDifferentialDrive**: This service provides basic driving methods of forward, backward, turn left, turn right, and stop. This service can be bound to a concrete robot, such as an iRobot and LEGO NXT. This experiment, we have bound the service to the “Simulated Robot”.
- **Drive**: This is a user-defined service. It consists of four methods (actions), Stop, Left, Forward, and Right. The methods use the GenericDifferentialDrive and the Timer services to accomplish their driving tasks.

As can be seen in the main diagram in Figure 5, the application is designed in data-driven and service-oriented architecture. First, the Variable status is initialized to “forward”, and the GenericDifferentialDrive is initialized to drive forward at the power level 0.3, resulting in the robot simply drive forward. There are no control flows (links) from them to the other components.

Concurrently, the SimulatedSonarSensor measures the distances repeatedly. The measurement (distance value) is checked in an If-activity with three composite conditions (distance value and status variable’s value), resulting in different actions: Stop, Forward, and Right turn. The status variable is updated in the meantime. Left turn is done after the Stop is finished.

The main diagram in Figure 5 implements a heuristic algorithm for traversing the maze in Algorithm 1.

**Algorithm 1**

1. Initialize the robot to move forward;
2. Sonar sensor reads distance to obstacle repeatedly;
3. If the distance is less than 200 mm, stop for 1 second and then issue an event StopFinished;
4. After the emission of StopFinished, turn 90 degree to “Left” first.
5. If the distance is greater than 1000 mm after the Left turn, move forward.
6. If the distance is less than 1000 mm after the Left turn, turn 180 degree right and then move forward.

There is no loop in the diagram code. The loop is created by the repeated sensor’s readings of the distances. The occurrence of certain condition (for example, the distance is less than 200 mm) will trigger an event, which will in turn trigger a series of actions.
The code for the Right turn method in the Drive service is shown in Figure 6. In the first (left-most) SetDrivePower link, the LeftWheelPower and the RightWheelPower are initialized to 0.3 and -0.3, respectively, as shown in the Properties field in the diagram, resulting in the robot turning towards right. The action of turning right in 90 degree is jointly affected by timing, the power level, and the resistance between the tire and the floor. In the experiment, for the given simulation environment, the power level of (LeftWheelPower = 0.3, RightWheelPower = -0.3) and the timing of 0.642 seconds will make a 90 degree right turn. When the timer runs off in 0.642 seconds, the GenericDifferentialDrive’s drive power to both wheels will be reset to 0, resulting in the robot to stop. After 1.3 seconds, an event notifying “RightFinished” will be emitted, which will trigger another action, as shown in the main diagram in Figure 5.
The code for the Stop method in the Drive service is shown in Figure 7. In this method, the GenericDifferentialDrive’s drive power to both wheels will be reset to 0. The timer will be set to one second after the stop, before the method emits the event "StopFinished".

The code for the "Left" turn is similar to that of the "Right" turn, while the forward method simply call the GenericDifferentialDrive’s forward method.
For the maze given in Figure 8, this simplified algorithm can help the robot finding its way out of the maze. The execution sequence is:

1. Forward –
2. Turn Left 90 –
3. Forward –
4. Turn Left 90 –
5. Forward –
6. Turn Left 90 –
7. Turn Right 180 –
8. Forward –
9. Turn Left 90 –
10. Turn Right 180 –
11. Forward –
12. Turn Left 90 –
13. Forward –
14. Turn Left 90 –
15. Forward

The complete execution process of the code is captured in a video and is given at the link: http://asusrl.eas.asu.edu/Share/services/robotics/MazeTraverse1.wmv

We also implemented a more complex algorithm (Algorithm 2) that compares the distances to the walls on the left and one the right when the robot is approaching an obstacle. Based on the comparison, the robot turns to the side with more space.

Figure 8. A screenshot of the simulated execution of a robot traversing a maze
Algorithm 2

1. Initialize the robot to move forward;
2. Sonar sensor reads distance to obstacle repeatedly;
3. If the distance is less than 200 mm, stop for 1 second and then issue an event StopFinished;
4. After the emission of StopFinished, turn 90 degree to “Right”; Store distance in DR;
5. Turn 180 degree “Right”; Store the distance in DL;
6. If DL < DR
7. Then Turn 180 degree.
8. Move forward.

