Lessons from Applying Modern Software Methods and Technologies to Robotics

Lorenzo Flückiger
Carnegie Mellon University
NASA Ames Research Center
lorenzo@email.arc.nasa.gov

Hans Utz
USRA/RIACS
NASA Ames Research Center
hutz@email.arc.nasa.gov

***Abstract*— Autonomous mobile robots of today are software-intensive systems. The rapidly growing code base of robotics projects therefore requires advanced software development methods and technologies, to ensure their scalability, reliability and maintainability. In this paper, we analyze modern concepts and technologies that we applied to our software development process and show how they contribute to enhancing these design dimensions.**

I. INTRODUCTION

Today most of robotics systems are software-intensive systems, and thus call for appropriate use of development methodologies and tools. Advances in robotics capabilities go in pair with advances in computing systems. The first robots obviously had very limited computing capacities compared to today. In the past a single person could carefully craft the entire software for a robot, often using a bottom-up approach to keep the system functional with the scarce resources available. Today, most robots are driven by powerful processor(s) with ample memory, and their software is designed and written by an entire team of people, often distributed among various institutions.

The previous SDIR-05 workshop focused on identifying problems in robotic software development and proposing solutions. These solutions mostly consist of elaborate robotic architectures that have been developed to address the complexity of robotic systems. Rather than presenting another architecture, we would like to illustrate in this paper how a wider adoption of proven methodologies from the software engineering field is highly beneficial to robotics. We acknowledge that robotics software has many specificities that make it difficult to develop for, but we also strongly believe that advanced software engineering practices and technologies can be leveraged in the field of robotics to improve the quality, reliability and maintainability of robotic systems.

Current research platforms for space robotics, such as the SCOUT, ATHLETE or Robonaut robots [1]–[3] significantly extend the capabilities available to today's planetary robots such as the Mars Exploration Rover (MER) [4]. In space robotics, space qualified hardware apply that will probably restrict these robots' computational power to below what earth-bound robot systems have available at the same time. Nevertheless, the targeted feature set will result in software systems with such a degree of complexity that it will vastly benefit from modern software development technology with respect to maintainability, scalability and reliability.

The remainder of the paper is organized as follows. The second section of the paper introduces the requirements linked to robotic software development in the context of our research area. The third section of the paper expose how some key practices in software development can help construct robotic software. In particular we show how reusing existing assets, using middle-ware, creating abstract interfaces and decoupling the components lead to a more scalable and flexible software system. The benefits of the implemented solution regarding the scalability and flexibility of our robotic systems are exposed in the fourth section.

II. ROBOTIC SOFTWARE FOR EXPLORATION ROBOTS

The Intelligent Robotics Group (IRG) at NASA Ames is dedicated to improving the understanding of extreme environments, remote locations, and uncharted worlds. IRG conducts applied research in a wide range of areas with an emphasis on robotics system science and field testing. Current applications include planetary exploration, human-robot fieldwork, and remote science. In this context, the IRG rover software is subject to the two classical difficulties encountered by current robotics systems: 1) managing the intrinsic complexity due to the multiple domains involved in robotics and its inherent connection to a large number of unique hardware devices; 2) managing scalability as more sensors, actuators and control schemes are integrated as well as with respect to multi-robot missions that include human-robot interaction.

The robotic software developed at IRG needs to support the variety of hardware platforms currently in use: six wheeled Martian rover analog *K9*, multiple versions of our low cost four wheel rover *K10*, and the latest Antarctic traverse rover *K11*. IRG uses these robots for diverse experiments calling for various sensor or actuator configurations (orientable spotlight, pan-tilt camera, indoor tracking system, outdoor GPS, etc.) and conducts field tests requiring integration of various scientific instruments (microscope imager, drilling system, ground penetrating radar, etc.). In addition to the robot controller itself, the robotic software developed at the IRG includes components from the group's areas of expertise in applied computer vision and human-robot interaction. Finally, the robot controller needs to smoothly

