<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>Lowering the bar for robotic development: driver generation for ubiquitous robotic systems</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Authors(s)</strong></td>
<td>Treanor, Jennifer; O’Hare, G. M. P. (Greg M. P.)</td>
</tr>
<tr>
<td><strong>Publication date</strong></td>
<td>2011-08-31</td>
</tr>
<tr>
<td><strong>Conference details</strong></td>
<td>Paper presented at the 22nd Irish Conference on Artificial Intelligence and Cognitive Science (AICS 2011), University of Ulster, Northern Ireland, 31 August - 2 September, 2011</td>
</tr>
<tr>
<td><strong>Publisher</strong></td>
<td>Intelligent Systems Research Centre</td>
</tr>
<tr>
<td><strong>Item record/more information</strong></td>
<td><a href="http://hdl.handle.net/10197/3452">http://hdl.handle.net/10197/3452</a></td>
</tr>
</tbody>
</table>
Lowering the Bar for Robotic Development:
Driver Generation for Ubiquitous Robotic Systems

Jennifer Treanor* and Gregory M.P. O’Hare

School of Computer Science and Informatics, University College Dublin
Belfield, Dublin, Ireland
{jennifer.treanor,gregory.ohare}@ucd.ie
http://robotics.ucd.ie

Abstract. Robotics has developed, technologically, to a level where it becomes a field of both interest and importance to other disciplines, either as a proof-of-concept or demonstrative tool, or else as the main focus for implementation of theories. This is particularly evident in the areas of computational and theoretical cognitive science where, despite this progress, robotics remains sufficiently inaccessible to non-specialists as to dissuade its use. This is due in no small part to the issue of code re-usability across differing hardware platforms and the lack of low-level support for developing suitable drivers for the main robotics development tools. To address this issue, this work presents ACorDE: Autonomous Control Development Environment. This development environment takes in data pertaining to the robotic platform and generates suitable driver and behavioural code in a standardised format.

1 Introduction

Code reuse has always been an important aspect of development throughout the computer sciences. While still applicable for software-only projects, once hardware is brought into the question, compatibility between implementations drops significantly, due to a number of reasons. Firstly the vast array of different hardware platforms and modules employed throughout the sciences makes standardisation implausible. Secondly, despite their respective impressive levels of development, disciplines such as AI, computation-based cognitive science, and in particular the broad field of robotics are still relatively young and arguably not yet developed enough to undergo standardisation [1].

The attempt to balance this difficult pivot point between a need for cross collaboration, intra- or inter-discipline, and a lack of applicable standards, has led to the introduction and subsequent popularity in the robotics community

* This work is supported by an IRCSET (Irish Research Council for Science, Engineering, and Technology) Embark Initiative Scholarship, and an SFI (Science Foundation Ireland) grant (grant number 07/CE/11147), under CLARITY: Centre for Sensor Web Technologies.
of Robotic Development Environments (RDEs), such as MARIE [1], or Robotic Development Frameworks (RDFs), such as ROS (Robot Operating System) [2] or Player [3]. These systems provide device abstractions so that generalised functional modules can be used across any generic applicable robotic platform.

Despite their innovation and versatility, RDEs still possess inefficiencies and barriers to their uptake, albeit more pronounced in some cases that others. In order to avail of the resources compatible with any one RDE, the device to be connected must be supported or suitable drivers must be developed. For many research projects, this is not of great consequence as, for the working platform, the drivers are written once and then essentially forgotten about. However, for research that requires rapid prototyping, frequent hardware reconfiguration, or those that focus on proof-of-concept and wizard-of-oz experiments, this can pose a significant difficulty when the steep learning curve associated with many RDEs is considered.

To address these issues, this work presents ACorDE, the Autonomic Control Development Environment, a layer of development involved in the code construction stage that takes in information about the hardware system and outputs the relevant driver code along with low-level behavioural and capability control code. The system implements a higher degree of abstraction through component and behavioural description templates, leaving the higher level architectural development to the user of the system. While work has been presented on the notion of reusing previously developed robotic tools [1], that work focuses on bringing together multiple RDF functionalities and other tools while still leaving the low-level development to the user. Instead, ACorDE allows for rapid development in, and easy access to, robotics by generating this low level linking code so that those higher level modules can be accessed on previously unsupported hardware.

