Trends in Component-Based Robotics

Davide Brugali[1], Alex Brooks[2], Anthony Cowley[3], Carle Côté[4], Antonio C. Domínguez-Brito[5], Dominic Létourneau[4], Françis Michaud[4], and Christian Schlegel[6]

[1] Universitá degli Studi di Bergamo, Italy brugali@unibg.it
[2] University of Sydney, AUSTRALIA a.brooks@cas.edu.au
[3] Federal University of Minas Gerais, MG, Brasil chaimo@dcc.ufmg.br
[4] Université de Sherbrooke, Department of Electrical Engineering and Computer Engineering, Sherbrooke (Québec), CANADA {Dominic.Letourneau, Carle.Cote, Francois.Michaud}@USherbrooke.ca
[5] Universidad de Las Palmas de Gran Canaria, Spain adominguez@iusiani.ulpgc.es
[6] University of Applied Sciences Ulm schlegel@fh-ulm.de

1 Introduction

Component-Based Software Engineering (CBSE) is an approach that has arisen in the software engineering community in the last decade. It aims to shift the emphasis in system-building from traditional programming to composing software systems from a mixture of off-the-shelf and custom-built components [Cas00, HC01, Szy02, DW98, CC01]. Component-Based Software Engineering is said to be primarily concerned with three functions [HC01]:

1. Developing software from pre-produced parts
2. The ability to reuse those parts in other applications
3. Easily maintaining and customizing those parts to produce new functions and features

As robotics systems are becoming more complex and distributed there is the need to promote a similar environment, where systems can be constructed as the composition and integration of reusable building blocks. System modularity and interoperability are key factors that enable the development of reusable software. If a system is modular, its functionalities can be customized by replacing individual components. When two or more systems are interoperable, they can be (re)used as components of more complex systems.

As an analogy, in the electronics domain re-usable off-the-shelf electronic components have been available for many years in the form of integrated chips (ICs) which can be bought and deployed in other parts of the world. This is possible because each IC packages a clear set of functionality and provides a

D. Brugali (Ed.): Software Eng. for Experimental Robotics. STAR 30, pp. 135–142, 2007.

well-defined external interface. Furthermore, numerous standard tools exist to design electronic devices based on the composition, assembly and combination of these electronic components.

Out of the multiple definitions for software components we can find in literature, we adopt the one given in [Szy02] as it illustrates clearly the concept:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties".

According to the previous definition, a software component is a piece of software that has been independently developed from where it is going to be used. It offers a well-defined external interface that hides its internals, and it is independent of context until instantiation time. In addition, it can be deployed without modifications by third parties. Also, it needs to come with a clear specification of what it requires and provides in order to be able to be composed with other components.

In an attempt to clarify the definition, we offer four properties of components identified by Collins-Cope [CC01]:

1. A component is a binary (non-source-code) unit of deployment
2. A component implements (one or more) well-defined interfaces
3. A component provides access to an inter-related set of functionality
4. A component may have its behaviour customized in well-defined manners without access to the source code.

Components can be put together in various configurations to form a system. This flexibility comes from modularity, enforced by the contractual nature of the interfaces between components. Interfaces and their specifications are viewed in isolation from any specific component that may implement or use the interface. In addition, the contractual nature of interfaces allows components on either side of the interface to be developed in mutual ignorance. Designing each component and interface in isolation allows a component that implements an interface to be seamlessly replaced with a different component having the same interface.

The component approach improves software development by reducing the amount of code that has to be written by the application designer. In particular, assembling previously-existing components greatly reduces time to test new applications. When a component is used in a large number of systems by different developers, the knowledge about the component usage, robustness and efficiency is available in the user community. The more a component is used the less it costs.

From the component developer's viewpoint, domain analysis is the most critical phase. In order to be broadly reusable, individual components have to be written to meet the requirements of a variety of applications in a specific domain. The current trend is toward a "product line" approach: the developer designs and implements a complete set of components for a family of similar applications.

