Trends in Robotic Software Frameworks

Davide Brugali[1], Gregory S. Broten[2] Antonio Cisternino[3], Diego Colombo[4], Jannik Fritsch[5], Brian Gerkey[6], Gerhard Kraetzschmar[7], Richard Vaughan[8], and Hans Utz[9]

[1] Universitá degli Studi di Bergamo, Italy brugali@unibg.it
[2] Defence R&D Canada Greg.Broten@drdc-rddc.gc.ca
[3] University of Pisa, Italy cisterni@di.unipi.it
[4] IMT Alti Studi Lucca, Italy diego.colombo@imtlucca.it
[5] Bielefeld University, Germany jannik@techfak.uni-bielefeld.de
[6] SRI International, USA gerkey@ai.sri.com
[7] Fraunhofer Institute for Autonomous Intelligent Systems, Germany gerhard.kraetzschmar@ais.fraunhofer.de
[8] Simon Fraser University, Canada vaughan@sfu.ca
[9] Ames Research Center, USA hutz@mail.arc.nasa.gov

1 Introduction

In the software community, a framework indicates an integrated set of domain-specific software components [CS95] which can be reused to create applications. A framework is more than a library of software components: It defines the common architecture underlying the particular applications built on the framework. Frameworks are a powerful development approach as they consist of both reusable code (the component library) and reusable design (the architecture).

Frameworks have acquired popularity in the object-oriented (OO) community. Here, the interpretation of "framework" ranges from structures of classes of cooperating objects, which through extension provide reusable basic designs for a family of similar applications [JF88], to the more restrictive view [Sch95] of complete, high level modules, which through customization directly result in specific applications for a certain domain. Customization is done through parameterization or by writing functional specifications.

Frequently, the two views of framework, referred to as white box and black box approaches to reuse, may be simultaneously present in one framework. In fact, features which are likely to be common to most applications can be offered and therefore reused as black boxes. Black-box frameworks support extensibility by defining interfaces for components that can be plugged into the framework via composition. Existing functionalities are reused by defining components that conform to a particular interface and integrating these components into the framework. On the other hand, the class library accom-

D. Brugali (Ed.): Software Eng. for Experimental Robotics. STAR 30, pp. 259–266, 2007.

panying the framework usually provides base classes (seen as white boxes) that can be specialized, by adding subclasses as needed, and easily integrated with the rest of the framework. White-box frameworks rely heavily on OO language features like inheritance and dynamic binding in order to achieve extensibility. Existing functionality is reused and extended by (1) inheriting from framework base classes and (2) overriding predefined hook methods using design patterns like the Template Method [GHJV95].

Framework based development has two main processes: development of the framework and development of an application adapting the framework. Every application framework evolves over time; this is called the framework life span [BMA97]. Within this life span, the basic architectural elements which are independent from any specific application are implemented first. When new applications are built using the framework, new components are developed which eventually are generalized and integrated as new black box elements. This means that in its early stages a software framework is mainly conceived as a white box framework. However, through its adoption in an increasing number of applications, the framework matures: More concrete components, which provide black box solutions for common difficult problems, as well as high level objects, which represent the major abstractions found in the problem domain, become available within the framework.

Since developing the framework requires a lot of effort, it is justified only if several applications will be built based on the framework. On the other hand, when the framework is available, developing an application from it offers considerable saving in time and effort compared to the development from scratch. The application is built by slightly modifying the framework, but by accepting the framework's high level design in general. In other words, most of the framework is reused, and reuse happens both at the high level design and at the code level.

Most framework-based software development is organized along the value-added principle [MFB02]. This divides any software system into a set of horizontal layers, each one built on top of another. This approach promotes separation of concerns and enforces modularity. Typically, a software framework is subdivided into a system infrastructure layer, a functional (also called horizontal or business domain) layer, and an application (also called vertical) layer.

The *system infrastructure layer* (usually called *middleware*) consists of a collection of software components that offer basic services like communication between distributed applications, uniform access to heterogeneous resources, independence from processing platforms and distributed location, distributed component naming, service brokering and treading, and remote execution. Examples of distributed middleware systems are Microsoft .NET, the implementations of OMG CORBA, and the Sun Enterprise Java Beans (see Sidebar *Middlewares for Distributed Computing* by Davide Brugali).