2 Robotics in AI & Cognitive Science

Through the years, our understanding of cognitive processes has changed dramatically, from one centred around world modelling and complex problem-solving to one eliminating internal modelling as a factor altogether and back again to the notion of balanced middle ground combining aspects of each extreme [4]. This viewpoint was mirrored in AI, specifically through the various robotic control architecture structures from the Sense-Plan-Act (SPA) architecture [5], through [6] Subsumption architecture, and resting on hybrid architectures combining both reactive and deliberative elements, initially formulated via the Procedural Reasoning System (PRS) [7, 8]. Those studies grounded in cognitive science tend to be heavily focused on formalised theories, using robots as a later proof-of-concept tool to show the versatility of the approach taken. A well known example of this comes in the cognitive architecture applied to social robotics that controls the learning and interaction capabilities of the Leonardo social robotic platform [9]. This platform, while interacting via intuitive gestures, and facial expressions, learns via a range of modalities from voice tone and pitch to language understanding. The architecture does not specifically support autonomous mental de-
While robotic control architectures are typically designed to be independent of specific robotic platforms, their implementations, however, tends to be less so. This situation persists for a number of reasons. Firstly, most implementations are intended as proof-of-concept for the architectural theory and are not meant to be made directly available. Secondly, to develop these architectures through code to be as generic as possible is a significant undertaking few are thus far willing to commit to.

This is one aspect that this work intends to tackle, to provide a level of device abstraction greater than that currently offered by RDEs alone, that allows for the implementation of intelligent control in a sufficiently generalised way to be implemented on a vast array of platforms with minimal effort.

3 Robotic Development Software

Robotics development software comes in a number of varied tools. The most readily used are of a generic form termed Robotic Development Frameworks (RDFs) here, but in some cases referred to as Robotic Development Environments (RDEs), as coined by [11]. This distinction is dependent on the scope and target of the tool. A tool such as Player [3], that is a set of libraries and tools for the direct development of robot control code, is an RDF, where access to other toolkits tends to be implemented via a wrapper or driver. On the other hand, tools such as MARIE [1], that natively encapsulate other RDFs to provide access to all the varied functionalities of the different tools in one project, are closer to the traditional sense of a development environment, and hence more relevant to the title of RDE. As with all disciplines, however, there are always systems that sit on the fence between two or more titles. For example, Pyro (Python Robotics) [12] is a tool that follows the structure and ideals of Player, but also wraps much of Player’s functionality. It is written through Python and is targeted towards easing the teaching of robotics. This tool, depending on the angle, could be regarded as an RDF or loosely as an RDE. Specifically for its similarity to Player’s implementation, it is regarded here as an RDF. While in-depth categorization of robotic development tools is outside of the scope of this work, Table 1 provides a summary of the main points of interest of some of the more well known development tools.

Specifically, the table compares the total number of downloads to date (Downloads), the description given by the system’s developers (Classification), whether or not it provides for functional modules and how it supports them (Modules), what hardware it supports and whether it can be easily extended (Support & Extendibility), and finally the documented supported programming languages (Language). While all development tools of the RDF persuasion support some level of extensibility, integration of new hardware devices is typically not a straight-forward task, particularly for those less contented with hardware.
ACorDE aims to tackle this difficulty to allow for rapid development of thus far unsupported platforms for a range of RDFs.

## 4 Abstraction & Templating in ACorDE

This work is focused on a number of goals. The first is, as mentioned in Section 3, to allow for rapid development, and the second, building upon the first, is to enable the cross pollination of research ideas and findings across disciplines involving robotics. To achieve this, a further level of abstraction above that already achieved in typical RDFs is proposed. Abstraction here is implemented at four levels: the component level; the behavioural and capability level; driver level; and an architectural level.

### 4.1 Component Level Abstraction

The first, and perhaps most important, level of abstraction in ACorDE’s structure is at the component level. The theory is similar to that in traditional RDFs: a robot is a device composed of a number of components, typically some form of drive system (for mobile robots), a suite of sensing devices, and occasionally actuation and communication modules. Each of these categories encompasses a multitude of subcategories and options. For example, drive systems can be subdivided into legged and wheeled drive systems, among others. Considering wheeled devices, there are differentially driven, holonomic, car-like, killough driven and many more implementations. However, each of these low-level components can be described abstractly, taking in certain parameters such as wheel size, baseline distance, footprint, max and min speeds and accelerations, etc., into account. Functionalities wrap the hardware’s own APIs in a standard, supplying very low-level abilities such as `move_forward(distance)`, `turn_left()`, or `move_in_a_circle(radius)`. The component information is captured in an object template that is an aggregation of the appropriate number of functionalities and capabilities.