From the application developer's viewpoint, the user requirements specification should be made by having in mind which components are already available. According to Boehm [Boe95], "in the old process, system requirements drove capabilities. In the new process, capabilities will drive system requirements...it is not a requirement if you can't afford it". In the early stages of an application development, the designer has to take into consideration which components are available, which integration effort they require, and whether to reuse them as they are or build new components from scratch.

2 Opportunities to Exploit CBSE in Robotics

General purpose component approaches require far too much software engineering knowledge, in particular if one has to address all the challenges of robotics software development and integration. Therefore, there is a need for a methodology and technology that tailor general purpose component approaches such that one can take advantage from the progress made in the software engineering community without being forced to become a software engineering expert as robotics researcher.

The following domain characteristics make experimental robotics particularly suited to a component-based development approach:

Inherent Complexity Even a fairly simple robotic system, a single vehicle working in isolation, is complex. Correct operation requires the interaction of a number of sensors, actuators and algorithms. Robotics software always requires to cope with the inherent complexity of concurrent activities, the deployment of software components on networked computers, ranging from embedded systems to personal computers, a bunch of different platforms, operating systems and programming languages and even with different real-time requirements. Typically, only some parts require hard real-time and for many parts soft real-time or even no real-time requirement exist.

Requirements for Flexibility Researchers need systems that are flexible enough to allow them to experiment with one particular aspect or algorithm, without their experimentation having large repercussions for the rest of the system. The possibility of the development of software systems based on interoperable components would be extremely beneficial to the research community since every research group, even the smallest, would have the possibility to concentrate its efforts on a small piece of the robotics puzzle. For example, experts in automated planning could experiment new path planning algorithms for a mobile robot relying on the obstacle avoidance and self-localization functionalities encapsulated in off-the-shelf components. Reuse of consolidated and shared components would allow different teams to test their algorithms on common benchmarks in order to assess performance.

Distributed Environments Robotic systems are inherently distributed. When dealing with single vehicles it's often convenient to develop on workstations, controlling robots remotely. Often the processing power on a single robot is insufficient, requiring the use of off-board processing power. Code should not have to change if the off-board components are moved onboard. Increasingly, complex single vehicles are becoming multi-processor [UAC04]. Multi-robot systems and sensor networks have many more distribution issues. Location transparency allows components to be distributed readily and easily, according to processing and bandwidth constraints.

The chapters in the second Part of this book advocate Component-Based Software Engineering (CBSE) as an valuable approach for facing the challenges listed above. When the application domain, like Robotics, has rapidly evolving requirements, a computational model which allows a higher adaptability of the reusable software components and flexibility in their pattern of interactions is mandatory.

The Sidebar *Software Architectures* by Patricia Lago and Hans van Vliet reports on the role that software architectures play in the software development process.

In the following sections we identify four issues that need to be addressed in order to define a computational model of reusable components for robotic software applications. Finally Section 3 draws the relevant conclusions.

2.1 Specifying a Component Internal Behavior

Certain level of uniformity in behavior and structure in software components, in spite of the functionality they have individually, is critical to allow easy integration and replacement of interchangeable components.

Components have an internal state, which reflects their knowledge. What are the specific requirements of a component model for robotic applications with respects to the definition of its internal state? In other words, which requirements in terms of internal structure should components have in order to apply CBSE to robotics? At first sight, a question comes naturally out from a development approach based on integrating software components. Is the whole the joining of its parts?, or, in other words, is a system just the joining of its components?

Components have a behavior which is determined by the set of admissible operations on their internal state. What are the specific requirements of a component model for robotic applications with respects to the definition of its behavior?

Chapter *CoolBOT: a Component Model and Software Infrastructure for Robotics* by Antonio Dominguez-Brito et al. will present a component model for robotic applications that faces the above issues on internal structure and behavior uniformity. The proposed component model captures two key aspects that enable component integration: the component behavior should be

observable and *controllable* by an external supervisor (e.g. another component).

