Computational Reflection in order to support Context-Awareness in a Robotics Framework

Sheila Mendez  
R&D Manager  
B2B E-Commerce  
s.mendez@b2b2000.com

Francisco Ortin  
Computer Science Department  
University of Oviedo  
ortin@lsi.uniovi.es

Miguel García  
Computer Science Department  
University of Oviedo  
be37378@uniovi.es

Vicente García-Díaz  
Computer Science Department  
University of Oviedo  
garcavi@uniovi.es

Abstract—The development of service robots has gained more and more attention over the last years. Advanced robots have to cope with many different emerging at runtime situations, while executing complex tasks. They should be programmed as context-aware systems, capable of adapting themselves to the execution environment, including the computing, user and physical environment. Since computational reflection is a programming language technique that offers a high level of runtime adaptability, we have analyzed the suitability of this language feature to fulfill the dynamism requirements of context-aware robotic systems. In order to evaluate their appropriateness, we have implemented an example scenario in a dynamic reflective language and compared it with Java.

Index Terms—Computational reflection, context-awareness, robotics framework, dynamic languages.

I. INTRODUCTION

Robotic systems which should be able to interact in everyday life have to manage the high dynamism and complexity of real-world environments. An important element to be considered is the environment context, commonly referred to as location, the identification of nearby people and objects, plus changes to those objects [1]. Contexts are defined to be the constantly changing execution environment, including the computing, the user and the physical environment [2]. Therefore, context-aware software is software that must adapt itself according to its location of use, the collection of nearby people and objects, and changes to those objects over time [1]. An advanced robotic platform should take into account the context-awareness requirements of future robotic applications.

Dynamic languages have recently turned out to be suitable for specific scenarios such as Web development, application frameworks, game scripting, interactive programming, rapid prototyping, dynamic aspect-oriented programming and any kind of runtime adaptable or adaptive software. The main benefit of these languages is the simplicity they offer to model the dynamism that is sometimes required to build high context-dependent software. Common features of dynamic languages are meta-programming, reflection, mobility and dynamic reconfiguration and distribution. Computational reflection is one of the most distinguishing features of these languages, defined as the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions [3]. Due to the strong connection between context-awareness and computational reflection, we present in this paper how computational reflection can be used to facilitate the programming of context-aware services in a robotic platform. In fact, it has been previously stated that in order “to support context-awareness in an open and much larger setting, a reflective, or self-describing, context model is required” [4].

Dynamic languages have been previously used in different robotics scenarios. An example is the SmartTCL language, an extension of Lisp that was used to implement the sequencing layer in a three layer robotic architecture [5]. The sequencing layer is the place to store procedural knowledge on how to configure skills to behaviors. Due to the dynamic features of SmartTCL, the plans stored inside the task coordination module can be modified easily at runtime.

The dynamic language Lua has also been used to implement the behavior engine of the humanoid robot Nao [6]. The formalism of hybrid state machines (HSM) was used to bridge the gap between high-level strategic decision making and low-level actuator control. The model of HSMs was extended with dependencies and sub-skills to call the behaviors or skills hierarchically. Lua turned out to be an expressive language to implement these HSM-skills.

Other examples of existing robotics products that use dynamic languages to obtain a high level of adaptability are UrbiScript, an orchestration language for robotics systems; ROS (Robot Operating System), a set of libraries and tools to help software developers create robot applications in Python; and Pyro (Python Robotics) a programming environment for exploring advanced topics in artificial intelligence and robotics, facilitating the creation of interfaces for accessing and controlling a wide variety of real and simulated robots.

II. THE TIC4BOT PROJECT

The work presented in this paper is part of the TIC4BOT Project developed by the Treelogic Company, the Cartif Foundation, and the University of Oviedo. The aim of the project is to provide the necessary infrastructure to develop complex services in the social robotics field, by raising the abstraction level. The implementation was tested over a real robotic platform (SCITOS-G5), though the system was designed with total independence of the hardware. In order to get this independence, the Player/Stage [7] system was used during the development process. The SCITOS-G5 robot is controlled by an embedded PC with an Intel Core 2 Duo processor, and multiple small hardware units that monitor several functions of the robot. The PC operating system is Fedora Core 8.