The customization of the system infrastructure layer is usually black-box. It consists in the aggregation of the frameworks' elemental components (such

as mechanisms for logical communication, concurrency, and device abstraction) in which the whole application has to be written.

The *functional* (also called *business domain*) *layer* is the model of a robotic application and consists of software components, which directly map to real entities like a vision system or to common robot functionalities such as sensor processing, mapping, localization, path planning, and task planning, or even to typical robot capabilities such as reactive behaviors.

Typically, they are implemented on top of the system infrastructure layer. For example, reactive behaviors delegate device abstractions for motion actuation, sensor processing, path planning, and task planning rely on the framework concurrency model in order to execute on different time scale, multi-robot mapping and localization build on data exchanged through a common distributed environment.

The customization of the functional layer is usually white-box. The basic components are intermediary classes, which by their very nature are fairly application independent, although they have been conceived bearing in mind a specific application domain (e.g. robot mobility or visual servoing). They are written using the elemental components of the framework, and they have to be specialized for each concrete application. It is a critical layer in frameworks for robotics applications because often functionalities depends, at least in part, on the underlying robotics architecture.

The *application layer* connects the functional components according to the information and control flows of any robot task. Examples of robot tasks are soccer playing, vacuum cleaning, museum tour guiding, mars surface exploration.

The customization of the application layer is usually grey-box. It consists of the interconnection of pluggable application-specific components through the middleware system. These components are the result of the adoption of the framework for the development of more and more applications and in some cases they are open source components-off-the-shelf (COTS). COTS products are designed to support a standard virtual interface, which consists of a set of rules for accessing the component's functionality in a homogeneous way, regardless of the execution platform, the programming language, and other specifics.

Software frameworks offer a unified view to model and develop robotic software systems at every level of the vertical decomposition from the system infrastructure to the final application through the functional model.

The chapters in the third Part of this book describe different approaches that address to some extent most or all of the aspects of robot framework development (such as black box or white box reuse). Individually, they propose innovative contributions on how to apply framework development concepts in the robotics domain. These approaches are illustrated in the next section.

Thereafter, the Sidebar *Middlewares for Distributed Computing* by Davide Brugali analyzes some of the most widely used distributed middleware frame-

works, i.e. OMG CORBA, Sun Java distributed technology, and Microsoft .NET.

2 Opportunities to Exploit Framework-Based Development in Robotics

Robot software developers have often experienced a sense of frustration when developing an application that is (nearly) the same as several other releases for different projects, but they have not been able to capture and exploit the commonalties. These problems frequently and typically occur in four circumstances.

1. When the application migrates to a different robot platform. Despite the differences in the mechanical structure, many commonalties can be identified in the control architecture, such as how to acquire sensory data or transmit commands to the actuators.
2. When the application scales up from a single robot to a multi-robot system or when it is restructured from centralized to decentralized and distributed control paradigm. Despite the differences in the communication mechanisms (shared memory or networked environment) the interactions among the control modules of the robot control architecture remain unchanged or show many commonalties.
3. When the same robotic system is used in different environments or for different tasks. Despite the differences in the control logic, most of the robot capabilities and functionalities are common in every application.
4. When the robotic system is build from the composition of heterogeneous or legacy subsystems. Despite the differences in the programming languages, concurrency models, or data interchange formats, commonalties can be found in the semantics of the components' behavior.

Software frameworks are a possible solution to the above problems, as illustrated in the following four sections.

2.1 Frameworks for Device Access and Control

One vital task for every robot system is to interface with hardware sensors and actuators. Familiar mass-market I/O devices such as mice and joysticks were once specialist experimental devices, comparable to the robot devices we build and buy today. Each device had a unique interface, and software had to be specially written to take advantage of it. Over time, as these devices became common, their external interfaces have converged to well-known logical standards. The hardware details are hidden behind an interface that encapsulates the logical functionality of the device. Other hardware, such as keyboards, disks and printers are treated similarly. A significant fraction of

the source code of a operating system such as Windows or Linux is composed of such "device driver" code.