By *observable* we mean that components have some internal aspects about their internal state which could be observed and monitored by an external entity. Examples of observable aspects of a component are its execution priority and its error handling behavior in the occurrence of any faulty situation.

As to *controllable* we mean that an external supervisor should be provided with means and mechanisms to control and to affect some aspects of a specific component during its execution. Examples of some of these controllable aspects might be, for instance, forcing the component to transit to specific states, changing the priority at which a component is executed at runtime, injecting artificially an exception in a component in order to originate a faulty situation during its execution, interrupting, suspending or resuming its execution.

2.2 Specifying a Component External Interface

Components have a message-based interface, which define the set of messages that the component can interpret and process. What are the specific requirements of a component model for robotic applications with respects to the definition of its external interface?

While hardware platform designers benefit from reusable device components with standard interfaces, application developers benefit from the existence of tested, documented components that implement common algorithms and techniques. This composition approach requires that each component must be strictly limited in scope and designed such that it functions entirely in its own context, completely independent of other components, or any container application. A clear definition of a component's interface facilitates modular robot design by allowing an incremental construction system wherein software capabilities may be added to a platform in step with additional hardware capabilities.

To aid this process, it is often helpful to consider the potential explicit coupling between components. This is the familiar process of interface definition, but with the requirement that any transaction between components be entirely self-contained and independent of context. This context-free, explicit coupling can be used as a blueprint for the functional scope of the component: all functionality should follow directly from the explicitly defined component interface. In this way, the component's general identity is immediately established by how it may be coupled with other components.

Chapter *ROCI: Strongly Typed Component Interfaces for Multi-Robot Teams Programming* by Luiz Chaimowicz et al. presents the ROCI approach for the definition of component interfaces that relies on the use of strongly typed, self describing data structures. The Components built using this type of interface can be coupled in a simple and bug proof manner, making the task

of the application developer easier. The ROCI approach has been applied to the programming teams of robots.

2.3 Enforcing Components Interoperability

Robotics software always requires to cope with the inherent complexity of concurrent activities, the deployment of software components on networked computers, ranging from embedded systems to personal computers, a bunch of different platforms, operating systems and programming languages and even with different real-time requirements. Typically, only some parts require hard real-time and for many parts soft real-time or even no real-time requirements exist.

The basic idea of the approach presented in Chapter *Communication Patterns as Key Towards Component Interoperability* by Christian Schlegel is to provide a small set of generic communication patterns that support easy integration of robotic components. All component interactions are squeezed into those predefined patterns. Using communication patterns with given access modes prevents the user of a component from puzzling over the semantics and behavior of both component interfaces and usage of component services.

Communication patterns relieve the component builder from error-prone details of distributed and concurrent systems by providing approved and reusable solutions for inter-component interactions. They handle complex synchronization problems in distributed systems and decouple the component internals from the externally visible and standardized behavior.

Communication patterns provide the basis for providing dynamic wiring. Dynamic wiring supports a context and task dependent assembly of components as is needed in nearly all robotics architectures.

2.4 Supporting Components Integration

Many existing software components can be found in the robotics community and it would be beneficial to reuse them in an integrated fashion using their initial implementation, saving time and avoiding introducing errors when reimplementing everything from scratch. Unfortunately, integration of existing software components is difficult knowing that they are typically developed independently, following their own set of requirements (e.g. timing, communication protocol, programming language, operating system, objectives, applications). Moreover, those components are often available "as is", with not much support from their developers and with minimal documentation. Reusability in this context is challenging but crucial for the evolution of the field, avoiding becoming experts in all the related areas that must be integrated.

