Trends in Robot Software Domain Engineering

Davide Brugali[1], Arvin Agah[2], Bruce MacDonald[3], Issa A.D. Nesnas[4], and William D. Smart[5]

[1] Universitá degli Studi di Bergamo, Italy brugali@unibg.it
[2] University of Kansas, USA agah@ku.edu
[3] University of Auckland, New Zealand b.macdonald@auckland.ac.nz
[4] Jet Propulsion Laboratory, USA nesnas@jpl.nasa.gov
[5] Washington University in St. Louis, USA

1 Introduction

Domain Engineering is a set of activities aiming at developing reusable artifacts within a domain. The term domain is used to denote or group a set of systems or functional areas within systems, that exhibit similar functionality.

The fundamental tenet of Domain Engineering is that substantial reuse of knowledge, experience, and software artifacts can be achieved by performing some sort of commonality/variability analysis to identify those aspects that are common to all applications within the domain, and those aspects that distinguish one application from another [CHW98].

In some application domains, such as telecommunications, factory automation, and enterprise information systems, large companies or international committees have defined standards for reference architectures (e.g. the ISO-OSI model or the USA-NBS Reference Model for Computer Integrated Manufacturing [MMB83]) and software frameworks (e.g. TINA [TIN], CIMOSA [ZK97]). We argue that this is not a viable approach for solving the problem of developing reusable and interoperable robotic system.

The reason can be found in the peculiarity of the robotic domain: a robot is a multi-purpose (what it is for), multi-form (how it is structured), and multi-function (what it is able to do) machine. It is this diversity in form and function that requires a model for flexibility and efficiency beyond those developed in other domain application. The goal of defining a one-size-fits-all architecture or framework for every robotic application is elusive.

In contrast, we believe that robotic software experts should pursue the goal of identifying stable requirements, common design issues, and similar approaches to recurrent software development problems while developing every new robotic application. If the robotic community shares this common vision, then reusable software solutions can naturally emerge from the profitable exchange of knowledge and experience and from common practice.

D. Brugali (Ed.): Software Eng. for Experimental Robotics. STAR 30, pp. 3–8, 2007.
© Springer-Verlag Berlin Heidelberg 2007

2 Opportunities to Characterize the Robotics Domain

We propose five opportunities to characterize the Robotics domain, namely the need of conceptual tools to identify the stable requirements of robotic systems, the need of abstract models to cope with hardware and software heterogeneity, the need of development techniques to enable a seamless transition from prototype testing and debugging to real system implementation and exploitation, the need of a methodology to manage the complexity of real world deployment of robotic applications, and the need of tools to assist the software engineer in the development of robotic applications.

These opportunities have led to the definition of innovative software development approaches and solutions that are thoroughly described in the five chapters of the first part of this book.

Thereafter, the Sidebar *Programming Commercial Robots* by José María Cañas et al. reports on several software development environments for commercial robots.

2.1 Assessing and Documenting a Robotic Domain Model

In order to make the large software corpus available today within the Robotics community reusable, there is a need to make the domain knowledge and design experience behind it transparent to the developers of new robotic systems.

We argue that this can be achieved if the robotics community pursues two main objectives:

1. To identify recurrent and stable aspects in the robotic domain, which emerge from the analysis of the current practice of software development in robotics. A domain reference model, and a development methodology which incorporates reuse both of design and code, are essential vehicles for the successful development of large and complex robotic control systems.
2. To formulate these stable and reusable aspects in a common language that can allow the understanding of the proposed solutions and make them reusable for the resolution of new problems.

The Chapter "Stable Analysis Patterns for Robot Mobility Software" by Davide Brugali formulates the problem of developing stable software systems in the robotics domain and in particular of stable robot mobility software. It presents a set of conceptual tools (Stable Analysis Patterns) that help the robot software developers identify the stable aspect of a robotic application and guide them in the process of building stable and reusable software components (Stable Design and Architectural Patterns) and systems.