Robotics devices are far more rare than mice and printers, but recently we have seen a limited amount of commoditization in research robots. The Pioneer 2 and 3 robots from Mobile Robots Inc, and the SICK LMS laser scanner are good examples of devices that are sold and used in the hundreds, all over the world. The Pioneers ship with the software framework "ARIA" that contains drivers for the Pioneer's hardware and the LMS. ARIA provides a high-level, structured (Object Oriented) environment for writing robot controllers, and provides some powerful modules for doing common tasks such as mapping and localization. The "Mobility" APIs provide similar functionality for RWI Robots, and also support the SICK LMS. ARIA and Mobility are proprietary systems that work only with their companies' robots. Indeed, they provide a key part of the value of the robot to their customers: they make programming robots much easier. But they target different robots and their APIs are very different. If you move your LMS from a Pioneer to an ATRV Jr. robot, you have to use completely different code to access your range data.

Another approach is taken by the "Player" robot interface, described in the Chapter *Reusable Robot Software and the Player/Stage Project* by Richar Vaughan and Brian Gerkey. The design philosophy of Player is to extend the computer's operating system up to meet the robot controller. Like an operating system, Player is designed to provide convenient access to hardware and software resources, but otherwise keep out of the way. Robot controllers using Player can be written in any programming language, and devices can be accessed via a network socket or via a linked C library. Thus Player places very few restrictions on the user's robot control code.

Player is a popular system, but its lack of constraints means that it imposes no structure or high-level abstractions that can usefully guide and facilitate controller design. Several projects have built on top of Player, wrapping the device drivers and network transparency with a more structured development environment, providing powerful object hierarchies or visual programming interfaces.

2.2 High-Level Communication Frameworks

A very challenging robotic field that demands effective integration techniques is that of personal robot companions. Looking at the more abstract cognitive abilities of robot companions, the integration of multiple modalities for achieving human-like interaction capabilities requires flexible communication frameworks that can be easily used by interdisciplinary researchers. The interdisciplinary researchers involved in integrated projects are neither integration nor middleware experts, so ease of use is crucial for the successful application of such a framework. Another aspect of research prototypes is that specification changes occur frequently during the development of these systems. Thus, the impact of interface changes on the system architecture should be minimal

to avoid versioning problems. This, in turn, allows rapid prototyping and iterative development that must be supported by a framework not only for single modules but also for the integrated system.

A framework that addresses these requirements is the XCF-SDK described in Chapter *An Integration Framework for Developing Interactive Robots* by J. Fritsch and S. Wrede. XCF provides a solution for flexible data-driven component integration as well as event-driven coordination by focusing on concepts from different middleware architectures that are relevant for the domain of intelligent systems research. This yields a lightweight API and provides features from both black-box and white-box frameworks. Within XCF, components are mainly described by the XML data they deliver to other system modules either directly or mediated through instances of an active XML database.

Another robotic communication framework is Robotics4.NET a message-oriented framework that supports the implementation of control software for robots designed to operate in human-based environments. It will be presented in Chapter *Increasing decoupling in the Robotics4.NET framework* by Antonio Cisternino et al. Its peculiarity relies in its structure centered around the notion of body of a robot. Body organs, called roblets provide an abstract view of the underlying hardware subsystem they are responsible for. An XML-based, message-oriented communication infrastructure acts like the spine, connecting organs to the brain.

2.3 Frameworks for Functionality Packaging

Supporting specific target applications, such as common tasks in robotics scenarios, is mandatory to allow robotic experts to focus on the problem domain, instead of dealing with the entire problem set of the application domain.

An application framework supports the development of applications for common robotic tasks (such as video image processing or behavior based reactive control), which manage the control flow and organize the data flow for the task at hand, shielding the application programmer from much of the intrinsic difficulties of software development within a concurrent, distributed software environment. Providing a framework for the tasks control and data flow also facilitates to add centralized support for addressing timeliness and scalability problems within the application.

Chapter *VIP: The Video Image Processing Framework based on the MIRO Middleware* by Utz et al. presents a video image processing framework (VIP) that targets the problem of applying computer vision in the field of real-time constraint autonomous mobile robotics. It aims to provide an environment that allows the developer to concentrate on the robot vision task without risking to jeopardize the reactivity and performance of the system as a whole.

The VIP framework features the above discussed different aspects of robotics framework design. The white-box framework manages the control flow and organizes the data flow, shielding the developer from the subtle

locking issues and memory management problems. In order to meet the requirements of high-performance real-time-constraint robotics applications, it provides a powerful sensor-triggered, on-demand processing model for image operations. The framework also offers a set of tools that assist in the debugging, assessment and tuning of the vision application. Furthermore, the middleware integration provides a sophisticated communication infrastructure for distributed robotics applications as well as powerful facilities for the configuration and parameterization that are exploited by the framework components to enhance and their flexibility and reusability. A fast growing library of black-box components for the framework confirms the applicability of this application framework in very different robotics scenarios.