Chapter *Using MARIE for Mobile Robot Component Development and Integration* by Carle Côté et al. presents MARIE, a software architecture that addresses three important issues in component-based robotic software development:

1. Being able to interconnect heterogeneous software components, from legacy to novel "state of the art" software components.
2. Being able to support a wide range of communication protocols and mechanisms to cope with the fact that there is no unified protocol available, and no consensus has yet emerged from the robotic software community.
3. Being able to support multiple sets of concepts and abstractions within the integration frameworks, to help multidisciplinary team members to collaborate without having them to become experts in every aspect of robotics software development.

MARIE's layered software architecture provides integration tools to help wrapping existing and new application functionalities in reusable software blocks. Each layer abstracts and manages different aspects needed to integrate heterogeneous software components together, such as communications, data handling, shared data types, distributed computing, low-level operating system functions, component architecture, reusable software blocks and system deployment, etc. With this approach, MARIE offers a flexible and extendable software architecture suitable in various integration situations, and allows software developers to work at the required abstraction level to craft reusable software blocks or to build robotic systems.

2.5 Deploying Components in Large-Scale Robotic Systems

A particular issue which is often overlooked concerns work that is required after a set of interoperable components have been developed, downloaded or purchased. There are several steps which must occur before reliable operation of the system can be achieved. Components must be connected to form a system, configured with appropriate parameters, and deployed onto the hardware hosts. To ensure reliable operation and to detect and correct any faults, the components must be monitored effectively.

For small systems operating over a short period of time, these extra steps can be and often are performed manually. For more realistic systems with more components, distributed over more hosts, or required to operate continuously for long periods of time, these steps can come to dominate the total effort required. In addition to the failures caused by components' internals, larger systems are more likely to exhibit communication failures and errors due to the interactions between components. The mean time between failures in the system can become far smaller than the mean time between failures of the least reliable constituent component.

Chapter *Orca: a Component Model and Repository* by Alex Brooks et al. identifies some of the more problematic issues and discusses ways in which a CBSE framework can assist developers and system integrators in overcoming them. The discussion is based on experience using Orca to implement systems ranging from single indoor robots to teams of heterogeneous outdoor vehicles. Details of several of these systems are presented.

3 Conclusions

The different solutions presented are giving hints concerning what a complete component-based approach should provide in order to support the development of reusable robotic software building blocks:

- A component model that allows the observation and control of its internal behavior.
- Well defined component interfaces and data structures that clearly define its usage modalities.
- Communication patterns that enable the interconnection of reusable components.
- Communication abstractions that support the interoperability of heterogeneous components.
- Well identified components repositories that simplify documentation, retrieval, and deployment of a large number of reusable components.

The combination of all these concepts is by itself a hot topic for future research.

References

[Boe95] B. Boehm, *A technical perspective on systems integration*, Proceedings of the SEI/MCC Symposium on the Use of COTS in Systems Integration (1995).

[Cas00] Castek, *Component-based development: The concepts, technology and methodology*, Castek Company's white paper, available at www.castek.com, 2000.

[CC01] M. Collins-Cope, *Component based development and advanced OO design*, White paper, Ratio Group Ltd., 2001.

[DW98] D. D'Souza and A. Wills, *Objects, components, and frameworks with UML: The catalysis approach*, Addison-Wesley, 1998.

[HC01] G. T. Heineman and W. T. Councill (eds.), *Component-based software engineering : putting the pieces together*, Addison-Wesley, Boston, 2001.

[Szy02] C. Szyperski, *Component software - beyond object-oriented programming*, Addison-Wesley / ACM Press, 2002.

[UAC04] C. Urmson, J. Anhalt, M. Clark, T. Galatali, J. P. Gonzalez, J. Gowdy, A. Gutierrez, S. Harbaugh, M. Johnson-Roberson, H. Kato, P. L Koon, K. Peterson, B. K. Smith, S. Spiker, E. Tryzelaar, and W. Whittaker, *High speed navigation of unrehearsed terrain: Red team technology for grand challenge 2004*, Tech. Report TR-04-37, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, June 2004.