For the same maze, algorithm 2 can help the robot finding its way out in the following execution sequence is:

1. Forward –
2. Turn Right 90 –
3. Turn Right 180 –
4. Forward –
5. Turn Right 90 –
6. Turn Right 180 –
7. Forward –
8. Turn Right 90 –
9. Turn Right 180 –
10. Turn Right 180 –
11. Forward –
12. Turn Right 90 –
13. Turn Right 180 –
14. Turn Right 180 –
15. Forward –
16. Turn Right 90 –
17. Turn Right 180 –
18. Forward –
19. Turn Right 90 –
20. Turn Right 180 –
21. Forward

The code as well as the execution of Algorithm 2 are longer, but can better scope different mazes. The video of the execution of algorithm 2 can be viewed at the link: http://asusrl.eas.asu.edu/Share/services/robotics/Maze Traverse2.wmv

Both algorithms 1 and 2 are heuristic which can find help the robot finds its way out of certain mazes, but for all possible mazes, which may need much more complex AI, as discussed in [13].

4. Distributed Robotics Application Development

SOC software is distributed in the nature, with the services residing in different computers. This section presents the requirement and design of an event-drive, distributed, and service-oriented architecture of a security robot for unmanned building patrolling. The design is based on off-the-shelf components without any customization.

System architecture

- The system consists of a Remote Collaboration Center (RCC) and a number of robots with onboard computers. RCC has a human user interface, which can
turn the system into a manually controlled mode or an autonomous mode. The discussions of this report will focus on the autonomous mode.

- Each robot is equipped with multiple sensors, including camera, keypad, sonar sensors, and touch sensors; and actuators, including motors, servos, and alarm;
- Each robot offers other robots a service that reports its status.
- Each robot can access other robot’s services and make an independent decision and action based on the collected information. To reduce the complexity of onboard (robot) software, the robots do not communicate with each other directly. They acquire other robots' status via RCC.

**Robot design**

- A robot consists of an onboard computer and a set of sensors and actuators, including a sonar sensor, a touch sensor, keypad, a video camera with a horizontal and a vertical control servo, and two independent servos that control the wheels;
- Sensors and actuators are wrapped as services.
- Each service includes an event emitter, an event subscription vector, and an event listener. The event handlers are associated with the orchestration process.
- An orchestration process reads local and remote sensory values, handles events, and make decision based on the values and application requirements.
- A number of utility services are available onboard, such as mathematic functions, routing, and timing.

**RCC design**

- In the autonomous mode, RCC is a peer to the robots in design, and it has the same architecture as a robot: It consists of a set services and an orchestration process.
- RCC’s services are different. RCC is normally running a more powerful computer, and the time consuming utility services will be executed on RCC. Furthermore, RCC has a user interface and does not directly connect to sensors and actuators.
- In the manual mode, the RCC will be the master, while the robots will be slaves.

Figure 9 shows the architecture of the distributed robotics system and the major components in RCC and the robots.

The Events & Handlers are fully distributed into each subsystem and into different services in each subsystem, as shown in Figure 10. A component that is interested in the event of a service, e.g., a touch sensor, can subscribe to the event. The service maintains a vector of subscribers T_1, T_2, …, T_m (for the touch sensor). Once the sensor is touched, an event will be emitted, which will in turn invoke the handlers of T_1, T_2, …, T_m. The sonar sensor can be set to one of the two modes. In the periodic mode, the sensor sends the reading value of the distance to the obstacle it sees in certain interval. In this case, the event notifications may not be necessary. In the other mode, an event will be emitted if the distance reading changes from its previous value. Both touch sensor and sonar sensor are one-way services. The servomotors are however two-way services. In one way, the motor takes instructions through the motor control to move the robot accordingly. In the other way, the motor sends back the position data after each move, so that the robot’s position can be computed based on the information.
Figure 9. RCC and onboard computers form a distributed system

Figure 10. Part of services onboard a robot
All the event handlers are directly associated with the orchestration process, so that the
process has all the information to make the decision on actions that need to be taken.
Furthermore, the orchestration process can also subscribe to the sensory and motor
services of other robots to make collaborative decisions. For example, if one robot detects
an intruder and the intruder is moving away from the detecting robot. The other robots
can move towards the intruder based on the sensory and position information from all
participating robots.