### 4.2 Behavioural Level Abstraction

Behavioural level abstraction follows the same theory that drives component level abstraction. Given a set of required component types, namely robotic capabilities, all compliant robots should be able to perform the same behaviours. For example, any mobile robot with sensing capabilities should be able to implement a simple avoid behaviour. This then requires that a behaviour template has a list of components required for implementation, or at least a list of capabilities required. Using this list to gain access to the now defined components, ACorDE implements the behavioural template to create the behavioural code through use of the robotic component variables and capabilities. As with component templates, the robotics community is free to create new templates at all levels, thus allowing for constant extension to the number of platforms that can
Table 1. Cursory comparison of the more popular robotic development tools.

<table>
<thead>
<tr>
<th>Name</th>
<th>Downloads</th>
<th>Classification</th>
<th>Modules</th>
<th>Support &amp; Extendibility</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player</td>
<td>241578</td>
<td>Robot Control Interface/Network Server for Robot Control</td>
<td>Predefined proxies</td>
<td>Supported hardware [14], New proxies, interfaces and drivers can be created by hand</td>
<td>C, C++, Python and Ruby. (Java, Ada, and Octave - 3rd party support)</td>
</tr>
<tr>
<td>ROS</td>
<td>unavailable</td>
<td>Robotic Operating System</td>
<td>Specific nodes within stacks</td>
<td>Supported hardware [15]. A robot driver is encapsulated in a node that exists within a stack</td>
<td>C++, Python, Java</td>
</tr>
<tr>
<td>MARIE</td>
<td>4634</td>
<td>Component-based software tool for building robotics software systems/Middleware framework</td>
<td>Access to those of the encompassed RDFs and middleware</td>
<td>Supports the platforms and extensibility of the underlying robotic tools</td>
<td>C++</td>
</tr>
<tr>
<td>MSRDS</td>
<td>0.5M</td>
<td>A comprehensive set of development tools, samples and tutorials</td>
<td>Web Services</td>
<td>Supported hardware [19] with the ability to write an RDS service for a new platform</td>
<td>Any .NET compatible language</td>
</tr>
<tr>
<td>Pyro</td>
<td>1559$^a$</td>
<td>An open-source Python robotics toolkit for exploring topics in AI and robotics</td>
<td>Player modules</td>
<td>Khepera, Pioneer, Sony AIBO and many simulated robots. Extension to other robotic platforms via Player extensions</td>
<td>Python</td>
</tr>
</tbody>
</table>

$^a$ Number of downloads from 25/05/2011 [20][21], project moved from SourceForge.net so earlier statistics not available
be implemented, and also what behavioural actions can be taken. The structure of these behavioural templates is a point of note. Control implementations typically take one of two stand points, the first primarily holds when the notion of behaviours running in parallel is followed. That is, behaviours are subsumable and inhibitable as per the Subsumption architecture [6]. The second point of view considers behaviours that are collected in a stack-like structure, typically after some form of deliberation, each to be implemented in turn. These behaviours are interrupted at any point to allow for a reactive element in the programming. Whether the progress of these behaviours is saved upon interruption or not is a design choice that can be made at a later stage, specifically during the architecture design.

4.3 Driver Level Abstraction

Driver abstraction is the process of structuring the form of a driver for a particular RDF, separating out the standard required functions and methods, and including place holders and separate smaller template snippets for each of the main components that can be implemented. These snippets are pulled in and filled by ACorDE once it is known what components have been chosen to represent the robotic platform currently in question. Typically only one such template is required per RDF. Along with driver and low-level control code generation, ACorDE also generates the project structure around the drivers often consisting of a particular directory structure. This is particularly important in projects utilising such RDFs as ROS, which is built upon the notion of stacks and nodes, and uses its own tools to create these stacks in the appropriate form. This directory structure and command call information is also encapsulated in the driver template.

4.4 Architectural Level Abstraction

Abstraction at the architectural level is a more difficult concept to pin down and define. Much of what characterises a control architecture is implemented in the structure of its behaviours and how those select behaviours interact with one another. In some cases, planners and reasoners are required, along with access to databases. At this level, ACorDE does not intend to generate specific goal-directed, architecturally designed control code. Rather it aims to provide structure diagrams, theory-based text and, potentially, sample code so that decisions at this level can be left purely with the developer. This is the portion of the system that can develop furthest with interaction from AI, cognitive science and the social robotics communities.