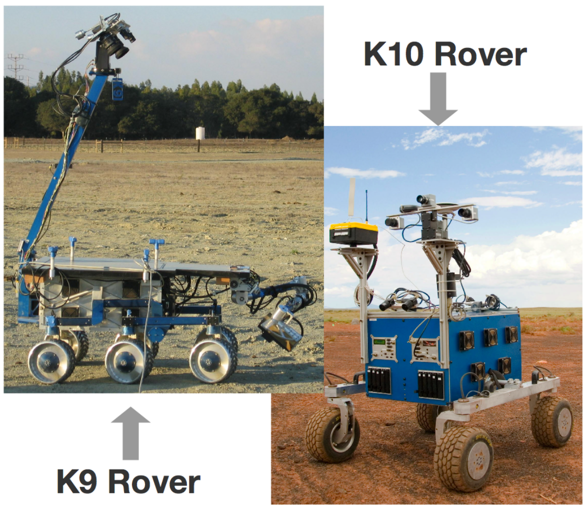

Fig. 1. The IRG's rovers currently in service. Each rover has completely different hardware controllers and internal architecture and was designed for different types of experiment. Nonetheless they share a large common code base and can be controlled using the same network-transparent interfaces.

integrate with the interactive 3D visualization and monitoring systems for ground control.

To manage this complexity and create a scalable system the robotic software is required to:

- reuse control frameworks developed in-house, or by external groups, to minimize the code development and maximize the reliability (same framework used in multiple scenarios)
- handle the distributed characteristic of mission scenarios involving multiple robots and multiple modes of interaction
- include a modular and flexible robot controller to quickly and easily adapt the system to new scenarios

III. SOLUTION APPROACH

Despite all its specificities, robotic software nonetheless remains software, and thus good general principles and best practices are applicable. So it is no surprise that the solution we choose to address our robotic scalability and complexity problem follows one of the key practices used as foundation of the Rational Unified Process (RUP) [5], a widely used software development methodology: ”Elevate the level of abstraction” (see Fig. 2).

This RUP principle encompasses a number of practices that contribute to the overall scalability and flexibility of software design. In this paper, we concentrate on the following proven practices: ”reusing existing assets”, ”leverage higher level frameworks”, ”focus on the architecture” and ”decoupling of components”. In addition, this paper introduces how our project leverages some advanced software technologies, like the middleware CORBA [7]. We are especially interested in how these practices and technologies could be applied within robotic software architectures.

ELEVATE THE LEVEL OF ABSTRACTION
Benefits: Productivity, reduced complexity
Pattern: Reuse existing assets, reduce the amount of human-generated stuff through higher-level tools and languages, and architect for resilience, quality, understandability, and complexity control.
Anti-pattern: Go directly from vague high-level requirements to custom-crafted code.

Fig. 2. ”Elevate the level of abstraction” pattern [6]

A. Reuse existing software assets

Reuse of software assets can save significant time to a robotic team, enabling it to focus on its core research instead of developing or redeveloping software. However, code reuse is not an easy task since it presupposes that the existing assets were designed to be reused. It is hard to anticipate all the usages of the component during the original design of a reusable component. CLARAty [8], the Coupled Layer Architecture for Robotic Autonomy lead by the Jet Propulsion Laboratory (JPL), provides an extensive set of robotics frameworks such as locomotion and navigation subsystems. Applying them to different physical robots can both save development time and leverage the robotic expertise encapsulated in the framework. IRG contributed for several years to the CLARAty project on specific topics, and in return, benefits from this large code base targeted to planetary rovers. Currently IRG's robots are using several high level capabilities offered by CLARAty. One requirement to gain access to the CLARAty control frameworks is to write adaptations of generic hardware abstractions for the targeted hardware device.

1) Hardware Devices: As with most robotic systems, the various platforms differ vastly with regard to the sensor, actuator and controller hardware. IRG created adaptations of several base CLARAty classes to benefit from higher level constructs. The best example is the adaptation of the CLARAty generic ”Controlled Motor” to the K9, K10 and K11 rovers[1], enabling to use the ”Locomotion” framework which computes motor commands from higher level drive commands.

2) Control Frameworks: By abstracting the hardware devices, the IRG rovers benefit mainly from two control frameworks provided by CLARAty: the ”Locomotor” and the ”Navigator”. For example, the Navigator will compute a sequence of drive commands to reach a goal while avoiding obstacles extracted from a point cloud.

B. Use high level software systems

To enable the scalability of our robotic system while minimizing its complexity, we consider each robot controller, all control systems (user GUI, astronaut commands, planner actions, etc.) and every high level scientific instrument as

[1]K9, the various K10 and K11 have different hardware controllers, meaning several different adaptation of the generic Controlled Motor

individual components. An additional requirement we put on the architecture is to allow each of these components to run on different nodes and communicate through the network, while keeping efficiency for localized components. Middleware technologies, like CORBA, can manage the distributedness of components and their communication.