Figure 1 shows the system architecture, consisting of three layers: primary modules, robotics framework and service modules.
has happened.

the robotic platform. An event is a notification that something

events. A primitive is defined as a low-level function that can

libraries in C and C++ and they implement primitives and

Primary Modules are developed accessing the hardware

platform greatly suffices this requirement. The

to perform the required tasks. In the case of our project, the

robot hardware should have enough processing capabilities

at the expense of lower runtime performance. This implies that

collection and multi-threading. However, these benefits come

chose the Java programming language. Java raises the abstrac-

platform independence and a higher level of abstraction, we

of primary modules and service modules [8]. In order to obtain

correspondence with the primary functions.

Dynamic runtime detection of primary modules

Primary modules are developed accessing the hardware

libraries in C and C++ and they implement primitives and

events. A primitive is defined as a low-level function that can

perform a simple task or a query over any sensor or actuator of

the robotic platform. An event is a notification that something

has happened.

In the framework, primitives are classified into namespaces. Each namespace contains a set of primitives, with a one-to-one correspondence with the primary functions.

The framework provides a middle layer for the integration

of primary modules and service modules [8]. In order to obtain

platform independence and a higher level of abstraction, we

chose the Java programming language. Java raises the abstraction

level, providing automatic memory allocation, garbage collection and multi-threading. However, these benefits come at the expense of lower runtime performance. This implies that the robot hardware should have enough processing capabilities to perform the required tasks. In the case of our project, the SCITOS-G5 processor greatly suffices this requirement. The main functionalities provided by the framework are:

- **Dynamic runtime detection of primary modules**, allowing the addition of new primary modules at runtime. We allow the implementation of these modules in both C and C++, generating a Java proxy class at runtime for each C/C++ module. Proxies are classes providing the access to primary module functions. These Java proxies were developed using the JNI (Java Native Interface) standard through SWIG (Simplified Wrapper and Interface Generator) [9], that provides an automated connection of programs written in C and C++ with a variety of high-level programming languages.

A. Primary Modules

Primary modules are developed accessing the hardware libraries in C and C++ and they implement primitives and events. A primitive is defined as a low-level function that can perform a simple task or a query over any sensor or actuator of the robotic platform. An event is a notification that something has happened.

In the framework, primitives are classified into namespaces. Each namespace contains a set of primitives, with a one-to-one correspondence with the primary functions.

B. The Framework

The framework provides a middle layer for the integration of primary modules and service modules [8]. In order to obtain platform independence and a higher level of abstraction, we chose the Java programming language. Java raises the abstraction level, providing automatic memory allocation, garbage collection and multi-threading. However, these benefits come at the expense of lower runtime performance. This implies that the robot hardware should have enough processing capabilities to perform the required tasks. In the case of our project, the SCITOS-G5 processor greatly suffices this requirement. The main functionalities provided by the framework are:

