

A Robotics Coordination Case Study

Davide Brugali, Luca Gherardi, Patrizia Scandurra

Abstract—Robotics software require and provide a number of different functionalities, which are typically encapsulated in components that cooperate and compete in order to control the behavior of a robot. Cooperation and competition are forms of interaction among concurrent activities and so they have to be coordinated. In order to achieve a good level of reusability and flexibility the coordination and the computation (how the component provides the service) need to be managed separately.

According to this principle we propose an innovative approach based on two frameworks that are widespread in other domains such as the web services. We will show by means of a use case how we have implemented the component through the Service Component Architecture and how we have managed the coordination by means of the Abstract State Machine. In particular we will illustrate that is possible to change the coordination policy without modifying the implementation of the services provided by the component.

I. INTRODUCTION

Robots are controlled through complex software, which requires and provides a number of different functionalities such as motion planning, kinematics, perception and actuation. According to the Component Based Software Engineering principles [1][2], these functionalities can be encapsulated into components. These components improve the level of reusability and reduce the effort and the time needed to develop applications.

The implementation of the components defines how the functionalities are realized. This concern is called computation and is related with the data processing algorithms required by an application [3]. Another important concern, which is orthogonal to the computation, is the coordination. Coordination is more concerned with the interaction of the components [3] and so it defines when the functionalities are used. Components can typically interact in two ways: they cooperate with each other in order to achieve a common goal and at the same time compete for using shared resources such as memory, CPU and external devices (e.g. sensors and actuators). Cooperation and competition are forms of interactions among concurrent activities, which overlap in the time and are interleaved with one other on a single processor. Correct interleaving of concurrent activities can be reached by means of coordination algorithms.

In the context of the European Project BRICS [4] we have chosen two different frameworks in order to model and implement these two concerns. Regarding the computation we have adopted the component model offered by the Service Component Architecture project [5]. SCA is a set of specifications which describe a model for building applications and systems using a Service-Oriented Architecture. Coordination instead is managed by means of the Abstract State Machines (ASM) formal method. ASM is an operational (read: executable) formalism that provides accurate yet practical industrially viable behavioral semantics for pseudo-code on arbitrary data structures. This specification method is tunable to any desired level of abstraction, and besides ASMs comes with a rigorous mathematical foundation [6], it provides rigor without formal overkill.

In particular we adopt a service-oriented flavor of the ASM formalism, named SCA-ASM. SCA-ASM is a formal and executable modeling language based on the open standard model SCA for heterogeneous service-oriented component assembly and on the ASM formal method, which is able to model behavioral notions of service interactions, orchestrations, compensations, and the services internal behavior (see the preliminary work [7]).

The paper illustrates by means of a case study how we have defined a component-based system using Java and the SCA framework and how we have coordinated it using the Abstract State Machines.

The paper is structured as follows.

Section II presents the case study and proposes a high level solution. Section III briefly introduces the ASM and describes how the solution presented in section II can be realized through ASM. Section IV illustrates a SCA implementation of the case study. Finally section V draws the relevant conclusion.

II. THE USE CASE

We propose a simple scenario where a laser scanner offers its services to different clients, which compete for the use of this shared resource.

A. The problem

The scenario is defined by the following three participants, which are illustrated in figure 1.

- A Laser Scanner, which executes scans of the environment on demand and writes the acquired values on a data buffer. A scan is a sequence of measures executed in a single task (for example 360 values, one for each degree). We suppose that the Laser Scanner allows its client to request a scan from an initial angle (start) to a finale one (end) defined as the number of steps between start and end.
• A 3D Perception application, which requests the measures to the Laser Scanner in order to generate a set of meshes that describe the surface of the objects present in the environment.
• An Obstacle Avoidance application, which requests the measures to the Laser Scanner in order to detect the obstacles along the robot path.

Fig. 1. The three participants

B. The requirements

The proposed scenario is subjected to the following requirements:

1) The laser scan is an activity that requires an amount of time in order to be completed. This time is not fixed, and depends on the number of measures requested by the client. During this time the Client could have the need of executing other operations and so it doesn’t have to be blocked while it waits.
2) A client could request a single scan or multiple scans (for example 4 scans composed each one by 20 measures).
3) While the Laser Scanner is executing a scan requested by a client A, a client B could require another scan. These requests have to be managed according to one of the following request management policies:
   • Policy 1: Discard the scan request.
   • Policy 2: Queue the scan request.