The robotics middleware Miro [9] makes extensive use of CORBA as communication infrastructure and customizes it for the robotics domain. Our approach is to apply these middleware concepts to our robot software infrastructure and factor CLARAty frameworks into network transparent services with high-level abstract interfaces.

Miro offers support for the following paradigms to the robotic world:

- Distributed or localized communication using the CORBA infrastructure
- A set of abstract interfaces to allow communication between objects and the propagation of data structures
- A Publish/Subscribe protocol to distribute telemetry among components of the system
- A Parameter and Configuration Management framework

C. *Focus on architecture with interface definitions*

Separating interface definition from implementation is a pre-requisite for a consistent and extensible architecture. It also enables parallel development that often takes place within a team or among distributed teams across institutions. This separation can be performed using interface classes that are a proven concept for encapsulation. Interface classes contain only method declarations, no method definitions and no data. Recent programming languages, such as Java, provide this concept as part of the language. It is possible to enforce this concept in the C++ domain by using a neutral language to declare the interfaces [2].

In our project, the interfaces between all the subsystems and devices are defined in CORBA Interface Definition Language (IDL) [10]. As shown in Fig. 3, language specific interfaces are then automatically generated from the IDL. The code generator can create interfaces for multiple language bindings. The benefit is that programs written in different languages can all interact with the same robot controller. Using this scheme, the interface definition is kept unique across the project for all programming languages used. Although the generated code by itself does not implement strictly pure abstract interfaces, developers cannot modify the code without changing the interface definition. This scheme effectively enforces the separation of interface from implementation.

D. *Decoupling*

Abstract interface definitions and a service-based design allow for decoupling the individual subsystems into a component-based architecture. Component-based design

[2]Interface classes can be created in C++ using pure virtual methods. However, nothing prevents a developer to mix interface definition and implementation.

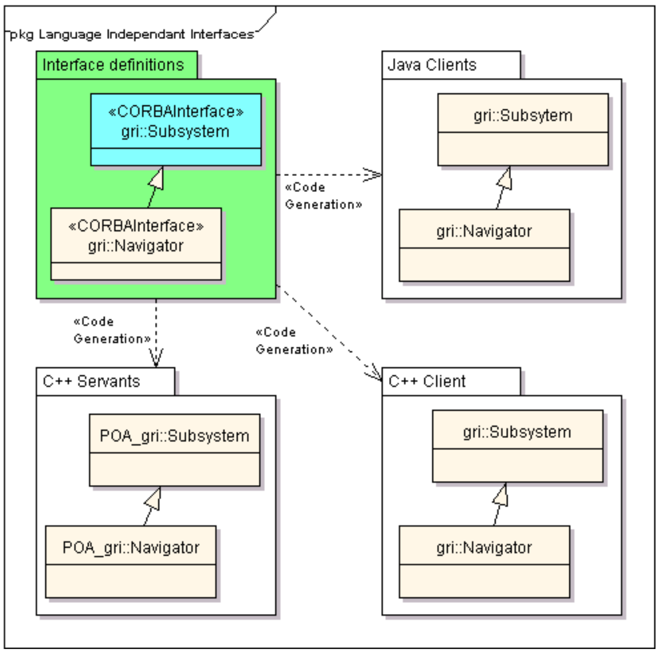

Fig. 3. From the Interface Definition Language, interfaces for various programming language are generated

requires the ability to flexibly wire the inputs and outputs of the different components for the different run-time configurations of a robotic system. Sensors are added to the system or temporarily removed. Different application scenarios require additional components in the system. So the software configuration of the applied system changes rapidly during the initial testing, and even after deployment.

CORBA based systems and similar middleware-oriented designs allow different modules of the system to be started in their own process, allowing the individual addition and removal of modules to the run-time configuration. Nevertheless, managing half a dozen or more processes is not a trivial task. A specific startup sequence is usually required due to dependencies of the different modules. Often, the parameterization changes for individual runs of a module. Furthermore, separating modules into individual processes on the same machine can result in unnecessary interprocess communication overhead where co-location optimizations are possible. Tools like MicroRaptor [11] help in managing processes on different machines in a distributed robotics scenario, but can do little about combining services on the same machine.

1) Component model: So far, generally applicable component models are not readily available to the robotics domain. They are either limited in scope by design, such as JaveBeans, which is essentially tied to the Java world. Others are limited in availability, such as the CORBA component model (CCM) [12], for which interoperable implementations have still not hit the mainstream. Nevertheless, off-the-shelf available frameworks (such as ACE [13]) can provide important features of a component-based archtitecture within todays robotic applications.