- **Dynamic runtime detection of primary modules**, allowing the addition of new primary modules at runtime. We allow the implementation of these modules in both C and C++, generating a Java proxy class at runtime for each C/C++ module. Proxies are classes providing the access to primary module functions. These Java proxies were developed using the JNI (Java Native Interface) standard through SWIG (Simplified Wrapper and Interface Generator) [9], that provides an automated connection of programs written in C and C++ with a variety of high-level programming languages.

```java
// Authentication
Subject subject = null;
subject = Authorization.login("admin", "defaultPassword");
// Retrieve Namespace and Primitive
 WeinerManager api = WeinerManager.getWeinerManager(FactoryManagerType.Default);
Namespace namespace = api.getNamespaceByName("Navigation");
Primitive primitive = namespace.getPrimitiveByName("goTo");
// Parameters creation
Object[] parameters = new Object[2];
parameters[0] = x;
parameters[1] = y;
// Invocable element for the framework
InvocationElement invocationElement =
    new InvocationElement(subject, namespace, primitive, parameters);
// Create a list of invocation elements
List<InvocationElement> list = new ArrayList<InvocationElement>();
list.add(invocationElement);
// Create the task (Execution Mode)
Task task = new DefaultTask("Sample Task");
task.setInvocationElements(list);
// Create the Runnable (Priority)
Runnable runnable = executionManager更重要的InsertTask(task,
    Runnable.Priority.MAX, RunnableType.SYNCHRONOUS);

Figure 2. Sample code of framework programming.

Figure 2 shows a sample code of the invocation of a primitive. First, authentication over the framework is performed and credentials for executing tasks are obtained. Then, the Navigation namespace and its goTo primitive are obtained. An Object array is created containing the values of parameters for the invocation. A list of InvocationElement is also created. The InvocationElement type is provided by the framework and it could contain a set of primitive elements that will make up a complete task.

- **Transparent publication of all the elements offered by the primary modules**. The framework provides the necessary mechanisms to discover and invoke primitives at runtime. It also provides event subscription management.

The next step of our sample code in Figure 2 is to create a task with the InvocationElement list previously mentioned. In this sample code, the type of the created task is DefaultTask. The task type indicates the execution mode of the primitives within the task (a DefaultTask indicates non-transactional behavior, while a TransactionalTask indicates that the task cannot be interrupted by any other task until it finalizes its execution). Finally, a Runnable object is created as the result of introducing the previously created task into the framework execution engine. In this stage, the execution mode is established to SYNCHRONOUS. Synchronous tasks cannot be executed in parallel with other synchronous ones, whereas asynchronous tasks can. The task is assigned the maximum priority execution level (MAX). The Runnable object is then used to retrieve the result of the primitives executed. The loop in the code waits until the result of the primitive is ready.

- **Execution engine for service modules**, providing a priorityization mechanism of tasks and different execution modes (transactional or non-transactional, and syn-
chronous or asynchronous), which can be combined in each specific case.

- **Remote hot reprogramming at runtime.** This service enables remote applications to send their Java source code to the framework at runtime. This was developed through Web Services, using standard technologies. We used the W3C recommended WSDL (Web Services Description Language) for describing Web Services, and the SOAP (Simple Object Access Protocol) protocol to specify the interchange of data with Web Services by means of XML messages.

**C. Service Modules**

Service modules are complex modules that provide end-user services by means of the composition of primary modules functionalities. The access to primary modules is always performed through the framework. Service modules insert sets of tasks in the framework with a concrete priority and execution mode. These modules can be added at runtime through the hot reprogramming service.

### III. COMPUTATIONAL REFLECTION

Reflection is “the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions” [3]. The computational domain of reflective languages includes their self-representation. Therefore, they can offer their structure and semantics as computable data.

Since context-aware systems should dynamically adapt to runtime changing environments, computational reflection seems to be a suitable technique to face this kind of scenarios. We previously used computational reflection as a suitable technique for adaptive systems such as persistence management [10], dynamic aspect-oriented programming [11] or heterogeneous device support [12].

One classification of reflection considers the observation and modification issues of the system self-representation:

- **Introspection**: Self-representation of programs can be dynamically consulted but not modified. The applications can obtain information about runtime classes, objects, methods, etc. This level of reflection is offered by languages such as Java or C#.
- **Intercession**: The ability of a program to modify its own execution state, interpretation or meaning. Most dynamic languages offer this feature.

Another classification can be established according to what can be reflected:

- **Structural Reflection**: System structure can be accessed. In case the system structure is modified, changes will be reflected at runtime. An example of this kind of reflection is the Python feature of adding fields and methods to both objects and classes.
- **Behavioral Reflection**: This level of reflection implies access to system semantics. In case the semantics is modified, it involves a customization of the runtime behavior of programs. As an example, Python offers overriding the semantics of member lookup; if a class has a __getattr__ method, it will be called whenever a non-existing member is accessed.

Another feature that is commonly used together with computational reflection is runtime generative programming [13]. It consists of the capability of programs to generate new (parts of) programs. This feature is usually offered in conjunction with reflection, because those new parts of generated programs modify the structure or semantics of running applications.

### IV. ADAPTIVE DYNAMIC PROGRAMMING LAYER

The adaptive dynamic programming layer is a new tier over the framework that supports adaptive context-awareness capabilities. This layer is designed to offer runtime adaptation to dynamic environments. Context-aware scenarios often involve the addition of new requirements at runtime, making use of services offered by the framework that are consulted at runtime.

Since dynamic languages offer a high degree of runtime adaptability, we propose their use to provide a runtime adaptive system that additionally provides a simplified way of context-aware tasks. This enhances the framework services with dynamic languages features, obtaining an additional layer with a higher degree of flexibility and adaptability.

Since the framework is developed in Java, an intercommunication mechanism between Java and Python is required. The standard Java Scripting API (JSR 223) [14] allows the use of script engines from Java code. By using it, all the functionality provided by the framework in Java will be accessible at the adaptive dynamic layer using a dynamic language.

We selected the Python programming language to implement the adaptive dynamic programming layer because it is a mature dynamic language that provides runtime structural and behavioral reflection, dynamic generative programming, a simple syntax, a substantial number of powerful libraries and functions.

**A. Framework Services**

Although the framework provides useful features for many use cases, the code for one single operation over the framework may become verbose (as shown in Figure 2) due to the lack of metaprogramming features in Java. It implies the codification of several lines of code that performs non-functional actions, entailing a less agile development of new services. One of the main goals of the dynamic adaptive layer is to simplify the programming over the framework.

In the adaptive dynamic programming layer, authorization and synchronization data is stored along the entire programmer session, and it is used in all the operations in a transparent way. Furthermore, this data could be changed at any moment by the programmer. Figure 3 shows how the authentication and task priority assignments (and synchronization specification) are performed in the adaptive layer. In addition to this, credentials and synchronization data are stored in the adaptive layer module. When an operation is executed over the framework, this data is retrieved and used.
B. Primitive Management

In the case of primitives, the dynamic adaptive layer allows the final programmers to write their code in a more natural and compact way, facilitating its maintainability and legibility, and without losing any functionality. This layer provides primary module discovery at runtime, making it possible to act over new services discovered at runtime, even if they were not present at design time.

As it was explained in Section 2, primitives in the framework are organized in namespaces. At runtime, when a primary module is discovered, its primitives and namespaces objects are created. In the adaptive dynamic programming layer, generative programming and structural reflection are used to transparently create classes that wrap the framework services. The resulting code is a Python class that has a one-to-one equivalence for each namespace object in the framework. For every object instance representing a Namespace with sets of Primitive objects in Java, a class with the corresponding methods is transparently created in Python. The purpose of this generation is to provide a simple and natural way to invoke those primitives discovered at runtime. Figure 3 shows the Python source code that invokes the goTo primitive at the adaptive programming layer. This code is equivalent to the Java program in Figure 2, being much more compact and legible.