According to these requirements it is possible to imagine at least the following three situations, in which the Laser Scanner receives requests from its client.

The first situation is described in figure 2. The client requests a scan to the Laser Scanner and then it waits until the end of the scan process. When the Laser Scanner finishes its work it returns to the client the measures. In this case the request is synchronous.

The second situation is described in figure 3. The client requests a scan to the Laser Scanner and then it continues to execute its work. In this case the request has to be asynchronous.

The third situation is described in figure 4. In this case the Client 1 acts as in the situation two. However while the Laser Scanner is executing the scan requested by the Client 1, the Client 2 sends another request to the Laser Scanner. This example highlights how different clients could simultaneously access to the services offered by the Laser Scanner and so the need of managing these different requests.

C. A high-level solution

The first two situations don’t require a simultaneously access to the Laser Scanner services and so the Client and the Laser Scanner can directly interact. Figure 5 illustrates how the two parts interact. The Client 1 requests a scan to the Laser Scanner, which writes each measure on a Measures Buffer. Then, when the scan is finished:

• The Laser Scanner returns to the Client 1 the measures — Situation 1.
• The Laser Scanner notifies the Client that it has finished its work. After that, the client requests the measures to the Laser Scanner — Situation 2.

The third situation instead presents a simultaneously access to the Laser Scanner services. In this case the interactions between the clients and the Laser Scanner have to be managed by a third part: a coordinator. The Sensor
The Sensor Coordinator policy can be defined by means of the finite state machine. A first version is reported in figure 7. It implements the request management policy 1: if a request is received while the laser is already scanning the new request will be discarded.

The states have the following meaning:
- **IDLE**: the Laser Scanner is idle and is ready for a new scan. In case of a scan request the Sensor Coordinator forwards it to the Laser Scanner. The Sensor Coordinator enters this state on the initialization or when the Laser Scanner returns a “Done” notification.
- **BUSY**: the laser scanner is executing the operations needed for starting the scan. The Sensor Coordinator enters this state when it has sent a scan request to the Laser Scanner. If a new scan request is received the sensor coordinator discards it.
- **SCANNING**: the laser is scanning and writing the measures on the Measures Buffer. The Sensor Coordinator enters this state when the scan request is sent to the Laser Scanner and it has returned the “Ack” notification. If a new scan request is received the sensor coordinator discards it.

Depending on the number of scan requested the Sensor Coordinator will forward to the Laser Scanner one or more single scans.
Scanner. Furthermore this version, as well as the first, is able to satisfy single scan requests.

The states have the same meaning described above. What changes are the transition rules:

- **IDLE → BUSY**: the transition is triggered when the Sensor Coordinator receives a scan request.
- **BUSY → SCANNING**: the transition is triggered when the Laser Scanner sends a “Ack” notification to the Sensor Coordinator.
- **SCANNING → BUSY**: the transition is triggered when the Laser Scanner sends a “Done” notification to the Sensor Coordinator and the number of remaining scans (“remScans”) is greater than 0.
- **SCANNING → IDLE**: the transition is triggered when the Laser Scanner sends a “Done” notification to the Sensor Coordinator and there are not remaining scans to execute.

Figure 9 shows the finite state machine that implements the policy 2: if a request is received while the laser is already scanning the new request will be queued. Also in this case the states and the functionalities provided by the laser scanner are the same. What changes are the transition rules.

The new transitions are described in the following list.

- **IDLE → BUSY**: the transition is triggered when the Sensor Coordinator receives a scan request.
- **BUSY → BUSY**: the transition is triggered when the Sensor Coordinator receives a scan request. The request is queued.
- **BUSY → SCANNING**: the transition is triggered when the Laser Scanner sends a “Ack” notification to the Sensor Coordinator.
- **SCANNING → BUSY**: the transition is triggered when the Laser Scanner sends a “Done” notification to the Sensor Coordinator and the number of remaining scans is greater than 0, or the Laser Scanner sends a “Done” notification to the Sensor Coordinator, there are not remaining scans to execute and the number of pending requests in queue is greater then 0.
- **SCANNING → SCANNING**: the transition is triggered when the Sensor Coordinator receives a scan request. The request is queued.
- **SCANNING → IDLE**: the transition is triggered when the Laser Scanner sends a “Done” notification to the Sensor Coordinator, there are not remaining scans to execute and there are not pending requests in queue.