5 Implementation of ACorDE

ACorDE’s front end is implemented through PHP CLI (Hypertext Preprocessor Command Line Interface) with support from PHP-Glade and the PHP Gtk libraries. Interactive graphics use the PHP GtkMozEmbed element and embedded
HTML5. PHP 5 also natively supports the encoding and decoding of JSON files, which is of particular importance here as all templates, session files, and robot parameter files (RPFs) are written in the JSON, (Javascript Object Notation) format. PHP CLI was chosen over some of the more common graphical design languages, such as C++ or Java purely for its ease of use that allows for rapid prototyping. As a scripting language, it is also highly suited to file manipulation. Note, however, that the generated code is C++, other language support may be added at a later date with little additional effort.

While it is the most seamless method, it is not necessary to follow the GUI to develop driver control code with ACorDE. Following prescribed formats, a user may link manually filled component templates to form a robot parameter file and then use ACorDE’s internal functions to generate the desired driver files.

Following Figure 1, ACorDE pursues a linear progression when developing drivers and control code. Stage I is primarily an information retrieval stage composed of a number of steps. The first step, Stage I.1, involves the user selection of the required components that make up the robotic platform. Additionally, the user is asked to link component functionalities to robot API calls, and to point to or attach any platform dependent API files. Once completed, ACorDE then requests the variables as earmarked in the abstract template files to build an internal model of the robot. This information is stored in JSON files, from which, ACorDE can generate a robot parameter file.

Stage II requires the selection of an RPF and an RDF to continue. With this choice made, ACorDE can select the appropriate driver and directory structures and create both. These are saved into a user-defined location. Similar to Stage II, Stage III merely requires a choice of driver, or existing RPF along with RDF and behaviours to implement. Theoretically all existing behaviours could be generated, however, considering the vast number of potentially implementable behaviours for each combination of components, this is deemed excessive. Therefore, the user is asked to select those pertinent to their implementation. The option is left open for future extension of this choice of behaviour to the project.

Each of these stages may be run independently of one another, assuming the required files are already in existence. However, as both Stages II and III require little more than a few button presses, the system hinges on the abstracted templates and their resulting RPF.

6 Discussion

ACorDE aims to, essentially, lower the bar for non-specialists to use robotics in their work, and to enable rapid and reusable development code for those more comfortable with robotic development. Aside from findings external to robotics, there are a huge number of results and implementations within robotics that, although are vital in many applications, are not accessible. Consider, for example, the field of social robotics. Currently there is a significant push to provide both domestic and assistive robotics to the public [22]. However, acceptability is an issue. Social roboticists have been working with the theories of cognitive science
Fig. 1. Schematic showing a user’s intended progress through ACorDE to develop control code for a new robot. Stage I: A robot parameter file is created holding all of the platform’s physical and API data. Stage II: A driver is created for a particular RDF either in with the current robot parameter file or one previously saved. Step III: The user selects those behaviours most pertinent to their end application, this level is populated only by those behaviours relevant to the current robot parameter file configuration. This process is further discussed in the text.
for a significant time now and have amassed a wealth of results in acceptability of robots across the age range of potential users [23] [24]. Unfortunately, however, little if any of the code used for such studies is accessible to the wider community due to a number of reasons. Firstly, many of the HRI (Human-Robot Interaction) experiments that have taken place have been wizard-of-oz experiments, due to the nature of interaction experiments, the resulting short turnover time for robotic configurations, and the inaccessibility of the main RDFs for frequently changing hardware. Secondly, those that are not wizard-of-oz experiments have been programmed specifically for the hardware at hand.

The problem here is two-fold, the HRI community can not make use of RDFs for either providing their results as usable generic modules or use those already developed by the wider community for more rapid implementations of their work, nor can the wider community gain access to the code that implements these studied and peer-reviewed behaviours. This lack of cross-pollination across Robotics is a common issue which is both a cause and a result of the discipline’s relative immaturity. ACorDE provides a stepping stone to allow the various areas of research to converge more easily where it is applicable and not lose sight of what the other research areas are working on.

7 Conclusion

This work presented the main theory behind ACorDE and an outline of its implementation along with theoretical justification. Future publications will present much needed empirical justification to go along with the given stand point.

References

10 Robotic Driver Generation