```python
from FrameworkLoader import *
# Authorization request
authenticate(USER, PASSWORD)
# Priority, execution mode and name of the task
setTaskType(RunnablePriority.ONE, TRANSACTIONAL_TASK, "HOT-REPROGRAMMING")
# Object corresponding to Namespace Navigation
navigation = Navigation()
# Primitive execution and retrieve of result
result = navigation.goTo(x, y)
```

Figure 3. Programming over the adaptive layer.

Code generation is implemented using the exec Python function. This function is allows strings to evaluate strings that generate new classes at runtime. The strings are parameterized with the name of a namespace class using the % operator. Figure 4 shows the skeleton string used to generate namespaces classes (we do not show the code in the methods for the sake of brevity; it can be consulted in [15]). Generated classes do not contain primitive methods; they provide a mechanism to alter the message passing mechanism through behavioral reflection. Generated classes implement a dynamic lazy search for the invoked method (primitive) using introspection. This search is performed over the framework and it is completely transparent to the final user.

As Figure 3 and its sequence diagram in Figure 5 show, these generated classes make use of the two __call__ and __getattr__ built in methods that offer behavioral reflection. The goTo message is passed to the navigation object. Since this member is not offered by the object, the __getattr__ method is called. This method stores the name of the requested member (i.e., goTo) so it can be later used for searching and invoking the corresponding primitive. Since the returned object (navigation) implements the __call__ method, it can be called as it was a function. The __call__ method receives as many parameters as the actual arguments of the call, in a variable-length argument list. This method receives the x and y parameters, and performs the search of the primitive that fits this signature. Once found, the appropriate primitive of the framework is invoked and the __call__ method returns the result of calling the primitive.

Figure 4. Sample code to be dynamically evaluated.