2.2 Defining Abstract Models to Manage Heterogeneity

Developing reusable robotic software is hard because of the inherent complexity and multi-disciplinary nature of the robotics domain. Developing robotic

capabilities with an overarching objective of supporting new platforms and algorithms that are not known a priori is a real challenge.

One of the primary difficulties stems from the heterogeneous nature of robotic hardware. Today's robots have different physical capabilities, sensor configuration, and hardware control architectures. On the physical differences, robots can be aerial, surface, or underwater vehicles. Many are mobile but some can be stationary such as manipulators. Some mobile platforms can be legged, wheeled, or hybrid of the two. Sensor suites can also significantly differ on various platforms. Some have stereovision cameras mounted on a pan tilt unit for 360 degrees coverage, while others have stationary stereo cameras that are only pointed by moving their mobile base. It is this heterogeneity of the physical capabilities that often makes robotic software unique to each platform.

Another set of challenges stems from the heterogeneity of robotic software. First, new algorithms developed in isolation (without a reference model) are difficult to integrate because they would often use different representations of information. Second, it is both desirable and necessary to develop robotic software in a modular fashion without sacrificing performance. Modularity is necessary for both reuse and for testing the individual components. Third, robotic software needs to be adapted to hardware or simulators at different levels of granularity. This is needed because functionality, in some cases, can be migrated to hardware.

Chapter *The CLARATY Project: Coping with Hardware and Software Heterogeneity* by Issa A.D. Nesnas, describes a methodology that uses a commonality / variability analysis coupled with iterative approach to define, design, and prototype software components with reuse as one of the primary objectives. By adapting these generic components to various heterogeneous platforms over several years, one hopes that some of the more stable components of a robotics framework mature to the point of becoming reusable robotic entites.

2.3 Managing the Evolution from Simulation to Reality

Development of software for autonomous field mobile robots can be a challenging task due to the difficulties associated with testing of the robot. Because testing is an integral part of all phases of software development, the robot and its software must undergo extensive testing during its software life-cycle. However, testing a field mobile robot requires access to the field, logistics, maintenance of the robot, and adhering to safety standards. .

Simulating some parts of the system can make testing and debugging much easier; realistic simulation can help in determining numerous hardware design parameters, such as physical dimensions, propulsion, payload distribution, etc.

When parts of the system are simulated, a seamless evolution to the real system is fundamental; also, the changes involved must be of a local nature.

If we make changes to the non-simulated components, we can no longer trust the results obtained by the simulation, and its whole value would be lost.

A twofold approach to address these issues is described in Chapter *Simulation and Testbeds for Autonomous Mobile Robots in Harsh Environments*, by Arvin Agah *et al.* It consists of (1) developing a software architecture that is reusable and platform independent, and (2) building a virtual prototype and a small mobile robot to serve as testbeds for the software development. The small robot is a scaled down version of the field robot with as many identical sensors and actuators to the field robot, as possible.

2.4 Coping with Software Deployment and Maintenance

When writing robot control software, we are not simply trying to control the robot itself. Rather, we are controlling the interaction between the robot and its environment. This makes development and testing in a laboratory very different from running the system in the real world. In particular, if the environment in which the system is to operate is different from that in which it was developed, we cannot guarantee that it will function as expected. To accommodate these changes, we write our software to be as general as possible. In practice, this often means a large number of environmentally-dependent parameters have to be set in order for the system to work properly. However, this leads to a problem: How do we set the parameters correctly, while meeting the deadlines often imposed by robot deployments?

Even well-parameterized software will not be able to deal with all contingencies, and it is almost certain that new software will have to be written on-site to make the deployment succeed. Often this software is written under extreme time pressure, against unmovable deadlines (the museum opens, the event to be monitored is about to happen). Software written under these conditions cannot be expected to be as robust as that developed and tested over time.

