SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2012; **00**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

On the suitability of dynamic languages for hot-reprogramming a robotics framework: a Python case study

Francisco Ortin*[1], Sheila Mendez[2], Vicente García-Díaz[1], Miguel Garcia[1]

[1] *University of Oviedo, Computer Science Department, Calvo Sotelo s/n, 33007, Oviedo, Spain*
[2] *B2B 2000, Research & Development Department, Av. de la Vega 4, 33940, El Entrego, Spain*

SUMMARY

The development of service robots has gained more attention over the last years. Advanced robots have to cope with many different situations emerging at runtime, while executing complex tasks. They should be programmed as dynamically adaptive systems, capable of adapting themselves to the execution environment, including the computing, user and physical environment. Recently, dynamic languages are becoming widely used due to the high runtime adaptability they offer. Therefore, we have analyzed the suitability of these languages to implement robotic systems with high runtime adaptability requirements, using Python as case study because of its maturity. In order to evaluate their suitability, we have implemented a reflective robotics framework that can be programmed in both Java and any dynamic language supported by the standard Java Scripting API. An example scenario has been developed using Python to show how its distinguishing meta-programming features have facilitated the development of runtime-adaptable robotics services.

Received . . .

KEY WORDS: Computational reflection; generative programming; robotics framework; dynamic languages; Python; Java Scripting API.

1. INTRODUCTION

The robotic systems that should be able to interact in everyday life have to manage the high dynamism and complexity of real-world environments. They should consider environment elements such as the identification and location of nearby people and objects [1]. Constantly changing execution environments including the computing, the user, and the physical environment should also be taken into account [2]. A highly adaptable robotic platform has to fulfill the adaptability requirements of future robotic applications to solve problems in dynamic reactive environments. The runtime discovery of new services to expose new behavior patterns must also be provided, without limiting the number of sensory capabilities and behavioral patterns [3]. In these systems, new requirements and services could appear at runtime, not foreseen at design time.

There are previous works focused on developing robotics services that require dynamic adaptation. An example is the *Nursebot* project, based on a mobile robotic assistant [4]. It was developed to assist elderly individuals with mild cognitive and physical impairments, and to support nurses in their daily activities. They considered the task of reminding people of events and guiding them through their environments. In the *Museum Traffic Control* project [3], a robot generated cues that cause visitors to travel to portions of the museum that were normally avoided, being part of an ambient-intelligent system that interacts with humans. In [5], a "smart walker" robot was designed

*Correspondence to: University of Oviedo, Computer Science Department, Calvo Sotelo s/n, 33007, Oviedo, Spain.

to provide mobility assistance to the frail elderly visually impaired. They used a Bayesian network to combine user input with high-level information derived from the sensors, providing an estimate of the user's current navigation goals. More information regarding highly runtime-adaptable robotic systems is depicted in Section 5.

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 runtime adaptable software. Common features of dynamic languages are reflection, dynamic code generation and evaluation, 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 [6]. Due to the strong connection between highly runtime-adaptable robotic systems and reflective dynamic languages, we have investigated how dynamic languages could be used to facilitate the programming of runtime adaptable services in robotic platforms. 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* [7]. Existing approaches have previously taken advantage of the distinctive features of dynamic languages for controlling and simulating robotic systems, being Pyro [8], ROS [9] and UrbiScript [10] three well-known examples—a comparison with our system is presented in Section 5.2.

The main contribution of this paper is the identification of how the distinguishing features of dynamic languages can be used to program dynamically adaptive services in a robotics framework. For this purpose, we have developed a Java robotics framework (called TIC4BOT) whose main purpose is programming robotics services in any dynamically typed programming language and showing how these languages can be used for the benefit of robotics. Using Python as a case study, we have implemented an example scenario to show how dynamic languages can be used to program runtime-adaptable robotics services. The particular features of dynamic languages have been used in different use cases to provide remote hot-reprogramming of services, adaptation of existing components at runtime, insertion of new primitives and events, and allowing the use of new Web services discovered at runtime. These features are offered dynamically, requiring neither stopping the framework execution nor knowing what elements are going to be adapted at runtime. Some other approaches like service-oriented component-based software systems also provide a high level of runtime adaptability. Using the example scenario presented in this paper, a comparison between both approaches is also detailed in Section 5.1.

The developed scenario is inspired by the *Nursebot* project [4], where a robot provides reminder services to patients in a nursing home. As an example, other implementations of this project extract the schedule information from the entries of a user calendar [4] [7], using an accurate model of the user daily schedule. In our proposed approach, in contrast, no information about the user schedule is provided when the application is executed. Our system provides dynamic adaptability to new services emerging at runtime, exploiting the distinguishing features of dynamic languages. Following the basic requirements of mobile robot architectures defined in [11], our work is mainly focused on evaluating with different use cases the appropriateness of dynamic languages to fulfill the *extendibility and scalability* requirements.

1.1. Specific Requirements of Highly Dynamic Robotic Systems

The following are particular requirements fulfilled by highly dynamic robotic systems such as service-oriented component-based software systems. We briefly analyze the suitability of dynamic languages to address each requirement and the distinctive feature of dynamic languages we have used to accomplish it. Each dynamic language feature is detailed in Section 2.

1. **Dynamic hot reprogramming of the system**. In some circumstances a robotic system should be able to react to dynamically emerging requirements, not foreseen at design time. We have

DOI: 10.1002/spe

used the *dynamic code evaluation* feature of dynamic languages to dynamically add new pieces of code that interact with the rest of the running system.

2. **Homogeneous discovering of dynamic services published by the framework**. Whenever a new service is published, it should be accessible to any other running service. *Introspection* is used to consult the list of services provided and the structure and functionality of each one. *Duck typing* allows the use of these new services without needing to define a complex hierarchy of interfaces and classes, dynamically discovering the object methods and fields.
3. **Interactive, compact and straightforward programming of services**. In many scenarios, the programming of new services simply "glues" together the existing primitives, modules and services in the robotic system. This is a common situation where dynamic languages have been identified as appropriate because of their high level of abstraction and their flexible and dynamic type system. They facilitate interactive development, used as scripting languages. We have also used *structural* and *behavioral intercession* to include cross-cutting concerns to the functional code provided by the programmers, freeing them from implementing unnecessary code.
4. **Discovering and interaction of running applications**. The existing applications and components in the robotics framework may be able to dynamically discover and interact with the rest of applications and components in the system. The main feature of dynamic languages we have used to fulfill this requirement is *introspection*.
5. **Standards-based bidirectional communication between the system and dynamically discovered remote devices / systems**. In our proof-of-concept scenario, we represent new remote devices and systems by means of Web services that publish their functionality with standard WSDL documents. The programmers may easily create new robotics services that establish a bidirectional communication with these remote devices. They only have to specify the URL of the WSDL: *dynamic code generation* and *structural* and *behavioral intersection* are used to dynamically create a specific proxy class for that Web service; *introspection* is used to analyze the messages offered by this new class; and *duck typing* is used in the code that invokes methods to be created later on, at runtime, when the Web service is dynamically discovered.
6. **Dynamic adaptation of existing services**. Existing applications may be adapted when new requirements emerge at runtime. Since the dynamic adaptation of running applications is one of the most common scenarios of dynamic languages [12], those services developed using dynamic languages in the robotics framework will benefit from this adaptability feature. *Intercession* is the main feature used to perform this adaptation.

The rest of this paper is structured as follows. In the next section we introduce the meta-programming features of dynamic languages and their relationship with adaptable robotic systems. Section 3 provides the description of the project, and an overview of the general architecture and the location of the adaptive dynamic programming layer within the system. Section 4 describes the adaptive dynamic programming layer, and related work is discussed in Section 5. Section 6 presents the conclusions and future work.

2. META-PROGRAMMING FEATURES OF DYNAMIC LANGUAGES

One of the distinguishing features of dynamic languages is meta-programming. It is commonly referred to as the programming language capability of writing (parts of) programs that manipulate programs, including themselves, as data. The same way a programmer modifies an application when new requirements must be fulfilled; meta-programming provides this program modification at runtime, without needing to stop the application. This involves a great runtime flexibility to handle new situations without stopping the running application.

The two main programming features offered by dynamic languages to allow meta-programming are computational reflection and dynamic code generation and evaluation (also known as generative programming).

DOI: 10.1002/spe

2.1. Computational Reflection

Reflection is *the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions* [6]. The computational domain of reflective languages includes their self-representation. Therefore, they can offer their structure and semantics as computable data. Reflection happens when the system is performing *computation about (and by possibly affecting) its own computation* [6].

Since runtime adaptable robotic systems should dynamically adapt to runtime changing environments, computational reflection seems to be a suitable technique to face this kind of scenarios. We have previously used computational reflection as a suitable technique to build dynamically adaptive systems such as persistence management [13], dynamic aspect-oriented programming [14] or heterogeneous device support [15].

There exist different classifications of reflection [16]. The first classification has been established considering 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 entities such as classes, objects and methods. Introspection is offered not only by dynamic languages, but also by some statically typed languages such as Java or C#. In our system, introspection is used to dynamically discover the primitives, components and services offered by the robotic platform. A basic example use of introspection in Python is the `globals` function. It returns a dictionary containing all the global variables, including functions. The key set collects the names of global variables, and the values collection holds the references to those global variables. Python has first-class functions, meaning that functions are treated as first-class objects. Therefore, the values collection in the dictionary returned by `globals` holds function variables.
- **Intercession**: The ability of a program to modify its own execution state, interpretation or meaning. Most dynamically typed languages offer this feature, whereas nearly all the statically typed ones do not.

Another common classification is established according to *what* can be reflected:

- **Structural Reflection**: The changes performed in the application structure are reflected at runtime. Python allows adding fields and methods to both objects and classes. Figure 1 shows a straightforward fragment of Python code that uses structural reflection. First, a `Proxy` class with no members and a `proxy` instance are created. Out of the scope of the class, it is defined an `f` function that assigns the second value received to the `item` field of the `self` parameter, and returns `self`. The next sentence assigns the `f` function to the new `__getattr__` method of the `Proxy` class. This is a simple example of dynamic method addition (`__getattr__`) to the `Proxy` class using structural reflection.
- **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. For instance, Python allows the overriding of the semantics of member lookup. If a class has a `__getattr__` method, it will be called whenever a non-existing member is accessed. Therefore, the code in Figure 1 assigns the name of the member accessed to the `item` field by means of behavioral reflection, even if it does not exist in the object.
  The source code in Figure 1 also uses behavioral reflection to change the semantics of the message passing mechanism. If an object implements the `__call__` method, then it will be callable; i.e., it can be invoked, receiving the actual parameters of the invocation. In the `proxy.getISSN("SP&E")` invocation, the `getISSN` member can be accessed even though the member does not exist, saving the name of the member (`getISSN`) in the `item` field, and returning the `proxy` object (see the `f` function in Figure 1). Since `proxy` implements the `__call__` method, the invocation to `getISSN` is correct, even though that method is not actually implemented by the object. In this case, a simple example message is displayed showing the name of the method called (`getISSN`) and its parameters (`"SP&E`"),

DOI: 10.1002/spe

```
class Proxy:
    pass
proxy = Proxy()
def f(self, item):
    self.item = item
    return self
def g(self, *args):
    print 'The "%s" method has been invoked, \
                with the following params:' % self.item
    for i in range(len(args)):
        print "\tParam %s, value = %s, type = %s" \
                % (i+1, args[i], type(args[i]))
# Structural reflection
Proxy.__getattr__ = f
Proxy.__call__ = g
# Behavioral reflection
proxy.getISSN("SP&E")
```

Figure 1. Sample reflective Python code.

but using this technique a real proxy mechanism [17] could have been implemented. The main advantage of this approach is that the delegate can be created at runtime, even after instantiating the `Proxy` class.