```
classCode = "*
# Import sentences
class $(className):
    def __init__(self):
        # Constructor code
    def __getattr__(self, item):
        # Saves the name of the member
    def __call__(self, *args, **kwargs):
        # Invokes the corresponding primitive
*"
```

Figure 4. Sample code to be dynamically evaluated.

Computational reflection and generative programming allowed us to fulfill the adaptability requirements without needing to specify all the classes and their methods at design time. This also allows the development of context-aware emerging at runtime scenarios. Furthermore, we can introduce complete service modules on the framework at runtime, using its reprogramming feature. These service modules could introduce new behavior patterns to undertake a specific goal.

We implemented a simple navigation prototype based on fuzzy rules to show the simplicity and adaptiveness provided by the dynamic programming layer. The main move function is shown in Figure 6. This function provides the functionality to make the robot advance a step, making use of fuzzy rules.

The rules have the following meaning, expressed with an antecedent and a list of consequents: 1) if there is no obstacle close, go to the target; 2) if the front-left side is freer than the front-right side, turn left and continue until right sonar sensors are free of obstacles; 3) if the front-right side is freer, turn right and continue until left sonar sensors are free of obstacles. An evaluateFuzzyRules function evaluates every rule premise and executes the consequents of that rule whose antecedent value is greater.

These three rules provide the logic of decision making for
def move(xPos, yPos):
    global x, y
    x = xPos
    y = yPos
    rules = [
        (fuzzyNot(obstacleClose), [goToTarget]),
        (fuzzyAnd(obstacleClose, isFrontLeftMoreFree),
            [turnLeft, moveUntilLeftIsFree])
    ]
    evaluateFuzzyRules(rules)

Figure 6. Sample fuzzy rules.

the navigation algorithm to avoid obstacles while going toward
a specific point. Although this sample logic is very simple, it
may be improved by just adding more fuzzy rules describing
an optimized behavior of the robot.

Since Python supports first-class functions, fuzzy rules were
defined using functions that represent fuzzy operators,
predicates and actions. Fuzzy operators (e.g., fuzzyAnd and
fuzzyNot) can be implemented in Python as higher-order
functions, i.e., functions that take functions as parameters. The
source code is freely available at [15].

C. Event Management

The adaptive dynamic layer enables any single function to
subscribe to a concrete event, complementing the framework
services and primitive management provided by this layer. In
addition, new events can be discovered at runtime, so that
programmers can dynamically access new events published
dynamically.

Following with our example, let’s suppose that we have an
artificial vision primary module capable of detecting a person’s
face. At the moment the face is recognized, the artificial vision
primary module triggers an event with the data of the face
(e.g., the name of the recognized person). Figure 7 shows
an example subscription to a concrete event in the adaptive
dynamic programming layer:

- Authentication is performed in the first place. Afterwards,
prioritization is established to the maximum, and the
asynchronous execution mode is chosen. With these
settings, the Greet task is executed in parallel with the
navigation task that is running in the framework. If the
robot is performing any other task at the moment the
event is triggered, like for example moving toward any
point, it will be able to look at the face detected and greet
at the same time, without having to stop its navigation.

- After this, the function that handles the event is defined.
This function instantiates a namespace of primitives
called Speech. This namespace performs tasks of voice
speech synthesis through the say primitive, which
retrieves a text and reproduces it by simulating human voice,
greeting the recognized person.

- The last statement subscribes the previous function to the
face detection event. Therefore, when the event is thrown,
the handlerFaceDetectionEvent function will be
automatically called.

By applying generative programming and structural reflection,
each event in the framework corresponds to a generated

Python class. These classes implement the Observer interface
(from the Observer design pattern) provided by the framework.
When an instance of any of these event classes is created, a
subscriber for that event is also registered. The constructor of
this class receives a Python function as parameter. When the
framework notifies the occurrence of an event, the subscriber
delegates its management to that function.

Because of the dynamic discovery and event generation
code, it is possible to add new event types at runtime without
having to specify at design time all the events offered by the
framework. Furthermore, it allows performing simultaneous
tasks, making the most of the prioritization and synchroniza-
tion features provided by the framework.

D. Remote Reprogramming

The framework execution engine can be remotely repro-
grammed at runtime. This remote hot reprogramming allows
the introduction of new behavior guidelines at runtime, without
having to foresee them at design time, even parallelizing them
with other tasks that might be running.

We are interested in allowing the robot to react to the
environment context. As an example, we have included in our
example the scenario proposed by Pineau et al. [16], where
the robotic platform has the capability of assisting elderly
individuals with an automated reminder system. This system
can, for instance, remind to the patient that it is time to have
her medicine.

The framework exposes one Web Service for the addition
of new service modules from a remote system, which, after
authorization, can send the source code to the framework exe-
cution engine. The code received is added to the tasks queue,
being executed according to its priority and synchronization
settings. Making use of the remote reprogramming feature
offered by the framework, the Reminder module can be added
at runtime.

Figure 8 shows the source code we used to develop the
reminder scenario. The Navigation module shown in Section
IV-B is imported, authentication is performed, and the task
is set as transactional. This execution mode (transactional)
allows obtaining the execution control and avoids other tasks
to interrupt it, notwithstanding the parallel execution of other
asynchronous tasks (e.g., when the Reminder module is run-
ning, other asynchronous modules like the Greeting one could
be executed in parallel). The patientLocatorWS reference
points to a Web Service that provides the coordinates of
patients. Then, a getPosition function that retrieves the
coordinates of a patient passing her identification it is defined.
The main program contains a loop that iterates while the

# Authentication and Priorization
authenticate("admin", "defaultPassword")
setTaskType(RunnablePriority.MAX, ASYNCHRONOUS_TASK, "Greet")

# Event Handler
def handlerFaceDetectionEvent(data):
    speech = Speech()
    speech.say("How are you + data.name + ")

# Event Subscription
event = FaceDetectionEvent(handlerFaceDetectionEvent)
design time. Additionally, the use of dynamic languages and computational reflection involved a significant simplification of the code to program the dynamic adaptive layer.

The source code of the whole example presented in this paper is freely available at http://www.reflection.uniovi.es/tic4bot. Although the robotics framework is not freely downloadable because it belongs to the TreeLogic Company, the URL above provides a demonstrating video showing the example presented in this paper running on the robotics framework.

ACKNOWLEDGMENTS

This work has been funded by the Spanish Ministry of Industry, Tourism and Commerce (TSI-020100-2008-235). It has also been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation; project TIN2008-00276.

REFERENCES