The orchestration process is also running on the RCC. It is called collaboration process
in order to differentiate it with the process on robot and to emphasize its role in
coordinating the actions among the robot. RCC has the same software architecture and in
principle is a peer of the robots, instead of the commander of the robots. The RCC will
become the commander of the robots only in the manual mode. Since RCC is running on
a more powerful computer, it makes more sense to allocate the more time-consuming
tasks to the RCC and let the robots to communicate with the RCC only. The
communications between the RCC and robots are also implemented through event
emitters and event-handlers, as shown in Figure 11.

The design is made for the Robotics Challenge project sponsored by Intel Arizona. The
robot must perform the functions of security guards, and the system must be built using
commercial-off-the-shelf (COTS) components. More details about the requirement,
design, and implementation of the security robot can be found at:
http://asusrl.eas.asu.edu/srlab/Research/RoboticsChallenge.html

5. Implementation and Experimentation

This section describes the implementation of the project and the experience in the
experimentation process.
5.1 Application Design and Implementation

To demonstrate the strength of SOC-based robotics development, the architecture and design presented in the previous section are being implemented in three parallel projects. Although based on the same SOC architecture and service layer, each project composes a different application. In the first project, the entire system is developed in a period of six months, while the next two projects will share the service layer and build different applications in a period of two months.

All three projects use commercial-off-the-shelf components. The core of the robot is the Intel's embedded single-board computer system 986LCD-M/mITX with T7400 core 2 duo processors. The other components are serial port, USB, or mini PCI devices that plug and play on the motherboard. The communication between the robot and the RCC is through 802.11/g wireless link.

The first project implemented service layer with all the sensors, actuators, functions, and services shown in Figure 10. The operating system is Windows XP and the programming language is Java. The application implemented is a security application for floor patrolling with two working modes: manual patrolling and autonomous patrolling. In the manual patrolling mode, an operator controls the robot through the RCC to move around in a maze-like building floor. Although the operator is in control of the robot, the sensors on the robot will warn the operator if the robot is about to touch an object. The on-board camera sends video stream to the screen of RCC for the operator's monitoring the floor for detecting any possible security issues.

The design focus of the security application is in its autonomous mode, which can be used to monitor the floor activities off the work hours. Following functions are implemented:

- Floor plan detection: When the robot is placed in a new environment, or when the environment is modified, the robot will move around to detect the floor layout and draw the map of the floor. A sonar sensor, multiple touch sensors, as well as the motor position feedbacks are used for floor plan detection.
- A compass sensor is being added to the robot to improve the floor plan detection and patrolling process.
- If the floor map is available, the robot will find an optimal path and then patrol the floor autonomously following the path. Video stream will be sent to RCC screen and will be recorded on RCC hard drive. Sensors and motor position feedbacks are also used in patrolling moves.
- Object detection: If an object that is not in the existing floor plan is detected, video of the object and warning signal will be sent to the RCC.
- Movement detection and chasing: If movement of a person is detected, the robot camera will detect the face of the person, lock to the face, and move following the person's movement. The person will be requested to enter a pass code in a given amount of time. Failing to enter the correct pass code will resulting in an alert to the RCC.

The second and third projects are used to validate the reusability of the framework and particularly, the service layer developed in project 1. Based on the framework and the service layer, each project will develop a different application using different sensors, including
- Using compass sensor to improve the floor plan detection and patrolling process;
- Using a motion sensor to complement the camera-based motion detection;
- Using an IR sensor to complement the sonar sensor;
- Using a siren to generate alerts when the pass code is requested and the person failed to enter the correct code in the given amount of time.