2.2. *Runtime Generative Programming*

Another feature that is commonly used together with computational reflection is runtime generative programming [18]. It is defined as the capability of programs to generate new (parts of) programs at runtime. This feature is usually offered in conjunction with reflection, because those new parts of generated programs may modify the structure and semantics of the running applications by means of reflection. The generated code could be persistent (i.e., saving it into files) when we want to use it in future program executions, or volatile (i.e., using character strings in memory) when it is not required to keep the new functionality after program execution.

Dynamic code evaluation in Python is provided by the `exec` and `eval` functions. These functions evaluate at runtime the character string or text file passed as an argument. Figure 2 shows a Python example that combines dynamic code evaluation and reflection. First, the `classCode` variable is assigned a string that, when evaluated, creates a class that, using behavioral reflection, is able to invoke any method. This code is parameterized with the name of the class to be created (`Navigation` in our example). When the `goTo` message is passed, the `__call__` method, using the `globals` introspective function, searches for a `goToPrimitive` function to be invoked. This search is performed at runtime. It is also possible to pass the `greet` message to the `navigation` object, but only after defining the `greetPrimitive`—otherwise, a runtime exception would be raised.

Another distinguishing feature of dynamic languages is *duck typing*. Although duck typing is not considered as a meta-programming feature, it is highly related with it. Duck typing[†] is a property offered by most dynamic languages that means that an object is interchangeable with any other object that implements the same dynamic interface, regardless of whether those objects have a related inheritance hierarchy or not. As an example, when we write the `navigation.goTo(12,3)` expression, `navigation` could be any object that implements a `goTo` method receiving two integer values. It is not necessary to declare that reference as an interface or class that provides that `goTo` message. It could be any object that, in that exact point of execution, provides that particular method.

[†]If it walks like a duck and quacks like a duck, it must be a duck.

Prepared using speauth.cls
DOI: 10.1002/spe

```
classCode = """
class %(className)s:
    def __getattr__(self, item):
        self.item = item
        return self
    def __call__(self, *args, **kwargs):
        return globals()[self.item+"Primitive"](*args)
"""
def goToPrimitive(x, y):
    print "going to (" + str(x) +", "+str(y) +  ")"
exec(classCode % ({"className": "Navigation"}))
navigation = Navigation()
navigation.goTo(12, 3)
def greetPrimitive(name):
    print "How are you, " + name +  "?"
navigation.greet("SP&E")
```

Figure 2. Sample dynamic code evaluation in Python.

One of the outcomes of duck typing is that it supports polymorphism without using inheritance. Therefore, the role of the abstract methods and interfaces as mechanisms to specify contracts is made redundant. Since it is not necessary to define polymorphic inheritance hierarchies, software can be developed more quickly and the source code is more compact. It also facilitates writing code that invokes methods that have not been created when the application is being developed, because type-checking is postponed until runtime.

3. THE TIC4BOT PROJECT

The work presented in this paper is part of the TIC4BOT Project [19][20]. This project was 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 adaptable services in the social robotics field (as well as primary modules to support them), by raising the level of abstraction. The implementation was tested over a real robotic platform (SCITOS-G5), though the system was designed with total independence of the hardware. The Player / Stage [21] system was used to obtain this independence, deploying the project directly over the Stage emulator. The SCITOS-G5 robotic platform 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 operating system is Fedora Core 8.

Figure 3 shows the system architecture, consisting of three layers: primary modules, the robotics framework and service modules. The system has a simple architecture compared to larger generic systems such as ROS [9] and SmartSoft [22]. This simplicity has facilitated its modification to experiment with the suitability of meta-programming features of dynamic languages to build runtime adaptable robotic systems.

3.1. Primary Modules

Primary modules are aimed at accessing the hardware libraries in C and C++ using the Java Native Interface (JNI) standard. Primary modules implement primitives and events. A primitive is a low level function that can perform a simple task or a query over any sensor or actuator of the robotic platform (e.g., a query that retrieves the distance from the robot position to an obstacle). An event is a notification that something has happened, sent by a primary module (e.g., the robot has hit something). Although possible, neither components nor applications should be modeled as primary modules. For that purpose, our system provides Java modules (Section 3.2) and services (Section 3.3).

DOI: 10.1002/spe

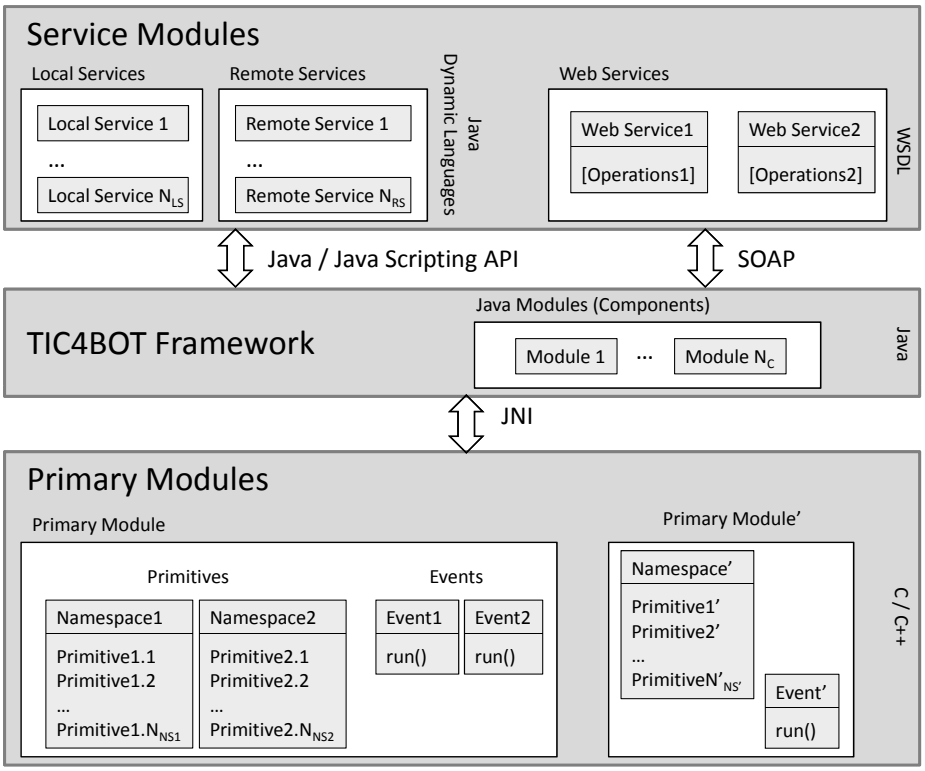

Figure 3. Architecture of the TIC4BOT system.

In the framework, primitives are classified into *namespaces*. Each *namespace* contains a set of primitives, with a one-to-one correspondence with primary functions. Therefore, it is necessary to retrieve a specific *namespace* and search for the correct primitive before invoking it. This process is interleaved with authentication and authentication subsystems, which have the responsibility to grant and deny access to the framework services (next subsection).

3.2. The Framework

The framework provides common basic functionalities plus a middle layer for creating Java modules (components) which integrate primary modules and service modules [23]. A Java module or component (Figure 3) is a collection of classes in a package that provides a functionality to be reutilized by different applications (service modules). They commonly increase the abstraction level of primary modules. Java modules can be parameterized like any Java class, and interact with other components to provide a specific functionality. Service modules may use components to implement robotic applications.

The framework allows developing components (and services) with different programming languages, without worrying about the underlying technology used to develop primary modules. It raises the level of abstraction and allows discovering of new services and modules added at runtime. We chose the Java programming language to obtain platform independence and a higher level of abstraction. Java raises the abstraction level, providing automatic memory allocation, garbage collection and multi-threading (among other features). However, these benefits come at the expense of lower runtime performance [24]. 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 platform fulfills this requirement. The main functionalities provided by the framework are:

- *Authentication and authorization*. All the operations in our system are controlled by the authentication and authorization subsystem based on the JAAS (Java Authentication and

DOI: 10.1002/spe

```
// Authentication
Subject subject = null;
subject = Authorization.login("admin", "password");
// Retrieve Namespace and Primitive
ApiManager api = FactoryManager.getApiManager(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)
ExecutionManager executionManager = FactoryManager.
            getExecutionManager(FactoryManagerType.Default, subject);
Runnable runnable = executionManager.insertTask(task,
            RunnablePriority.MAX, RunnableSyncronism.SYNCHRONOUS);
// Retrieve the result
Object result;
while (true){
    try {
        result = runnable.getResult(0);
        break;
    } catch (NullValueException e) {
        Thread.currentThread().sleep(100);
    }
}
```

Figure 4. Sample code of framework programming.

Authorization Service) framework [25]. We first check whether users can access the system, validating their login and password. Then, we use a Java security policy file to enforce a set of permissions granted to each user, classifying them into groups (currently we have *administrators*, *programmers*, *testers* and *users*). After authentication, the system generates the credentials associated to the user group. The authorization subsystem of the framework checks that the user has sufficient rights to perform an operation (e.g., hot reprogramming, retrieval of running tasks, access to a specific primitive, etc.) each time the user application requests to do so.

- *Runtime detection of primary modules*, allowing the dynamic addition of new primary modules. The framework provides the implementation of primary modules in both C and C++, generating one Java proxy class at runtime for each C / C++ module. Proxies are classes that provide access to primary module functions. These Java proxies are developed using JNI through SWIG (Simplified Wrapper and Interface Generator) [26]. SWIG offers an automated connection of programs written in C and C++ with a variety of high-level programming languages. This automation is essential to offer the dynamic addition of primitives and events, and their runtime discovery through reflection.
  Figure 4 shows part of a Java module (component) that invokes 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 (next point in the enumeration).
- *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, so that service modules and components can subscribe to any event in order to be notified when the event is triggered.

DOI: 10.1002/spe

The next step of our sample code in Figure 4 is to create a task by means of `InvocationElements`. An `InvocationElement` is the mechanism provided by the framework to access primitive modules. A list of `InvocationElements` is created to define the collection of modules that make up a complete task. Finally, a `Runnable` object is created as the result of introducing the previously created task into the framework execution engine. The `Runnable` object is used to retrieve the result of the `goTo` primitive invocation. The loop in the code waits until the result of the primitive invocation is ready.