The next section will illustrate how we have modeled these finite state machines through the ASM language.

III. THE ASM SPECIFICATION OF THE USE CASE

The Sensor Coordinator was implemented using the SCA-ASM formalism. In the following an ASM-based (abstract) implementation of the Sensor Coordinator is reported.

A. The Abstract State Machine method in a nutshell

Abstract State Machines (ASMs) are an extension of Finite State Machines (FSMs) [6] where unstructured control states are replaced by states comprising arbitrary complex data. The states of an ASM are multi-sorted first-order structures, i.e. domains of objects with functions and predicates (boolean functions) defined on them. The transition relation is specified by rules describing how functions change from one state to the next. Basically, a transition rule has the form of guarded update if Condition then Updates where Updates is a set of function updates of the form \( f(t_1, \ldots, t_n) := t \) which are simultaneously executed when Condition is true.

There is a limited but powerful set of rule constructors, reported in Table I, that allow to express simultaneous parallel actions (par) of a single agent, either in an atomic way, Basic ASMs, or in a structured and recursive way, Structured or Turbo ASMs, by sequential actions (seq), iterations (iterate, while, recwhile), and submachine invocations returning values. Appropriate rule constructors also allow non-determinism (existential quantification choose) and unrestricted synchronous parallelism (universal quantification forall). Furthermore, the ASM method supports a generalization where multiple agents interact in parallel in a synchronous/asynchronous way, Synch/Asynch Multi-agent ASMs. In this last model, the predefined variable self is interpreted by each agent as itself.

Based on [6], an ASM can be defined as the tuple:

\( \text{(header, body, main rule, initialization)} \)
The header contains the name of the ASM and its signature\(^1\), namely all domain, function, and predicate declarations. Function are classified as derived functions, i.e. those coming with a specification or computation mechanism given in terms of other functions, and basic functions which can be static (never change during any run of the machine) or dynamic (may change as a consequence of agent actions or updates). Dynamic functions are further classified into: monitored (read and write by an agent and by the environment or by another agent) and output (only write) functions.

The body of an ASM consists of (static) domain and (static/derived) function definitions according to domain and function declarations in the signature of the ASM. It also contains declarations (definitions) of transition rules. The body may also contain definitions of invariants to assume over domains and functions of the ASM.

The (unique) main rule is a transition rule and represents the starting point of the machine program (i.e. it calls all the other ASM transition rules defined in the body). The main rule is closed (i.e. it does not have parameters) and since there are no free global variables in the rule declarations of an ASM, the notion of a move does not depend on variable assignment, but on the machine state.

The initialization of an ASM is a characterization of the initial states. An initial state defines initial values for domains and functions declared in the ASM signature.

Executing an ASM means executing its main rule starting from a specified initial state. A computation of an ASM \(M\) is a finite or infinite sequence \(S_0, S_1, \ldots, S_n, \ldots\) of states of \(M\), where \(S_0\) is an initial state and each \(S_{n+1}\) is obtained from \(S_n\) by firing simultaneously all of the transition rules which are enabled in \(S_n\).

A lightweight notion of module is also supported. An ASM module is an ASM (header, body) without a main rule, without a characterization of the set of initial states, and the body may have no rule declarations. A module is written as an ASM with the keyword `asm` replaced by the keyword `module`.

An open framework, the ASMETA tool set [8], based on the Eclipse/EMF modeling platform and developed around the ASM Metamodel, is also available for editing, exchanging, simulating, testing, and model checking models.

### B. The ASM-SCA formalism

In addition to the ASM rule constructors, other commands capturing service behavioral aspects have been provided (see [7], [9] for more details) and formalized in terms of ASMs concepts as further actions offered by the SCA-ASM language, including constructs to express the control flow of component’s tasks, as well as primitive for services orchestration and interaction. Some of these actions correspond to predefined ASM rules whose AsmetaL implementation is provided in terms of an external library, named Common-Behavior, to be imported as part of a SCA-ASM module.