Chapter *MRT: Robotics Off-the-Shelf with the Modular Robotic Toolkit* by Bonarini et al. illustrates a modular approach to develop and integrate functionalities of mobile robots that are involved to perform tasks autonomously. The approach is based on the exploitation of a distributed middleware and the customization of an application framework. The middleware makes the interaction among functional modules transparent with respect to their physical distribution. The application framework allows to reuse the same functional units in different applications, thus reducing the development time and increasing the software reliability. Moreover, modules managing the interaction with the environment are designed to make independent the actual interface with physical devices from the needed data processing.

2.4 Frameworks for Legacy Systems Evolution

A well established practice in computing science, frameworks have only recently entered common use in robotics. For industry and government, the complexity, cost, and time to re-implement stable systems often exceeds the perceived benefits of adopting a modern software infrastructure. Thus, most persevere with legacy software, adapting and modifying software when and wherever possible or necessary – adopting strategic software frameworks only when no justifiable legacy exists. Conversely, academic programs with short one or two year projects frequently exploit strategic software frameworks but with little enduring impact. The open-source movement radically changes this picture. Academic frameworks, open to public scrutiny and modification, now rival commercial frameworks in both quality and economic impact. Further, industry is now realizes that open source frameworks can reduce cost and risk of systems engineering.

Chapter *Towards Framework-based UxV Software Systems, An Applied Research Perspective* by G. Broten et al. describes Defence R&D Canada (DRDC)'s experience as they migrated from a legacy, tele-operated system, to a modern frameworks based robotics program. Encountering limits to teleoperation, DRDC reoriented its focus to autonomous, intelligent systems. DRDC reviewed its software development methods and concluded that a new, extensible, flexible, modular and scalable approach was needed and that

older work would need to be rewritten or jettisoned. Combatting a "must be invented here" tradition, DRDC readily examined the current trends in robotics and quickly realized a framework approach could mitigate many risks and relieve many constraints incurred by pure in-house software development. DRDC's experience confirms the widely held belief that framework use simplifies application development and maintenance. DRDC adapted the Miro framework, initially designed for the robosoccer competition, to implement an autonomous, outdoor unmanned ground vehicle (UGV). This effort exploited Miro's White-box framework attributes to extend existing classes for DRDC's unmanned systems requirements. Using the Miro framework DRDC was quickly able to create a modern, fully autonomous UGV by integrating legacy code and other open source code sources to a workable application.

3 Conclusions

Although Framework-based software development is an emergent area in Robotics, it reflects the natural evolution in the development of robotic software applications.

As the notion is in its infancy, a proper context and understanding of the concepts, characteristics and challenges are needed for robotic systems developers.

In this chapter we have provided a taxonomy of characteristics for classifying robotic software frameworks and have described these characteristics with a focus on the challenges robotic software engineers commonly struggle with today. In this sense, we believe we have identified a rich set of opportunities for the robotic community to enhance software reusability in Robotics.

References

[BMA97] D. Brugali, G. Menga, and A. Aarsten, *The framework life span.*, Communication of the ACM **40** (1997), no. 10, 65–68.

[CS95] J.O. Coplien and D.C. Schmidt, *Pattern languages of program design*, ch. Frameworks and Components, pp. 1–5, Addison-Wesley, 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Villisides, *Design patterns: Elements of reusable object oriented software*, Addison-Wesley, 1995.

[JF88] R.E. Johnson and B. Foote, *Designing reusable classes.*, Journal of Object-Oriented Programming (1988).

[MFB02] Hafedh Mili, Mohamed Fayad, Davide Brugali, David Hamu, and Dov Dori, *Enterprise frameworks: issues and research directions*, Software Practice and Experience **32** (2002), 801–831.

[Sch95] H.A. Schmid, *Creating the architecture of a manufacturing framework by design patterns.*, Proceedings of OOPSLA'95 (1995).

[Vel05] T.L. Veldhuizen, *Software libraries and their reuse: Entropy, Kolmogorov complexity, and Zipfs law*, Proceedings of the first Library Centric Development (LCSD) workshop (2005).