- *Concurrent task execution and synchronization*. The framework allows the concurrent execution of different tasks. For this purpose, we define synchronous and asynchronous tasks. Asynchronous tasks can be executed in parallel with other tasks, whereas two synchronous tasks cannot be executed concurrently. The synchronous navigation task created in Figure 4 allows other asynchronous tasks to run in parallel. However, any other synchronous task should wait for this task to finish.
  To establish priorities in the execution of tasks, the framework provides different levels of prioritization—the task in Figure 4 has the maximum priority level for programmers (`MAX`). When two or more synchronous tasks are waiting to be executed, the one with higher priority is chosen (in case of a tie, a FIFO policy is used). Priorities are also used to compute the percentage of time slicing to be assigned in the scheduling of asynchronous tasks.
  As shown in Figure 4, tasks are created with a list of `InvocationElements` (i.e., invocations to primitive modules). A `DefaultTask` indicates that its list of invocations can be interrupted by any other task. When all the `InvocationElements` in a task must be executed atomically (without any interruption in their invocations), the `TransactionalTask` should be used instead. This behavior is orthogonal to the synchronization type of the task.
  Task execution can be terminated by the user. The `getExecutionManager` method of the `FactoryManager` *Singleton* object returns an implementation of the `ExecutionManager` interface. `ExecutionManagers` return `Runnable` objects by passing the task identifier as a parameter. Passing the `stop` message to a `Runnable` object causes the termination of the corresponding task. This operation is controlled by the authorization subsystem and can only be performed by either the task owner or the system administrator.
- *Remote hot reprogramming at runtime*. This service enables remote systems to send their components and services to the framework at runtime. This service is provided through Web services, using standard technologies. The first one is WSDL, the W3C recommendation for describing Web services. It is XML-based and widely used. The operations and messages are abstractly described and bound to a concrete network protocol and message format to define an endpoint. Moreover, WSDL is extensible, allowing the description of endpoints and messages, being independent of message formats and network protocols. In this project, WSDL is used in combination with SOAP (Simple Object Access Protocol) [27]. The SOAP protocol specifies the interchange of data with Web services by means of XML messages.
  Our framework allows remote reprogramming using either Java or any dynamic language supported by the Java Scripting API (see Section 4). Java developers can use tools like simulators (e.g., Player) to test their code before uploading the new functionality. The server compiles the Java code upon uploading. If there are compilation errors, the framework returns a string containing the error messages. Otherwise, the component code is installed in the framework, and executed in case it is a service module. Service modules are deleted by the framework once their execution is finished. The remote hot reprogramming Web service also provides component deletion and update. Java components are identified by their package name, whereas module names are used for Python. The authentication and authorization subsystem of the framework checks whether the user has sufficient rights to remove or update a component. If any runtime error occurs at runtime, the exception is caught by the framework and the execution of the service is aborted.

DOI: 10.1002/spe

3.3. Service Modules

Service modules represent robotic applications that provide end-user services by means of the composition of installed components. The access to components and primary modules is 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, using either Java or a dynamic language. The framework creates a new Java thread for each service module, but all the threads share the same task scheduler described in Section 3.2.

It is worth noting that the framework has been designed to offer a high level of runtime adaptability. First, new primitives, components and services can be added at runtime, while applications are running on the framework. Second, these new functionalities can be discovered and used by existing and future services. Finally, services and components developed with dynamic languages can be modified at runtime using the meta-programming features of these languages.

4. ADAPTIVE DYNAMIC PROGRAMMING LAYER

The adaptive dynamic programming layer is a new tier over the framework that supports the development of services capable of adapting to dynamic environments. As it was mentioned before, runtime adaptation scenarios often involve the fulfillment of new requirements at runtime, making use of the services offered by the framework.

Dynamic languages offer a high degree of runtime adaptability, facilitating the agile and interactive development. Therefore, we analyze the suitability of dynamic languages to provide a runtime adaptive system that additionally provides a simplified way of programming runtime adaptable tasks. We enhance the framework with an additional layer that provides a higher degree of adaptability using dynamic languages.

This is an outline of how we have used the specific features of dynamic languages to design the adaptive dynamic programming layer, facilitating the runtime development and deployment of new services:

- **Dynamic code evaluation**. Its main use has been the dynamic hot reprogramming of the system, facilitating the rapid interactive development of new services.
- **Introspection**. It has been used to dynamically discover primitives, services and components in the framework, along with remote devices and services.
- **Structural Intercession** allows the dynamic modification of the structure of objects, classes and modules. We have primarily used this feature to perform dynamic adaptation of existing services.
- **Dynamic code generation**. Its main use has been the runtime generation of proxy classes that perform a bidirectional communication with remote devices and systems. This code is transparently generated when the programmer specifies the URL of a remote WSDL document.
- **Behavioral Intercession** has been used to extend the semantics of the message passing mechanism, simulating methods and fields that objects do not actually implement. An example use is the bidirectional communication with other systems, where messages of a local proxy class are dynamically translated into SOAP message calls to a remote Web service.
- **Dynamic Duck Typing** makes the code simpler, because it is not necessary to define complex class and interface hierarchies to make the code statically type-checked by the compiler. Since type checking is performed at runtime, it is remarkably simple to write code that makes use of primitives, components and services that are not available when the application is being coded. The result is a more compact and simpler source code, facilitating the interactive remote programming of services.

Since the framework is developed in Java, an intercommunication mechanism between Java and dynamic languages is required. The Java Scripting API (JSR 223, Scripting API for the Java Platform) [28] is a scripting language framework to allow the use of script engines from Java code.

DOI: 10.1002/spe

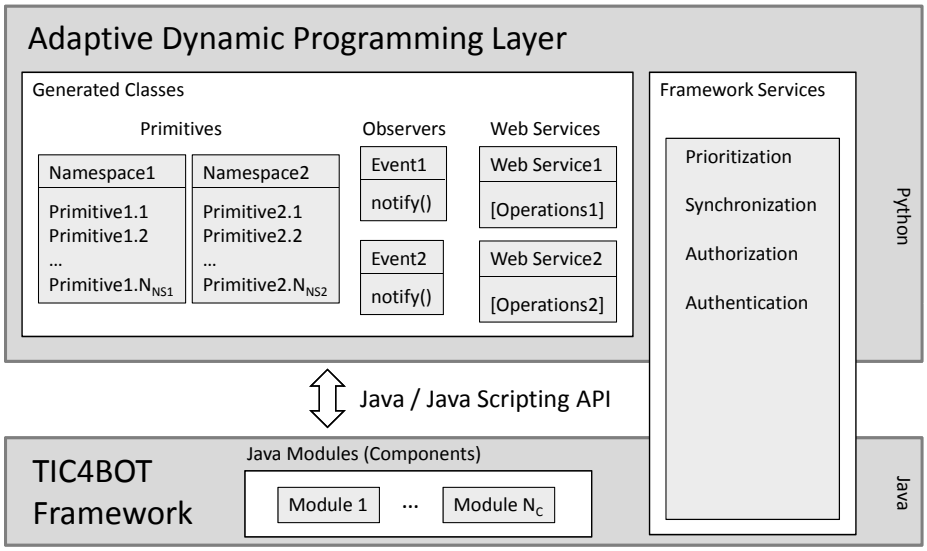

Figure 5. Design of the adaptive dynamic programming layer.

It allows the programmer to work with both Java and any dynamic language for which there is an implementation of it, regardless of whether it is a domain-specific language or a general purpose one. New languages can be added to our framework without changing the existing code.

Figure 5 shows an overview of the adaptive dynamic programming layer in the system architecture. Using the Java Scripting API, the adaptive layer allows the implementation of customizable and extensible applications over the TIC4BOT framework. All the functionality provided by the framework in Java can be programmatically accessed at the adaptive layer using a dynamic language.

We selected the Python programming language [29] to implement the adaptive dynamic programming layer because it is a mature dynamic language that provides runtime structural and behavioral reflection, dynamic code generation, a simple syntax, and a substantial number of libraries and built-in functions readily available.

This layer enables the discovery of primary modules and Web services at runtime, generating code at runtime that encapsulates these services. Different levels of computational reflection and generative programming have been used to undertake this goal. First, introspection has been employed to search the suitable method at each invocation performed by the generated code. Structural reflection and intercession have also been used, together with generative programming, to access wrapper classes (classes that wrap the framework primitives) and their dynamically created methods. Since new primitives could be dynamically added, wrapper classes implement a lazy method loading mechanism, searching for the suitable primitive at runtime—as shown in Figure 2. This technique has also been used to invoke the operations remote Web services. The lazy invocation mechanism is achieved by changing the semantics of the Python message passing mechanism. The next subsection describes how this process is performed using an example scenario.

4.1. An Example Scenario

We have developed a prototype inspired by the scenario proposed by Pineau *et al.* [4] to show how the adaptive layer can be programmed with dynamic languages to provide runtime adaptable services. In this scenario, the robotic system should assist elderly individuals with an automated reminder system (e.g., reminding patients to have their medicine). The main purpose of implementing this scenario is to identify how the distinguishing features of dynamic languages may be used to program this kind of runtime-adaptive robotics services. Therefore, we show how to use dynamic languages for the following tasks: user authentication, creation of synchronous tasks, access to basic movement primitives, creation of an example fuzzy-based navigation component,

DOI: 10.1002/spe

asynchronous event handling, remote reprogramming, and transparent access and use of Web services. We have first tested the implementation in a simulated environment using Player / Stage. Then, we have used SCITOS-G5 in a laboratory with obstacles, reproducing moving patients with simulated positioning devices.

4.1.1. Authentication, Synchronization and Task Creation

In Section 3 we described the overall responsibilities of each layer of the system architecture. Although the framework provides useful features for many use cases, the Java code for one single operation over the framework may become verbose, as shown in Figure 4. The inherent verbosity is mainly due to the lack of meta-programming features in Java, its static type system, the big range of services offered by the framework (authentication, authorization, synchronization and execution modes) and its own design (e.g., using widespread design patterns like *Singleton* and *Façade* [17], having to retrieve *Singleton* instances many times as shown in Figure 4). This implies the codification of several lines of code that perform non-functional actions, entailing a less agile development of services. One of the main goals of the dynamic adaptive layer is to simplify the programming over the framework, following the *Separation of Concerns* (*SoC*) principle [30][31], that allows the reutilization of non-functional code. Meta-programming features of dynamic languages are commonly used to achieve this goal [14].

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 performed over the framework in a transparent way. Furthermore, this data could be changed at any moment by the programmer. The code in Figure 6 creates a simple task, showing how authentication, synchronization and task priority assignment are performed in the adaptive layer. This code is the Python version of the Java program previously shown in Figure 4. User credentials and task synchronization data are stored in this service module, so that this data can be retrieved and used by the framework to check authorization when an operation is executed. The use of dynamic languages facilitates data storage along the session because Python is not pure object oriented and it allows choosing the scope of variables, classes and methods. In our case, session variables are stored in the Python global scope of each service module.