In particular, external services are invoked in a synchronous and asynchronous manner through the following interaction (or communication) primitives:

- `wsend[lnk,R,snd]`: Sends data “snd” without blocking to the partner link “lnk” in reference to the service operation “R” (no acknowledgment is expected).
- `wreceive[lnk,R/rcv]`: Receives data in the location “rcv” from the partner link “lnk” in reference to the service operation “R”; it blocks until data are received. No acknowledgment is expected.
- `wsendreceive[lnk,R,snd/rcv]`: In reference to the service operation “R”, some data “snd” are sent to the partner link “lnk”, then the action waits for data to be sent back, which are stored in the receive location “rcv”; no acknowledgment is expected for send and receive.
- `wreplay[lnk,R,snd]` Returns some data “snd” to the partner link “lnk”, as response of a previous “R” request received from the same partner link; no acknowledgment is expected.

These communication primitives rely on a dynamic domain Message that represents message instances managed by an abstract message passing mechanism, abstracting, therefore, from the SCA notion of binding. We assume only that components communicate over links according to the semantics of the communication commands reported above and a message encapsulates information about the partner link and the referenced service name and data transferred. A data binding mechanism also guarantees a matching between ASM data types and Java data types, including structured data.

\[^1\]Import and export clauses can be also specified for modularization.
The following listings report the ASM implementation of the Sensor Coordinator FSM depicted in figure 8 (request management policy 1). To this purpose, the AsmetaL textual notation to write ASM models within the ASMETA tool-set is used. Two grammatical conventions must be recalled: a variable identifier starts with an initial $; a rule identifier is used. Two grammatical conventions must be recalled: a variable identifier starts with an initial $; a rule identifier is used. Two grammatical conventions must be recalled: a variable identifier starts with an initial $; a rule identifier is used.

Listing 1 shows the first lines of the ASM implementation. The import clauses include the ASM modules of the provided service interfaces (in our case SensorCoordinating and EventObserving) and required interfaces (in our case the LaserScanning interface) of the component, annotated, respectively, with “@Provided” and “@Required”. The “@MainService” annotation when importing the SensorCoordinating interface denotes the main service (read: main component’s agent) that is responsible for initializing the component’s state (in the predefined “r_init” rule) and, eventually, for the start-up of the other agents by assigning programs to them.

The signature of the machine contains declarations for:

- References (shared functions annotated with @Reference), which are abstract access endpoints to services.
- Back references to requester agents (shared functions annotated with @Backref).
- Declarations of the following ASM domains and functions, which are used by the component for internal computation only.
  - The enumeration domain “State” defines the possible states of the ASM shown in section II-C.
  - The variable “ctl_state” stores the current control state of the ASM.
  - The variable “paramScan” is used to store the input parameters of the “request” function.
  - The variable “from” is used to store the start position of a scan request received from a client.
  - The variable “steps” is used to store the number of measures that compose a scan request received from a client.
  - The variable “remScans” is used to store the number of scans that are requested by a client.
  - The variable “event” is used to store the input parameter of the “update” function.

The body of the ASM, which starts with the keyword “definitions:” includes definitions of services (ASM transition rules annotated with @Service) “r_request” and “r_update”, the definition of the main transition rule “r_SensorCoordinator” (that takes by convention the same name of the component’s module) and the transition rule with the predefined name “r_init” that is in turn invoked in the initialization rule of the container composite to initialize the internal state (controlled functions). Another utility rule, named “r_acceptRequest”, has been introduced for modularization purposes and to advance the control state of the machine according to the arriving service requests properly.

Listing 2 reports the body of the “r_request” rule. It is in charge of requesting a scan to the laser scanner. When the rule is called, it executes the following operations in a parallel way:

1) Sets the state of the ASM to BUSY.
2) Stores the parameters of the requested scan in the variables “from”, “steps” and “remScans”.
3) Calls the function “scan”, which is provided by the service Laser Scanning.

Listing 3 reports the body of the “r_update” rule. It is in charge of receiving the notification from the laser scanner and updating the control state of the ASM. When the rule is called, it executes the following operations:

1) If the current control state is BUSY and the notification is an “Ack”: the rule sets the control state to SCANNING.
2) If the current control state is SCANNING and the notification is a “Done” and the number of remaining scans is greater than 0, the rule executes the following operations in a parallel way:
   - Sets the control state to BUSY.
   - Decrements the number of remaining scans.
   - Calls the function “scan”, which is provided by the service Laser Scanning.
3) If the current control state is SCANNING and there are not remaining scans to do: the rule sets the control state to IDLE.