Chapter *Writing Code in the Field: Implications to Robot Software Development*, by William D. Smart examines these issues in more depth. How can we design robot control software to make the process of deploying robots in the real world more efficient and effective? How can we mitigate the effects of hastily-written software during the deployment? How can we streamline the process of tuning and adapting the software to the deployment environment?

2.5 Software Environments for Robot Programming

The goal of building effective robotic assistants for humans has never been more relevant, yet robot system developers face the complexity of mobile robotic systems, the difficulties of variable and unpredictable environments, and the challenge of making human–robot interaction effective and safe. Unfortunately, robotic programming tools have not advanced as rapidly as the robots themselves, nor as rapidly as the demands for complex robotic tasks.

Existing tools are often *ad hoc*; they may be specific to the robot hardware, lack open standards, and ignore the human involvement.

We believe robot programming tools must be targeted more closely to robotics, paying attention to the needs of the robot developer and therefore the nature of typical robot programs, the typical skills of robot programmers, the interactions between humans and robots, and the programming constructs that prevail in robotic applications. In designing the developer's toolchain and choosing the underlying technologies for robotic programming, we must account for the presence of human developers in the immersive robot environment, and recognise that it is the *robot's interaction with the environment* that makes robot programming different and challenging. It is *programmer's lack of understanding of the robot's world view* that makes robot programs difficult to code and debug. It is the *expressiveness of the language* that the human programmer uses to describe the robot's behaviour.

Chapter *Software Environments for Robot Programming* by Bruce MacDonald *et al*, focuses on case studies of improvements in three areas related to the human programmer; the expressiveness of the robot programming language, the expressiveness of the robot software framework, and the need for human–oriented robot debugging tools.

3 Conclusion

Software development for robotics is somewhat different from (and more difficult than) software development for other computer systems.

1. Robotics is more a research field than an application domain and robot researchers seem to shun standardization, since everyone wants to write their own architectures and code. Robotic software is quite complex and custom crafted solutions have been more dominant because they initially appear less complex. However, custom crafted solution do not scale well and can lead to more complex systems when integrating multiple capabilities. The development approach most likely to succeed is a very generic one that leverages common functionality and where everyone can integrate their component technology. Domain analysis is a promising approach to identifying commonalties.
2. The multi-disciplinary nature of the robotics domain and the heterogeneity of robotic technologies (hardware platforms, control paradigms, capabilities) make the reuse of existing software solutions very hard. Nevertheless, by looking at robots as system with multiple levels of abstraction, we can develop reusable components at every level of the hierarchy and reuse appropriate components in different platforms. Abstractions can exist at the hardware interfaces, the low-level capabilities as well as at the higher-level capabilities.

3. In a robot system, we are designing the robot-environment interaction, and we often don't know the environment side of the system in detail as we are developing the robot control application. If we have a large, parameterizable library of robot code that implement consolidated models (both for simulated and real robots), this will make deployments easier.
4. Designers of robotic systems ought to include the human developer as a significant role. Robot development tools should provide expressive programming languages and frameworks that enhance the opportunity for human developers to describe robot behaviour. Debugging tools should be human–oriented and improve the immediate visualisation of robot data for the human developers.

These characteristics are what makes software development for robotics more an art than an engineering discipline. The goal of this first part of this book is to present four approaches that exemplifies the art of crafting software for robotic systems in an engineering way.

References

[CHW98] J Coplien, D Hoffman, and D Weiss, *Commonality and variability in software engineering*, IEEE Software **15** (1998), no. 6.

[MMB83] C. McLean, M. Mitchell, and E. Barkmeyer, *A computer architecture for small batch manufacturing*, IEEE Spectrum (1983).

[TIN] TINAC, Tech. report, http://www.tinac.com/.

[ZK97] M. Zelm and K. Kosanke, *Cimosa and its application in an iso 9000 process model*, Proceedings of the IFAC Workshop-MIM'97 (1997).