4.1.2. Primitive Management

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

As it was explained in Section 3, primitives in the framework are organized in *namespaces*. At runtime, when a primary module is discovered, its primitives and *namespace* objects are created. In the adaptive layer, generative programming and structural reflection are used to transparently generate 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 dynamically generated in Python. The purpose of this generation is to provide a simple way to invoke those primitives discovered at runtime. Figure 6 shows the Python source code in the adaptive layer that invokes the `goTo` primitive. This code is equivalent to the Java program in Figure 4, being much more compact and legible.

Code generation is implemented using the `exec` Python function. We evaluate strings that generate new classes at runtime, following the technique shown in Figure 2. The generated code creates new `Namespace` classes (the whole code can be consulted in [32]). These classes do not contain primitive methods actually; they provide a mechanism to alter the message passing mechanism through behavioral reflection. Generated classes implement a dynamic lazy search of the invoked method (primitive) using introspection. This search is performed over the framework and it is completely transparent to the final user. These generated classes make use of two built-in methods of Python behavioral reflection: `__call__` and `__getattr__` (their meaning was explained

DOI: 10.1002/spe

```
from FrameworkLoader import *
# Authorization request
authenticate(USER, PASSWORD)
# Priority, execution mode and name of the task
setTaskType(RunnablePriority.ONE, TRANSACTIONAL_TASK, "GoTo")
# Object corresponding to Namespace Navigation
navigation = Navigation()
# Primitive execution and retrieve of result
res = navigation.goTo(X, Y)
```

Figure 6. Basic task creation over the adaptive layer.

in Section 2.1). If a method of a class is called and it is not found, the `__getattr__` method is called instead. The `__call__` method receives as many parameters as the actual arguments of the invocation, in a variable-length argument list. By combining these two methods, we can simulate the behavior of a simple call to an object method. Figure 7 represents the UML sequence diagram of the code in Figure 6, depicting how the dynamically generated `Navigation` class responds to the `goTo` method invocation. The call process is as follows:

1. The user is first authenticated.
2. A new instance of the dynamically generated `Navigation` class is created.
3. The `goTo` method is searched in the `navigation` object.
4. The `__getattr__` method is called after failing the previous search.
5. The `__getattr__` method stores the name of the requested member (i.e., `goTo`), so that it can be later used for searching and invoking the corresponding primitive. `self` is then returned.
6. The call is performed over the object returned, and hence the `__call__` method is invoked. This method receives the `x` and `y` parameters, and performs the search of the primitive that fits this signature. If found, the appropriate primitive of the framework is invoked, and the `__call__` method returns the result of calling the primitive. Therefore, behavioral reflection is used to define new semantics of the message passing mechanism.

Generative programming and computational reflection have been the two main features of dynamic languages used to dynamically discover and invoke the new primitive modules provided by the framework. The code is significantly more compact and legible than the Java version.

4.1.3. A Navigation Component

The `goTo` primitive of the `Navigation` *namespace* shown in Figure 6 does not consider obstacles in the environment. Therefore, we have implemented a prototype navigation component based on fuzzy rules to make the robot navigate to a patient location. Although the navigation logic could have been expressed in Python using imperative functions, we have used a simple system based on fuzzy rules. The use of fuzzy rules was merely for flexibility purposes because it allowed us to easily modify the navigation logic by simply changing the rules.

This example also shows how Python features allow the representation of abstractions (such as fuzzy rules) commonly expressed by means of specific domain languages. We have chosen a general purpose dynamic language for the following reasons:

- Taking an existing language suppresses the necessity of designing a new language and implementing an interpreter.
- Any functionality, component, API or framework available for the Python programming language can be used in the implementation of applications.
- Source code is more readable, because Python is a widely used programming language [33] and there is plenty of documentation about it.

We used the three simple rules shown in Figure 8 represented in Python by means of lists and tuples. Each rule is modeled with a tuple, where the first value is the antecedent and the second

DOI: 10.1002/spe

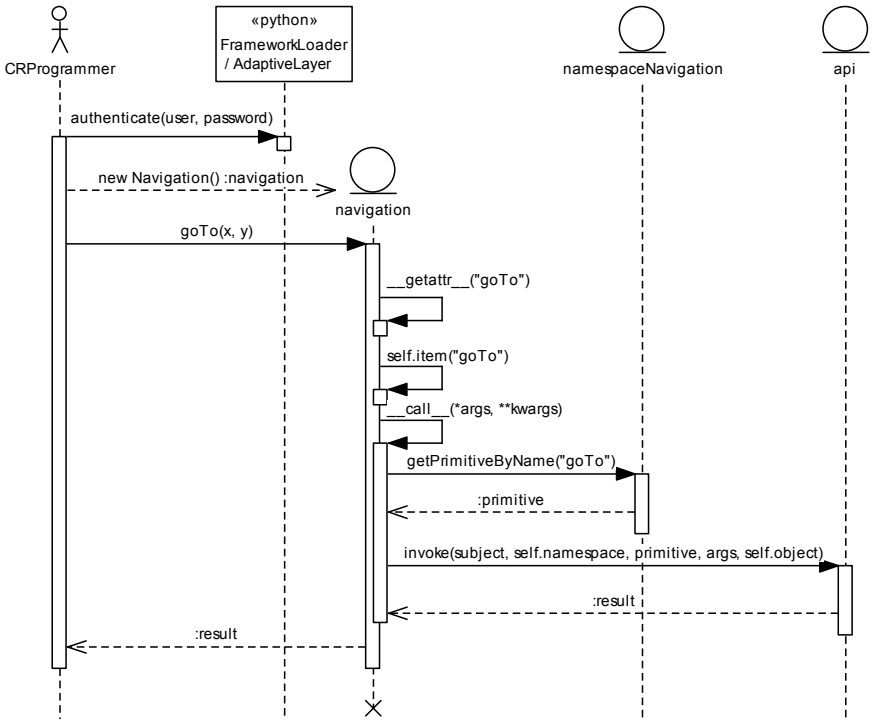

Figure 7. Access to primitive services.

```
rules = \
  [ ( FuzzyNot( obstacleClose ),                      [goToTarget]                     ), \
    ( FuzzyAnd( obstacleClose, isFrontLeftMoreFree ),  [turnLeft, moveUntilRightIsFree] ), \
    ( FuzzyAnd( obstacleClose, isFrontRightMoreFree ), [turnRight, moveUntilLeftIsFree] )  \
  ]
```

Figure 8. Sample fuzzy rules.

one holds the list of consequents. The rules we used in our example (Figure 8) have the following meaning:

1. If there is no obstacle close, go to the target.
2. If the front-left side is more free than the front-right side and there is an obstacle close, turn left and continue until right sonar sensors detect no obstacle.
3. If the front-right side is more free, turn right and continue until left sonar sensors detect no obstacle.

The antecedent of the first rule is not true when there is an obstacle close. Rules 2 and 3 make the decision about which path to take in that case. These three rules provide the logic of decision making for the navigation algorithm to avoid obstacles while going towards a specific point. Although this logic is very simple, it can be improved by adding more fuzzy rules describing an optimized behavior of the robot.

The way these rules are evaluated is shown in the `evaluateFuzzyRules` function in Figure 9. This function evaluates every rule antecedent and executes the consequents of that rule whose antecedent value is the greatest. At runtime, if the rules are modified, the changes would be automatically reflected in the behavior of the robot. The main `move` function is shown in Figure 9. This function provides the functionality to make the robot advance toward the target position. The rules are evaluated while the objective is not reached.

Since Python supports first-class functions (i.e., functions can be treated as first-class objects), fuzzy rules can be defined with variables representing predicates (`obstacleClose`,

DOI: 10.1002/spe

```
def move(xPos, yPos):
    global x, y
    x, y = xPos, yPos
    while not isTargetReached():
        evaluateFuzzyRules(rules)
```

```
def evaluateFuzzyRules(rules):
    values = []
    # Evaluates all the antecedents
    for rule in rules:
        values.append(rule[0]())
    # Gets the consequents with max antecedents
    consequents = rules[values.index(max(values))][1]
    # Calls the consequents
    for consequent in consequents:
        consequent()
```

Figure 9. Evaluating fuzzy rules in Python.

```
class FuzzyNot:
    def __init__(self, function):
        self.function = function
    def __call__(self, *args):
        return 1 - self.function()
```

```
class FuzzyAnd:
    def __init__(self, function1, function2):
        self.function1 = function1
        self.function2 = function2
    def __call__(self, *args):
        return min(self.function1(), self.function2())
```

Figure 10. Sample fuzzy operators.

`isFrontLeftMoreFree` and `isFrontRightMoreFree`) and actions (`goToTarget`, `turnLeft`, `moveUntilRightIsFree`, `turnRight` and `moveUntilLeftIsFree`). Making use of the `__call__` reflective method, it is possible to define operator objects and use them as functions. This allows using either functions or fuzzy operators as antecedents. Sample code of two fuzzy operators is shown in Figure 10. The `fuzzyAnd` operator was implemented as the minimum of both values, and the `fuzzyNot` as the difference to one. It is worth noting that operators can be combined passing operator objects (instead of functions) as parameters to their constructors because of duck typing.

The code in Figure 11 shows the predicate and action functions used in the fuzzy rules defined. These functions were developed making use of primitives of two different *namespaces*. The first *namespace*, `Sonar`, implements functions to control the sonar sensors in the robotic platform. `Sonar` was used in our prototype to measure distance to objects in order to avoid obstacles. The robotic platform has 15 sonar sensors around its base. By retrieving information from the appropriate sonar sensors it can be known whether there are any obstacles, and where they are. The `Sonar` *namespace* provides the `getFromDevice` function that receives an integer as an argument with the number of the sonar sensor to be queried. The second *namespace* used is `Navigation` that provides positioning functions such as `goTo` (to move to specific coordinates) or `setSpeed` (to accelerate or decelerate the robot).

4.1.4. Event Management

The adaptive dynamic layer permits 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 added at runtime, and programmers can dynamically access them.

Following with our example, let's suppose that we have an artificial vision primary module capable of detecting the face of a person. At the moment the face is recognized, the artificial vision primary module triggers an event with the data of the face. Figure 12 shows an example subscription to a concrete event in the adaptive dynamic programming layer:

1. Authentication is performed in the first place. Afterwards, prioritization is established to the maximum, and asynchronous execution mode is chosen. With these settings, the *Greet* task can be executed in parallel with a navigation task that may be running in the framework. If the robot is performing any other task at the moment the event is triggered (e.g., moving towards any point), it will be able to look to the face detected and greet at the same time, without having to stop its navigation.