Listing 4 reports the body of the "r_acceptRequest" rule. It is the main rule of the agent and is called every time a client requests a service offered by the Sensor Coordinator. This rule simply forwards the request to the "r_acceptRequest" rule (see above).

Listing 5 reports the body of the "r_SensorCoordinator" rule. It is in charge of processing the request received from the clients of the coordinator and so it receives as input a string, which contains the request. When the rule is called, it sequentially executes the following operations:

1) If the client has requested a new scan(r_request):
   a) It removes the request from the requests stack (operation "r_wreceive") and stores the input parameters in the variable "paramScan".
   b) If the parameters are defined (condition "isDef") the rule calls the rule "r_request" (see above)
2) If the Laser Scanner has sent a notification (r_update):
   a) It removes the request from the requests stack (operation "r_wreceive") and stores the input parameter in the variable "event".
   b) If the parameter is defined the rule calls the rule "r_update" (see above).

The rule is implemented in such a way that all the scan requests received while the scanner is already scanning are discarded (that’s what the policy 1 defines).

Listing 6 reports the body of the "r_init" rule. It is called in order to initialize the agent. This rule simply sets the status of the agent to READY, the control state to IDLE and initializes the scan parameters to 0.

Listing 7 shows the new implementation of the rule "r_acceptRequest". The lines 1-15 are the same of the previous implementation. A third "if" condition was added (lines 16-20) in order to manage the requests received while the control state is different from IDLE. In fact, when the control state is not IDLE and the request received is a scan request, the rule sequentially executes the following operation:

- It removes the request from the requests stack (operation "r_wreceive") and stores the input parameters in the variable "paramScan".
- It puts the parameters in a queue called "pendingRequests" (operation "append"). This queue is defined in the header of the state machine and is the equivalent of an array which stores triplets of integers (the three parameters of a scan request).

All what is needed to queuing the scan requests stays in this rule. Indeed, as mentioned above, this is the rule that implements the request management policy. The changes in the rule "r_update" are instead necessary in order to manage the requests queue.
public interface MeasuresBufferReading {
    public LaserScan getScan();
}

public interface MeasuresBufferWriting {
    public void writeMeasure(LaserMeasure measure);
}

public interface LaserScanning {
    //...
Fig. 10. The Sensor Composite

Listing 9. The Java interfaces of the components

In order to manage the notification received from the Laser Scanner the Sensor Coordinator also implements the interface reported in the listing 10. So far it is used as a Service in order to simulate a callback, because the callbacks are not yet supported in the SCA-ASM Eclipse plugin. It will be used as a SCA Callback as soon as this further feature will be implemented and supported.

Listing 10. The Java EventObserving interface

The ASM definitions of the sensor coordinators provided interfaces are reported in the listing 11 using the AsmetaL notation. They are ASM modules containing only declarations of business agent types, declared in terms of subdomains of the predefined ASM Agent domain, and of business functions, declared as parameterized ASM out functions.

Listing 11. ASM definition of the Sensor Coordinating interface

V. CONCLUSIONS

In this paper we have presented a case study regarding the coordination in the context of a robotics application. The case study was developed by using SCA and ASM and has demonstrated that the use of these frameworks, which are widespread in other domains, is feasible also in the robotics field.

The use of two different frameworks for modeling two different concerns (SCA for computation and ASM for coordination) improves the level of flexibility and reusability.
Thanks to this orthogonal separation, we were able to implement two different policies without modifying the implementation of the services offered by the Laser Scanner. In the same way it is possible to modify the implementation of the Laser Scanner services without modifying the coordination state machine.

Moreover ASM allows automatic validation and verification of the correctness and reliability of single components taken in isolation. It also consents runtime monitoring of the components and self-adaptation.

These positive features, the results demonstrated through our case study, the level of maturity of these frameworks and their significant spread in other domains such as the web services, support and advocate our thesis according to which the integration of SCA and ASM promises good results also in the robotics field.

VI. ACKNOWLEDGMENTS

The authors would like to thank all the partners of the BRICS project for their valuable comments.

REFERENCES