In the next phase, the system will be implemented on different operating systems and in different programming languages. Since the system is programmed in SOC paradigm, the only languages fully supporting SOC are Java and C#. In the phase 2 of our experiment, the first robot will integrated all the functions developed in phase 1 on all three robots, while the robots 2 and 3 will move to a different operating system and in a different programming language. Robot 2 will be using C# on XP, while robot 3 will be using Java and Linux.

5.2 Developing a Robot from Scratch and without a Limitation

We have worked on different robotics platforms for many years, including iRobot, Lego Mindstorm, NI Speed-33, and Parallax. In these platforms, we are given a specifically designed hardware components with built-in support to sensors and actuators, or a range of daughter boards that support different sensors and actuators. These platforms are easy to get started, and we can quickly focus on the research and experiments we want to do. It has been a challenge to develop a robot using standard off-the-shelf computer components. We are worked on the project, we start to realize that it is an advantage to use such components, as we are not limited to the availability and compatibility of robotics platforms and their components.

We started with a lower end JReX-PM board (Figure 12a) with an Intel Celeron M (BANIAS) ULV CPU, 600 MHz, and 512 KB Cache processor. It worked fine until all the parallel services are up and running with continuously data sampling, processing, as well as video streaming. Without a change to the programs developed, we can quickly move to a 986LCD-M/mITX board (Figure 12b) with a much more powerful Intel® Core™2 Duo Mobile processor. As can be seen, the number of ports on the new board also increased.

![Figure 12. Intel single-board computer used in the security robot](image)
Freedom in selecting boards and processors are not the only advantage of this generic approach of building complex robotics systems. On this generic computing platform, we can choose the OS and language that best support our mission. The availability of XP, Linux, Java and C# allow our students to get started with little training. Furthermore, an unlimited range of sensors and actuators from different vendors are simply available and compatible to the single-board computer we use. For example, we used high end USB2.0 Webcam with motion-detection and face-recognition functions, which can largely accomplish the security functions are building.

Implementation details, including the complete list of components used, as well as the code of the service layer are being published in the Robotics Challenge Web site at:

http://asusrl.eas.asu.edu/srlab/Research/RoboticsChallenge.html

6. Summary

Service-oriented robotics software development is a milestone, which extends SOC from its traditional electronic and Web-based applications into a new domain, which was considered not feasible a couple of years ago. Today, many robotics companies have moved into the SOC-based robotics application development paradigm.

This report first presented the concepts, principles, and methods of SOC-based robotics software development, which were first developed in our previous work in 2005, where SOC is applied to develop a single robot with a robot control center. Microsoft's release of the Robotics Studio and VPL in 2006 pushed service-oriented robotics computing into immediate applications. A case study is conducted to illustrate the SOC and event-driven robotics development concepts. Based these studies and Intel's security robot requirement, a distributed security robotics system is designed and presented in this report. The system consists of a Remote Collaboration Center and multiple collaborating robots for unmanned building patrolling. The design of applying SOC and event-driven concepts in the distributed robotics system is discussed in this report detail. The implementation of the first prototype has started in September 2007. Currently, three prototypes are being developed and the first demonstration of these robots is scheduled on April 4, 2008 in the event Arizona Robotics Challenge between teams coming from Arizona State University and the University of Arizona.

7. Acknowledgement

This work is partly supported through Intel's Robotics Challenge initiative and the design objectives are based on Intel's initial specification document of the security robot. ASU graduate students Srushti Abhyankar, Calvin Cheng, Qian Huang, Xin Sun, and Larry Xu, and the undergraduate students in the capstone project, Edward Raleigh, Steven Rowe, Mohammed Taweel, James Woodford, and Jesus Yepez are involved in the design and implementation process of the projects.

8. References


Wei-Tek Tsai, Xiao Wei, Ray Paul, Jen-Yao Chung, Qian Huang, and Yinong Chen, "Service-oriented system engineering (SOSE) and its applications to embedded system development," International Journal of Service Oriented Computing and Applications (SOCA), Springer London, Vol.1 Issue 1, April 2007, pp. 3-17.