DOI: 10.1002/spe

```
sonar = Sonar()
navigation = Navigation()
minFrontDist = 1
speed = 0.600
turnRate = 10
def rightObstacle():
    return (sonar.getFromDevice(6) < 2*minFrontDist) |
           (sonar.getFromDevice(7) < 2*minFrontDist) |
           (sonar.getFromDevice(8) < 2*minFrontDist)
def leftObstacle():
    return (sonar.getFromDevice(0) < 2*minFrontDist) |
           (sonar.getFromDevice(1) < 2*minFrontDist) |
           (sonar.getFromDevice(15) < 2*minFrontDist)
def fuzzyLeft():
    return ( sonar.getFromDevice(0) +
             sonar.getFromDevice(1) ) /10
def fuzzyRight():
    return (sonar.getFromDevice(6) + \
            sonar.getFromDevice(7))/10
def isFrontLeftMoreFree():
    return fuzzyLeft() > fuzzyRight()
def isFrontRightMoreFree():
    return not isFrontLeftMoreFree()
def obstacleClose():
    return sonar.isDistanceToObstacleLessThan(\
                minFrontDist)
def moveUntilLeftIsFree():
    while leftObstacle():
        goOn()
def moveUntilRightIsFree():
    while rightObstacle():
        goOn()
def goOn():
    navigation.setSpeed(speed, 0)
def turnLeft():
    while obstacleClose():
        navigation.setSpeed(0, turnRate)
def turnRight():
    while obstacleClose():
        navigation.setSpeed(0, -1 * turnRate)
def goToTarget():
    global x, y
    navigation.goTo(x,y,0)
def isTargetReached():
    global x, y
    return (navigation.getDistanceFrom (x, y) <= 1)
```

Figure 11. Fuzzy helper functions.

```
# Authentication and Priorization
authenticate("admin", "defaultPassword")
setTaskType(RunnablePriority.MAX, ASYNCHRONOUS_TASK, "Greet")
# Event Handler
def handlerOfFaceDetectionEvent(data):
    speech = Speech()
    speech.say("How are you " + data.name + "?")
# Event Subscription
event = FaceDetectionEvent(handlerOfFaceDetectionEvent)
```

Figure 12. Face detection event.

2. Afterwards, 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 receives a text and reproduces it by simulating a human voice, greeting the recognized person.
3. The last statement subscribes the previous function to the face detection event. Therefore, when the event is thrown, the `handlerOfFaceDetectionEvent` function will be automatically called.

We have used a double *Observer* design pattern in the design of the event management subsystem [17]. First, the framework acts as the listener of the events thrown by all the primary modules. Second, the service modules act as subscribers of events triggered by the framework. In Java, when a service module requires a subscription to a concrete event, it must provide a class implementing the `Observer` interface supplied by the framework. This interface forces to implement the method that the framework will use to notify the service module when the event is triggered. Therefore, it is necessary to create a Java class, when the real purpose is to implement a single method. This is not necessary in Python because of duck typing.

In order to undertake the goals of event management, generative programming and computational reflection were used. By applying generative programming and structural reflection, each event in the framework corresponds to a generated Python class. These classes implement the `Observer` interface 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.

DOI: 10.1002/spe

Due to the dynamic discovery and the 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, simultaneous tasks can also be executed, making the most of the prioritization and synchronization features provided by the framework.

4.1.5. Remote Reprogramming

Until this point, we have explained how the structure and behavior of the generated code that wraps the primitives and events improves code simplicity and dynamic adaptability. In this section, we describe how to make use of these features using the remote reprogramming engine provided by the framework. The remote hot-reprogramming subsystem allows the introduction of new behavior guidelines at runtime, not foreseen at design time, which might run in parallel with existing tasks. This is a valuable feature to facilitate the implementation of runtime adaptable services.

Dynamic languages provide interesting features to be used for runtime adaptation and hot reprogramming. Introspection can be used to know the structure of each component. Intercession and dynamic code generation are two valuable tools to adapt running components. As an example, a running application can be adapted by replacing the code of one of its methods (structural intercession) with a new source code implementation (dynamic code generation and evaluation).

Applications developed in a dynamic language can use the adaptive layer to offer a higher level of abstraction. Since their code is executed by the Java virtual machine, it can directly access the services in the framework. For instance, the authentication and authorization services used in hot reprogramming are those offered by the framework (Figure 5). Moreover, the execution of the source code is performed in a controlled way: exceptions thrown by syntactic and semantic errors are dynamically handled by the framework, aborting the execution of the invalid programs.

The higher level of abstraction commonly offered by dynamic languages, plus the flexibility of their dynamic type system commonly involve reducing the source code required to program new services. In addition, since they are not compiled statically, they facilitate interactive hot reprogramming. They can also be used as scripting languages to "glue" together existing components and services, by simply describing the logic of a service that orchestrates existing functionalities.

The framework exposes one Web service for the addition and deletion of service modules from a remote system, which, after authorization, can send the source code to the framework execution engine. The code received is added to the task queue, being executed according to its priority and synchronization settings. Following with our example, a *Reminder* service module that reminds patients to have their medicine can be added at runtime by means of the remote reprogramming feature of the framework. Figure 13 shows the source code we have used to develop the reminder scenario.

1. This module imports the `FuzzyNavigation` component shown in Section 4.1.3 and authentication in the framework is then performed.
2. Next, it is specified that the task is a `TRANSACTIONAL_TASK`. This execution mode allows obtaining the execution control and avoids other synchronous tasks to interrupt it, notwithstanding the parallel execution of other asynchronous tasks. Therefore, the *Reminder* module could be loaded and executed at runtime, while the *Greet* asynchronous task (Section 4.1.4) is being executed in parallel. It could happen that, when the robot is going to remind someone to take his or her medicines, a face is detected and, without interrupting its navigation task, the robot greets the person (at the same time it is moving around).
3. An instance of the `Speech` *namespace* is created.
4. The `patientLocatorWS` reference points to a Web service that provides the coordinates of patients (in Section 4.1.6 we detail how Web services are implemented in the adaptive programming layer).
5. Then, a `getPosition` function that retrieves the coordinates of a patient passing his or her identification is defined, returning a tuple with the `x` and `y` values (the positioning mechanism is explained in Section 4.1.6).

DOI: 10.1002/spe

```
from FuzzyNavigation import *
# Framework Initialization
authenticate("admin", "password")
setTaskType(RunnablePriority.ONE,TRANSACTIONAL_TASK,"Reminder")
# Global variables
speech = Speech()
patientLocatorWS = WebService(WSDL_URL)
PATIENT_ID = "1"
fuzzyNavigation = FuzzyNavigation()
def getPosition(id):
    position = patientLocatorWS.getPosition(PatientId = id)
    return position.getX(), position.getY()
# Main program
x, y = getPosition(PATIENT_ID)
while not isTargetReached():
    fuzzyNavigation.evaluateFuzzyRules(x, y)
    x, y = getPosition(PATIENT_ID)
speech.say("It is time to have your medicine")
end()
```

Figure 13. Reminder module.

6. The main program contains a loop that iterates while the target is not reached, performing one more step through the `FuzzyNavigation` component.
7. Once the loop ends, the robot is situated near the patient. The robot informs the patient by voice that it is time to have his or her medicine, using the `say` primitive.
8. Finally, the framework resources engaged by the transactional task are released by the `end` function.

The reprogramming process is initialized by the remote system, which requests the framework for an authorization credential. Then, the remote system passes the application source code like the one in Figure 13. The framework runs the code performing the dynamic authorization checks. The *Greet* asynchronous module shown in Section 4.1.4 will be executed in parallel (asynchronously). Thus, if the robot bumps into someone while it is heading the patient, it would say hello to him or her without interrupting its navigation—the *Greet* module is asynchronous.

This hot reprogramming system allows the use of runtime emerging primitives, components and events, fulfilling the adaptability requirements of runtime adaptable systems; not only discovering services at runtime, but also adding new programs dynamically. It allows implementing services that fulfill new requirements emerging at runtime (e.g., the *Reminder* module) without stopping the system execution. Reprogramming could be carried out by a person, system or device, implying an autonomously readjustment of the robot. Computational reflection and generative programming have significantly facilitated its implementation.

4.1.6. Remote Bidirectional Communication

The *Reminder* module can be added at runtime as a service module, so that the robot locates the position of the patient, goes to the position obtained, and performs some task (like providing speech reminder to the patient). Furthermore, in our scenario patients could be moving, so the robot has to adapt its destination while avoiding any obstacles found in its path. The patient location device would be any (indoor) positioning system that patients wear. It is accessed through a Web service whose URL is initially unknown by the robot. We simulate the patient location device with a simple graphical application that allows us to simulate the change of the patient position at runtime. The coordinates of patients are retrieved from a Web service containing their location. This service could be part of an application over the network, or even the remote application that performed the dynamic reprogramming of the *Reminder* service. In that case, bidirectional communication

DOI: 10.1002/spe

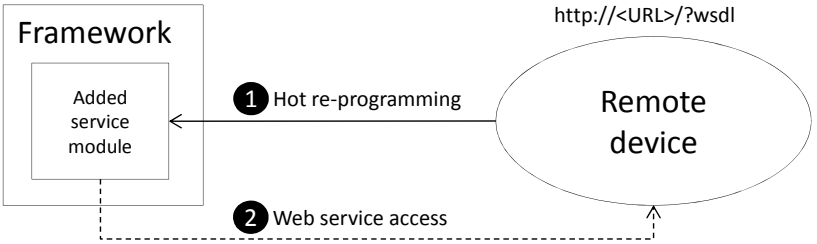

Figure 14. Bidirectional hot reprogramming.

has to be provided: in the first direction, the remote application adds the reminder code in the framework; in the second one, the inserted code asks the remote system for the patient coordinates. This bidirectional communication is shown in Figure 14. Notice that this way of bidirectional communication is more powerful than parameterizing a communication service, because it allows the incorporation of entirely new functionalities that had not been taken into consideration at design time.

The framework provides the necessary infrastructure to call new Web services discovered at runtime, exposing them as new functionalities grouped into *namespaces*. Generative programming is used to create proxy classes for accessing Web services operations. Apache Axis [34] is employed to dynamically generate the Java proxy classes. Axis is an implementation of the SOAP protocol provided by the Apache Software Foundation. The *wsdl2java* tool is used to generate Java code. This tool provides a set of classes and structures that act as proxies and skeletons of remote Web services described by WSDL documents. Operations described in WSDL files generate bean classes, representing parameters of operation as properties of each bean class. Making use of these standard technologies, any remote application, system or device that implements W3C standard Web services could be used in our robotics framework. Note that the service could be provided by an entity (person or device) while the robot is executing other applications.

The `WebService` object in Figure 13 (`patientLocatorWS`) shows how simple it is to add a reference to a new Web service using a dynamic language to program the adaptive layer. The `WebService` receives a WSDL URL and generates all the necessary code to access that Web service. The advantage is that dynamic languages facilitate the dynamic generation of code to transparently access services unknown at design time. This means that the value of `WSDL_URL` may be unknown statically because the code pointing to that WSDL is created dynamically (hot reprogramming). Through the use of generative programming and structural reflection, a Python class with the skeleton code shown in Figure 2 is generated. The `WebService` constructor returns an instance of the generated class. As shown in Figure 13, the operations in a Web service are simply offered as nonexistent methods using behavioral reflection.

The main issue at this step came from the Java code generation, where operations in Web services are wrapped by Java bean classes. Since parameters are modeled as properties of the Java beans, the order of parameters in each method is lost. Therefore, when it is required to call an operation in a Web service, we should first create an instance of the Java bean that represents that operation. Then, the invocation to the appropriate *setter* methods should be done; one invocation for each parameter. Although we should not follow the specific order defined in the WSDL, we have to make sure that suitable values are assigned to each single property that represents a parameter of the operation.

To solve this problem, we generate Python code that makes use of the named parameters feature of Python, obtaining very simple code. The third argument of the Python `__call__` method `(**kwargs)` is a dictionary structure that contains named parameters received in the invocation. Names of *setter* Java methods are generated from names of Python parameters and searched in the Web service class through introspection. Once generated, the operations are invoked with the value of the parameter received in Python (e.g., `PatientId` in Figure 13).

Generative programming and computational reflection were used to dynamically discover new Web services. This way, any application in the network that implements a standard Web service could be dynamically discovered and called, facilitating the implementation of runtime adaptable

DOI: 10.1002/spe

	Number of Executions			
	1	10	100	1,000
Java	237	2,178	21,647	216,512
Python	387	2,443	23,549	233,054
% Python slower	63.29%	12.17%	8.79%	7.64%

Table I. Execution time (in milliseconds) of source code in Figures 4 and 6.

services in our robotics framework. A high level of simplicity is offered to program the adaptive dynamic layer, requiring only one line of code to create a new Web service object. Method invocations are dynamically interpreted as operation calls to the remote Web service by means of behavioral reflection.

Currently, we are using introspection to inspect the syntactic description of Web services. Future research will be aimed at consulting semantic information when semantic Web services are used instead. This type of services specifies not only its syntactic information but also the meaning of its messages. Semantic Web services use standards for the interchange of semantic information, which facilitates the communication between service producers and consumers. A common way to specify the semantics of a Web service is using OWL (Web Ontology Language) [35] to describe ontologies. If OWL were used to describe the semantics of remote devices or systems, the robot could use a platform for the discovering of service semantics, such as SSWAP (Simple Semantic Web Architecture Protocol) [36], to know when a remote service is valid for its purposes. This would represent an important progress in the development of context-aware robotic systems.

4.2. Runtime Performance and Memory Consumption Assessment

Service modules and components coded in Java are directly executed by the framework. However, when a dynamic language such as Python is used, a runtime performance penalty could be introduced. In that case, the framework creates a Java module that wraps the Python implementation, loads the appropriate language engine of the Java Scripting API, and executes the Python application. This process causes a runtime performance penalty to be evaluated.

We have evaluated the runtime performance of the Java task shown in Figure 4 and its corresponding translation to Python displayed in Figure 6—the invocation to `Thread.sleep` in Figure 4 has been removed to make it run faster. We have run both tasks instrumenting the code with hooks to measure execution time, recording the value of the processor time stamp counter at the beginning and at the end of the task. This first evaluation measures the execution time of each task, without considering the above mentioned penalty of loading the Scripting API.

The first column in Table I shows the execution time of both tasks. In this case, Python is 63.29% slower than Java. The following columns display the same values incrementing the number of executions of the task. The code has been placed in a loop of 1, 10, 100 and 1,000 iterations. This common method of evaluation makes the Java Virtual Machine (JVM) reach a *steady state* [37], causing hot-spot optimizations to be applied at runtime. Incrementing the number of iterations, the performance penalty decreases to 12.17%, 8.79% and 7.64% running 10, 100 and 1,000 iterations, respectively. This performance reduction may be caused by the hot-spot optimizations introduced to the JVM when executing reflective code [38]. The Java Scripting API executes Python code with the Jython runtime, which makes wide use of Java reflection [39].

We have also evaluated the whole process of running a Python service module, including the dynamic loading of the Java Scripting API. In this case, the execution time of any task is increased with a constant value of 1,686 milliseconds. This cost is a constant penalty per Python service module.

We have measured memory consumption using the `Runtime` class of the Java standard library. The framework requires 2.54 MBs, while the execution of the Java module shown in Figure 4 uses 97.25KBs. In the case of Python, the Java Scripting API requires 6.4 MBs of memory including the Java execution engine of Python (i.e., Jython). This value remains constant independently of the number of Python services executed on the framework. The Python service displayed in Figure 6

DOI: 10.1002/spe

uses 750 KBs of memory. We think this difference may be caused by the entities used by Jython to simulate the Python computational model on the JVM, plus the classes we generate at runtime to provide transparent access to primary modules (Section 4.1.2).

5. COMPARISON WITH RELATED WORK

The objective of this section is twofold. First, to analyze the existing related work; second, to compare the fulfillment of the requirements we have identified at the beginning of this paper (Section 1.1). The objective of this comparison is not to say which approach is best, but to compare the suitability of existing approaches for hot reprogramming a robotics framework (the main contribution of this paper). This is the reason why some important issues such as runtime performance, discussed in Section 4.2, are not discussed here.

We have qualitatively assessed our TIC4BOT system together with some representative alternatives for creating highly adaptable robotic systems. The evaluation has been done based on the literature survey. We first analyze existing service-oriented component-based software systems (Section 5.1). Then, we examine those systems that employ dynamic languages for programming runtime adaptable robotics services (Section 5.2). Finally, we discuss other approaches that implement robotic systems providing a high level of runtime adaptability (Section 5.3).

The features we have used in the comparison are those we think a highly adaptable robotic system should fulfill, explained in Section 1.1:

1. Dynamic hot reprogramming of the system.
2. Homogeneous discovering of dynamic services.
3. Interactive, compact and straightforward programming of services.
4. Discovering and interaction of running applications.
5. Standards-based bidirectional communications between the system and dynamically discovered remote devices / systems.
6. Dynamic adaptation of existing services.
7. Language neutrality.
8. Non-proprietary standard language.
9. Visual domain-specific language.

We have added the three last requirements that, although they are not related to runtime adaptability, we think they facilitate the task of programming. Feature 7, language neutrality, is concerned with the possibility of developing robotics services in any programming language. Programmers may select the language they are more comfortable with, increasing their productivity. TIC4BOT fulfills this requirement as a result of employing the Java Scripting API. Feature 8, non-proprietary standard language, is related to the use of existing standard languages, preferably widely-used in the development of software. As we have mentioned in this paper, we think this feature involves the reutilization of existing source code and components, plus better accessible documentation. Finally, the visual domain-specific language requirement (feature 9) is focused on allowing the runtime programming of new services in a visual language that could be understood by non-programmers. Although we are currently working on this requirement (see Section 6), our system does not provide this feature yet.

Table II summarizes the analysis made. Systems have been analyzed from the point of view of the framework—not from the point of view of the systems built on the framework. Each column corresponds with each feature. The two following subsections explain in more detail the analysis of each system.

5.1. *Service-Oriented Component-Based Software Systems*

Service-oriented component-based systems allow for dynamic rewiring of services and replacing of components and services at runtime. They provide parameterization at runtime and task coordination

DOI: 10.1002/spe

functionality. Component-based service-oriented software systems are broadly used in robotics, providing a high level of runtime adaptability.

Zhang et al. studied a context-aware intelligent robot control system [40]. They defined a service-based architecture to break the tight coupling between components, providing a higher abstraction layer by means of services. They used the Jini middleware [41] to federate groups of devices and components into a single, dynamic and distributed robot service coordination system. The use of Jini provided the dynamic discovery of services, including robot, sensors and other services. One important feature of the proposed system is that components can be added and removed at runtime [40]. When services or robots are plugged into the system, they can be directly used by clients and other services. It is worth noting that Jini defines a programming model which exploits and extends Java technology to enable the construction of secure distributed systems, consisting of federations of well-behaved network services and clients. This way, the entire system is based on Java.

Orca is an open-source software framework for developing component-based robotic systems [42]. It intends to define and develop building-blocks that can be pieced together to form arbitrary complex systems. The main goal is to take advantage of software reuse in robotics. Orca relies on the Ice (Internet Communications Engine) technology, which is a proprietary middleware that is used for defining contracts between servers and clients, and compiles it into some programming languages. It aims to be as broadly applicable as possible and that is the reason why authors did not make assumptions about component granularity, system architecture, the set of provided or required interfaces, and the internal architecture of the components. As a result, Orca does not offer full native support for programming runtime adaptable services.

Fawkes is a robotics framework which follows the component-based software design paradigm by featuring components with communication interfaces [43]. It includes components for timing, logging, data visualization, software configuration, or decision making. The main features it has are: 1) a well-defined component concept; 2) a hybrid blackboard / messaging infrastructure for communication; 3) well-defined interfaces; 4) runtime loadable plug-in mechanisms; 5) multi-core computation facilities; and 6) a network infrastructure for communicating with remote software entities. Their goal is to design a framework to be highly flexible and portable to various robot platforms, adding new functionality over time. The components are implemented as dynamically loadable libraries, implementing a particular interface which gives access to descriptive and dependency information. The Fawkes core provides a blackboard that serves as the centralized storage unit. Services can be discovered at run-time by listing the blackboard interfaces, which can then be instrospected with a specialized API. The changes on the behavior of programs are automatically reloaded, facilitating interactive and straightforward programming of services.

ROS is a framework for robot software development, providing operating system-like functionality on the top of a heterogeneous computer cluster [9]. It supports services such as hardware abstraction, low-level device control, message-passing between processes, and package management. ROS has been designed to be language-neutral, currently supporting C++, Python, mainly, and Octave and Lisp [9]. Python has been heavily used to program different robots such as the Personal Robot 2 (PR2), producing less verbose applications than C++ [44]. Rather than providing a C-based implementation with stub interfaces generated for the offered languages, ROS has been implemented natively in each target language. To support cross-language development, ROS requires the use of a language-neutral IDL (Interface Definition Language) to describe the messages sent between modules. The IDL uses text files to describe fields of each message.

SmartSoft is a component-based robotics software system based on communication patterns [22]. Its component model includes standardized ports for component interaction and configuration during runtime as well as an internal state automaton. The semantics of the interface of components is predefined by the communication patterns, independently of the underlying implementation technologies. This allows the separation between component internals and the externally visible component interface. *Dynamic wiring* of components at runtime is explicitly supported by the wiring pattern. The benefit is the implementation of loosely coupled and distributed systems based on standardized components whose interaction can be adjusted according to the current context and

DOI: 10.1002/spe

Approach	Fea.1	Fea.2	Fea.3	Fea.4	Fea.5	Fea.6	Fea.7	Fea.8	Fea.9
CAMUS	○	○	◗	●	○	○	◗	◗	◗
DCD-VM	○	○	○	○	○	○	○	◗	●
Fawkes	◗	●	●	●	○	●	○	●	○
Lacey et al.	○	○	○	○	○	○	○	●	○
Nao / HSM	◗	●	●	●	○	●	○	●	●
Orca	◗	●	●	●	○	●	◗	◗	○
ORCOS	◗	●	○	●	○	●	●	●	○
Pyro	●	○	●	●	○	●	○	●	○
RoboEarth	●	●	◗	●	◗	●	●	○	◗
Rönning et al.	○	○	○	○	○	○	○	●	○
ROS / SMACH	◗	●	●	●	○	●	◗	●	●
SmartSoft / TCL	●	●	●	●	◗	●	○	◗	○
TIC4BOT	●	●	●	●	●	●	●	●	○
Urbi	◗	○	●	●	○	◗	○	◗	◗
Zhang et al.	◗	●	○	●	◗	●	◗	●	○

Table II. Highly adaptable requirements fulfillment of related work.

requirements [45]. The sequencing layer of SmartSoft is implemented using the SmartTCL dynamic language [46], which is an extension of Lisp (see Section 5.2).

The OROCOS (Open RObot COntrol Software) project aim is to develop a general-purpose, free software, and modular framework for robot and machine control [47]. The OROCOS framework is developed in C++ under an open source license, and provides high modularity and flexibility because it is based on components. Components are software objects that can be dynamically added and removed from a network and offer their services through a neutral, programming language independent, CORBA interface. There are five distinct ways in which an OROCOS component can be interfaced: through its properties, events, methods, commands and data flow ports. OROCOS does not impose a specific programming language. The philosophy behind this strategy is that the best language for a job should be used, including scripting languages [47].

The difference between these service-oriented component-based systems and the approach presented in this paper is the mechanism used to obtain dynamic adaptability. In these systems, components (and services) may be added, replaced, removed and rewired at runtime to provide runtime adaptability. This adaptation is commonly performed by a middleware or framework. Dynamic languages also provide these features, but they are controlled by the dynamic language itself instead of by a middleware. The addition of new components is obtained with dynamic code evaluation—one component may even create another component at runtime. Inspection of new components is provided by the introspection services of the language, so no language for defining component interfaces (e.g., CORBA IDL) is needed. The adaptation of existing components is obtained with structural intercession, modifying the structure of components at runtime. Rewiring is achieved with behavioral intercession by changing semantics of the message passing mechanism. These operations are controlled by the interpreter of the dynamic language, because these reflective features are part of its computation model. The examples presented in this paper have also shown how dynamic languages facilitate the interactive, compact and straightforward programming of components and services, with lower runtime performance and compile-time error detection.

5.2. Dynamic Languages in Robotic Systems

Dynamic languages have been previously used in different robotics scenarios. An example is the SmartTCL language [46], an extension of Lisp that was used to implement the sequencing layer of SmartSoft, a three layer robotic architecture (Section 5.1). The sequencing layer is the place to

DOI: 10.1002/spe

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 easily modified at runtime.

The dynamic language Lua has also been used to implement the behavior engine of the humanoid robot *Nao* [48] over the Fawkes component-based framework [43]. 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.

The SMACH (State MACHine) library is a scalable Python-based library for hierarchical state machines that, the same as SmartTCL and Lua, is independent of the system in which it is used [49]. It allows designing, maintaining and debugging large and complex state machines. Robotics frameworks such as ROS (Robot Operating System) [9] have been accurately integrated with SMACH.

UrbiScript [10] is an orchestration language for robotic systems, used to glue together C++ components developed for the Urbi (Universal Robot Body Interface) robotic platform [50]. Usually, the CPU-intensive algorithms are implemented in C++ and the behavior part is left to UrbiScript (a proprietary programming language) because it is more flexible and easier to maintain. One important feature of UrbiScript is that it allows dynamic interaction during program execution, and provides a simple event-based programming as part of the language semantics. The use of C++ to develop components (known as UObjects in Urbi terminology) limits the dynamic adaptation of their structure, even though a dynamic language is used to glue them together.

Pyro (Python Robotics) is 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 [8]. Any program that controls a robot (physical or simulated) is referred to as a *brain*. Since *brains* are written in Python, they can use all the meta-programming features of that language to implement runtime adaptable services. Although the system is written in Python, it does not exploit the Python meta-programming features to provide advanced mechanisms such as bidirectional standard communication between the system and runtime discovered remote entities.

The common approach of these systems is using dynamic languages to orchestrate components and services. Components are commonly developed in compiled languages such as C++, while the coordination / behavior part is programmed in a dynamic language because it is more flexible and easier to maintain. Therefore, these systems allow modifying the coordination tasks at runtime and interacting with the user during program execution. Our approach is similar to these systems in the sense that components can be developed in Java, and the coordination logic can be defined in any dynamic language supported by the Java Scripting API. The main difference is that services and components can also be developed in a dynamic language, and executed by the framework. Using Python as a case study, we have shown how this approach facilitates exploiting the particular features of dynamic languages in the development of adaptable services. For instance, we have used the dynamic code generation and the behavioral intercession features of Python to provide a service that dynamically creates proxy classes for accessing Web services (Figure 13). The interface of Web services can be discovered at runtime and it can even be changed dynamically because the proxy classes are dynamically generated. The meta-programming features of Python have simplified the implementation of this service and its use.

Although the particular features of general purpose dynamic languages can be positively used in the development of robotic systems, visual domain-specific languages also provide notable benefits (feature 9 in Table II). The first advantage is that these languages can be understood by non-programmers, facilitating the dynamic adaptation and interaction with the user. Moreover, its domain-specific approach allows the declarative specification of how programs should behave, making programs more compact and maintainable.

DOI: 10.1002/spe

5.3. *Other Approaches to Build Runtime Adaptable Robotic Systems*

In [51], the authors introduce the concept of network-based intelligent robots called URC (Ubiquitous Robotic Companion). URC are robots that distribute their functional components through the network, using external sensors and remote processing servers. They propose the use of the CAMUS (Context-Aware Middleware for URC System) software framework as a system middleware to support context-aware services for network-based robots. CAMUS acquires, interprets and disseminates context information, offering the means for modeling the environment in which the robot provides services. CAMUS is composed of four parts: the main server for collecting contextual information; the service agent for executing the functions of legacy applications and sensors; the service agent manager for controlling agent managers within the environment; and a communication framework for handling long-term operations of robots. It has been developed to support programming languages such as Java and C++. They also define the PLUE (Programming Language for Ubiquitous Environment) to describe context-aware services for robots. PLUE is a Java extension that adds rule-based reasoning to Java using the Jess rule engine [52]. Since the PLUE compiler is implemented as a Java preprocessor, it is not easy to add new PLUE rules at runtime.

Röning and Riekki propose the architecture of a context-aware mobile robot system for managing and using services on behalf of the user [53]. Context-aware mobile robots perceive environmental signals, infer the context (i.e., the state of the system and its local environment) from these signals, and calculate appropriate actions for the detected context. The *Genie of the Net* architecture is composed of four agents: sensor, user interface, active user and context agent. The system collects information from sensors and databases, recognizes context based on this information, chooses relevant actions to serve the user on the basis of the recognized context, and performs the chosen actions. The actions to be performed by the robot can only be reprogrammed at runtime when they are foreseen in advance (prior to execution). If a new functionality is required while the robot is running, after compiling the source code of the new service, it is necessary to stop its execution to deploy the new functionality.

Lacey and MacNamara developed a smart walker mobile robot designed to provide runtime adaptable shared control of a robot mobility assistance to the frail elderly visually impaired [5]. They used a Bayesian network to provide context-aware shared control of the robot. The system is divided into five predetermined modules: 1) risk assessment for collision avoidance; 2) user interface for user input and audio feedback; 3) feature extraction for corridor and door features; 4) navigation for walking; and 5) action selection for setting the goal points for the navigation module. They combined user input with sensor data to effect context-aware goal selection for a mobile service robot. The user goals were modeled as goal directions to the navigation system. The probability of these goals is influenced by the probabilities in their parent nodes. These probabilities are fused to produce an estimate of the user goals (i.e., what robot action the user wants to take next).

Bordignon et al. [54] propose a solution for updating individual modules in a robotics framework based on a virtual machine design. The virtual machine (called DCD-VM) is programmed in a high-level role-oriented domain-specific language that allows the declarative specification of how programs should be deployed (low level code is programmed in the C programming language). That way, the solution enables fast and incremental dynamic updates without needing to restart the system. Their work focuses on the ATRON modular robot [55], which is a cheap and highly resource-constrained hardware designed to be cost-effective. For its part, DCD-VM enables efficient distribution of small bytecode programs, supporting live update of controller code in running modules—module interfaces cannot be modified at runtime.

RoboEarth is a World Wide Web for robots, representing a distributed database repository where robots share data to learn from each other about their behavior and environment [56]. RoboEarth is intended to be a Web community by robots for robots to autonomously share descriptions of tasks they have learned, object models they have created, and environments they have explored. It is implemented based on a three-layered architecture. The server layer stores the global world model including objects, environments and actions. This information is linked to semantic information (in OWL) and provided by means of Web services. The hardware-independent middle layer provides

DOI: 10.1002/spe

generic reusable components to allow any robot to interpret RoboEarth action recipes. These action recipes describe robot tasks in a human-readable high-level proprietary language [57]. The third layer implements skills and provides a generic interface to robot-specific hardware-dependent functionalities via a skill abstraction layer.

The main similarity between RoboEarth and the system presented in this paper is the publication of robotic components as Web services, where the last version of components is provided due to their dynamic inspection. We perform this inspection by means of introspection, whereas RoboEarth implements a distributed repository of component interfaces. Language neutrality of RoboEarth components is achieved by the definition of a high-level proprietary language [57] that should be translated into the particular language used in each robotic platform. On the other hand, our framework employs the Java Scripting API to execute any dynamic language over the Java platform. A distinguishing feature of RoboEarth is the semantic information stored to describe the components. This information helps robots to dynamically determine which components can be safely used for a specific purpose.

6. CONCLUSIONS AND FUTURE WORK

An example scenario has been developed to discuss how the distinguishing meta-programming features of dynamic languages can be used to build highly adaptable hot-reprogramming services in a robotics framework. The use of Python has allowed us to develop an adaptive dynamic programming layer to create highly runtime-adaptable services. This layer offers dynamic runtime discovery of new services and provides a transparent programmatic access. The framework permits the programming of services in Java when runtime adaptability is not a necessary issue, offsetting the runtime performance penalty commonly shown by the use of dynamic languages. In our example scenario, we have used the SCITOS-G5 platform. The robot processing capabilities have greatly sufficed the runtime performance requirements.

The use of different levels of computational reflection and generative programming has provided the desired degree of runtime adaptability. Introspection has been used to dynamically discover the services published by both the framework and external devices. We have applied generative programming to generate wrapper Python classes around these services. Generated classes are extended by means of structural intercession. The semantics of the message passing mechanism has been modified (by means of behavioral intercession) to implement a lazy method search in the generated classes. Dynamic duck typing system facilitates the interactive hot reprogramming of services, and makes the source code more compact and simpler. All these particular features of dynamic languages have been used to develop a dynamic adaptive layer that facilitates the development of highly runtime adaptable services.

We are currently working on the development of a visual programming language to facilitate the programming of the robot at runtime, making use of the hot re-programming subsystem. The visual-language compiler will generate Python code, sending it to the robot when the programmer wants to run it. With this approach, we will obtain the benefits of both a straightforward domain-specific visual language and a widely-used versatile general-purpose one.

In the future, we plan to investigate on the application of automatic machine learning [58], making the system able to learn the behavior of users on simulated scenarios. A runtime performance assessment comparing our framework with existing robotic platforms is another future activity to be done. Finally, services for other concrete scenarios, e.g. the assistance to the frail elderly visually impaired, will also be defined.

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 the source code of the dynamic adaptive dynamic programming layer, and a demonstrating video showing the example presented in this paper, running on the robotics framework with Player / Stage.

DOI: 10.1002/spe

ACKNOWLEDGEMENT

We would sincerely like to thank the anonymous reviewers for their detailed lists of indications, corrections and suggestions that have helped us to improve the article.

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation; project TIN2011-25978 entitled *Obtaining Adaptable, Robust and Efficient Software by including Structural Reflection to Statically Typed Programming Languages*. It has also been funded by the Spanish Ministry of Industry, Tourism and Commerce (TSI-020100-2008-235) with the project entitled *TIC4BOT – Research and Development in the application of Information and Communication Technologies in Social Robotics: Health sector*.

REFERENCES

1. Schilit BN, Theimer MM. Disseminating active map information to mobile hosts. *IEEE Network* 1994; **8**(5):22–32.
2. Schilit BN, Adams NI, Want R. Context-aware computing applications. *International Workshop on Mobile Computing Systems and Applications (WMCSA)*, IEEE Computer Society: Washington, DC, USA, 1994; 85–90.
3. Eng K, Douglas R, Verschure P. An interactive space that learns to influence human behavior. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans* Jan 2005; **35**(1):66–77.
4. Pineau J, Montemerlo M, Pollack M, Roy N. Towards robotic assistants in nursing homes: Challenges and results. *Robotics and Autonomous Systems* 2003; **42**:271–281.
5. Lacey G, MacNamara S. Context-aware shared control of a robot mobility aid for the elderly blind. *International Journal of Robotic Research* 2000; **19**(11):1054–1065.
6. Maes P. Computational reflection. PhD Thesis, Vrije Universiteit Brussel 1987.
7. Lei H. Context awareness: a practitioner's perspective. *International Workshop on Ubiquitous Data Management (UDM)*, IEEE Computer Society: Washington, DC, USA, 2005; 43–52.
8. Blank D, Kumar D, Meeden L, Yanco H. Pyro: A python-based versatile programming environment for teaching robotics. *Journal on Educational Resources in Computing* 2003; **3**(4):1.
9. Quigley M, Conley K, Gerkey BP, Faust J, Foote T, Leibs J, Wheeler R, Ng AY. ROS: an open-source robot operating system. *ICRA Workshop on Open Source Software*, 2009.
10. UrbiScript. Urbiscript user manual. Web 2011. URL http://www.gostai.com/downloads/urbi-sdk/2.x/doc/urbi-sdk.htmldir/urbiscript-user-manual.html.
11. Orebäck A, Christensen HI. Evaluation of architectures for mobile robotics. *Autonomous Robots* 2003; **14**:33–49.
12. Ortin F, Redondo JM, Perez-Schofield JBG, Garcia M. Including both static and dynamic typing in the same programming language. *IET Software* 2011; **4**(4):268–282.
13. Ortin F, Lopez B, Perez-Schofield J. Separating adaptable persistence attributes through computational reflection. *Software, IEEE* 2004; **21**(6):41–49.
14. Ortin F, Cueva JM. Dynamic adaptation of application aspects. *Journal of Systems and Software* May 2004; **71**(3):229–243.
15. Ortin F, Diez D. Designing an adaptable heterogeneous abstract machine by means of reflection. *Information and Software Technology* Feb 2005; **47**(2):81–94.
16. Ortin F. A flexible programming computational system developed over a non-restrictive reflective abstract machine. PhD Thesis, Computer Science Department of the University of Oviedo 2002.
17. Gamma E, Helm R, Johnson R, Vlissides J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman, 1995.
18. Krzysztof C, Eisenecker U. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
19. Garcia MA, Gonzalez I, Suarez P, Barranquero J, Mendez S, Garcia-Diaz V, Perez S, Garcia H, Rodriguez T, Martin S, *et al.*. TIC4BOT: A research robotic software and hardware platform for reflective HRI, navigation and vision fields. *International Conference on Artificial Intelligence (IC-AI)*, 2009.
20. Treelogic. TIC4BOT project. Web 2011. URL http://idi.treelogic.com/esp/proyectos/proyecto10.html.
21. Gerkey BP, Vaughan RT, Howard A. The player/stage project: Tools for multi-robot and distributed sensor systems. *11th International Conference on Advanced Robotics*, 2003; 317–323.
22. Schlegel C, Worz R. The software framework SMARTSOFT for implementing sensorimotor systems. *IEEE/RSJ International Conference on Intelligent Robots and Systems(IROS)*, vol. 3, 1999; 1610–1616.
23. Garcia-Diaz V, Mendez S, Barranquero J, Gonzalez I, Garcia MA, Cueva JM. RIF: A reflective integrator framework. *International Conference on Artificial Intelligence (IC-AI)*, 2009.
24. Kazi IH, Chen HH, Stanley B, Lilja DJ. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys* 2000; **32**(3):213–240.
25. Oracle. Java authentication and authorization service. Web 2002. URL http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html.
26. Beazley DM. SWIG: an easy to use tool for integrating scripting languages with C and C++. *4th conference on USENIX Tcl/Tk Workshop (TCLTK)*, USENIX Association: Berkeley, CA, USA, 1996; 15–15.
27. W3C. SOAP version 1.2 part 0: Primer (second edition). Web 2007. URL http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.
28. Grogan M. JSR 223. Scripting for the Java Platform December 2006. URL http://www.jcp.org/en/jsr/detail?id=223.
29. Van Rossum G, Drake Jr F. *Python reference manual*. Centrum voor Wiskunde en Informatica (CWI), 1995.

DOI: 10.1002/spe

30. Parnas DL. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 1972; **15**(12):1053–1058.
31. Hursch WL, Lopes CV. Separation of concerns. *Technical Report*, College of Computer Science, Northeastern University 1995. URL http://eprints.kfupm.edu.sa/64610/.
32. Ortin F, Mendez S. TIC4BOT, on the suitability of dynamic languages for hot-reprogramming a robotics framework 2011; URL http://www.reflection.uniovi.es/tic4bot.
33. Tiobe. Tiobe programming community index. Web 2012. URL http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.
34. Apache. Apache Axis Project 2002. URL http://ws.apache.org/axis/.
35. W3C. OWL Web ontology language overview. Web January 2009. URL http://www.w3.org/TR/owl-features.
36. Gessler DD, Schiltz GS, May GD, Avraham S, Town CD, Grant D, Nelson RT. SSWAP: A simple semantic web architecture and protocol for semantic web services. *BMC Bioinformatics* 2009; **10**(309).
37. Georges A, Buytaert D, Eeckhout L. Statistically rigorous Java performance evaluation. *Object-Oriented Programming Systems and Applications*, OOPSLA '07, ACM: New York, NY, USA, 2007; 57–76.
38. Oracle. The Java HotSpot performance engine architecture, white paper 2012. URL http://java.sun.com/products/hotspot/whitepaper.html.
39. Ortin F, Redondo JM, Perez-Schofield JBG. Efficient virtual machine support of runtime structural reflection. *Science of Computer Programming* 2009; **74**(10):836–860.
40. Zhang P, Lee KK, Xu Y. Context-aware robot service coordination system. *2005 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2005; 410–415.
41. Edwards WK. *Core Jini*. Upper Saddle River. NJ: Prentice Hall., 2001.
42. Makarenko A, Brooks A, Kaupp T. Orca: Components for robotics. *Conference on Intelligent Robots* 2006; :163–168.
43. Niemueller T, Ferrein A, Beck D, Lakemeyer G. Design Principles of the Component-Based Robot Software Framework Fawkes. *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Lecture Notes in Computer Science, Springer: Darmstadt, Germany, 2010.
44. Boren J, Cousins S. The SMACH high-level executive. *Robotics & Automation Magazine* 2010; **17**(4):18–20.
45. Schlegel C. Communication Patterns as Key Towards Component-Based Robotics. *International Journal of Advanced Robotic Systems* Mar 2006; **3**(1):49–54.
46. Steck A, Schlegel C. SmartTCL: An execution language for conditional reactive task execution in a three layer architecture for service robots. *SIMPAR Workshop on Dynamic Languages for Robotic Sensor Systems*, Darmstadt, Germany, 2010.
47. Bruyninckx H. Open robot control software: the OROCOS project. *IEEE International Conference on Robotics and Automation*, vol. 3, IEEE, 2001; 2523–2528.
48. Niemueller T, Ferrein A, Lakemeyer G. A Lua-based Behavior Engine for Controlling the Humanoid Robot Nao. *RoboCup Symposium*, Graz, Austria, 2009.
49. Smach. The Smach package. Web Site 2012. URL http://www.ros.org/wiki/smach.
50. Urbi. Urbi forge. Web 2011. URL http://www.urbiforge.org.
51. Kim H, Cho YJ, Oh SR. CAMUS: a middleware supporting context-aware services for network-based robots. *IEEE Workshop on Advanced Robotics and its Social Impacts*, 2005; 237–242.
52. Friedman-Hill E. *Jess Manual*. Sandia National Laboratories, Livermore, CA, USA, 1997.
53. Röning J, Riekki J. Context-aware mobile systems for managing services. *SPIE Intelligent Robots and Computer Vision XX: Algorithms, Techniques, and Active Vision*, 2001; 504–512.
54. Bordignon M, Stoy K, Schultz U. A virtual machine-based approach for fast and flexible reprogramming of modular robots. *IEEE International Conference on Robotics and Automation* 2009; :4273–4280.
55. Jørgensen MW, Østergaard EH, Lund HH. Modular ATRON: Modules for a self-reconfigurable robot. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE Computer Society Press, 2004; 2068–2073.
56. Waibel M, Beetz M, Civera J, Andrea RD, Elfring J, Galvez-Lopez D, Haussermann K, Janssen R, Montiel JMM, Perzylo A, *et al.*. Roboearth - a world wide web for robots. *IEEE Robotics and Automation Magazine* June 2011; **18**(2):69–82.
57. Tenorth M, Perzylo AC, Lafrenz R, Beetz M. The RoboEarth language: Representing and Exchanging Knowledge about Actions, Objects, and Environments. *IEEE International Conference on Robotics and Automation (ICRA)*, St. Paul, MN, USA, 2012.
58. Mitchell T, Buchanan B, DeJong G, Dietterich T, Rosenbloom P, Waibel A. Machine learning. *Annual Review of Computer Science* 1990; **4**(1):417–433.

DOI: 10.1002/spe