The high degree of decoupling of individual robotics services allows for combining them in a Service Oriented Architecture (SOA), using the Component Configurator pattern. Fig. 4 shows an example of SOA applied to our robots. The services are grouped into multiple dynamic libraries and the controller uses run-time linking to load and configure the individual components for a specific scenario. This minimizes the need for recompilation and relinking, shortening turnaround times, as well as reducing the memory footprint of the controller. Unused parts of the system, such as controllers for unused sensors or algorithms not needed within a specific scenario, do not get loaded into memory. The Component Configurator pattern also encapsulates reconfigurability, allowing one to discard and re-enable services without the need to stop the other controller services. It also helps to prevent the software system from becoming bloated by components from former application scenarios that accidentally get interlinked with the core system and can no longer be removed without significant development effort and risk to overall system stability. The discardability of individual components encourages factoring out reusable parts into base libraries, keeping the individual subsystems optional to the overall robot controller.

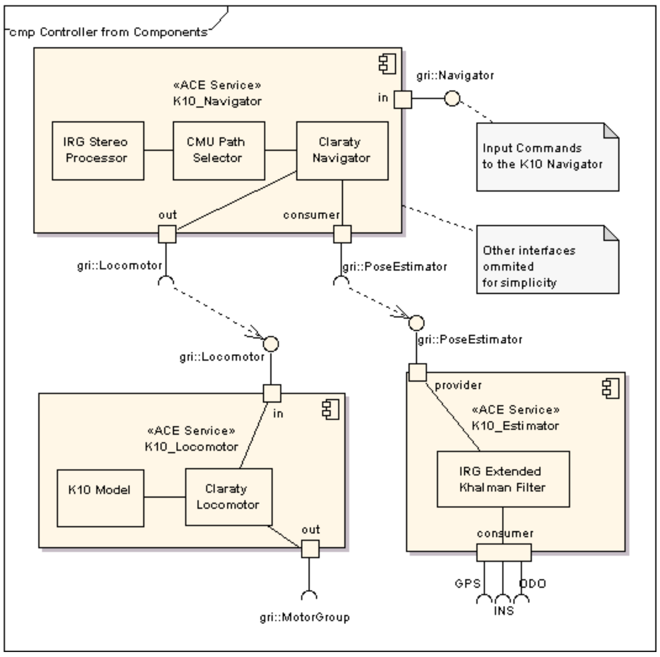

Fig. 4. Illustration of a subset of the components used to dynamically construct a robot controller using the Service Configurator pattern. The connections between the components are created at run-time. When a service is started, the corresponding component exposes its interfaces so they can be discovered by the other components.

2) Publish-Subscribe architecture: Middleware provided infrastructure can also enable further decoupling of the components of a software architecture. CORBA provides multiple specifications for publisher-subscriber architectures. The most feature-rich is the Notification Service, which is configured in Miro for data distribution purposes. Publisher-subscriber models decouple information sources (data suppliers) from information sinks (data consumers) by an intermediate object, the event channel. This is mostly used in a push-model of communication, where the data consumers register an instance of a callback interface with the event channel and are called when suppliers push new data into the channel. The notification service can push data to the consumer in a separate thread and also provides advanced filtering capabilities to decouple the control flow between the producer and the consumer side.

3) Control flow decoupling: We also applied another concept from the distributed systems middleware to further decouple control flow between clients and servers for regular method calls: Asynchronous Method Invocation (AMI). Robot operations often take a lot of time. For instance, a navigation task for a robot can take several minutes. At the client side, it is undesirable to have a blocking operation that hands off the thread of control to the server side for the complete operation. At the same time, it is of utter importance for the client to know the outcome of the task which was handed of to the server. The server side design is greatly simplified by the use of a blocking semantics, where the controller exits the control loop after the task is finished and returns success or failure to the caller. AMI provides a communication pattern for this problem set. In essence, it allows a blocking servant method to be called in a non-blocking manner. The IDL-compiler generates an alternate method to call in the client-side proxy, which will immediately return after the call is dispatched to the servant. The client can provide a callback that will be executed once the servant has finished. The return value and out-parameters of the method are the parameters of the callback's signature.

IV. PRELIMINARY RESULTS

The abstractions and modularizations described in the above section allowed for a set of improvements in our robot control infrastructure, that would have been very difficult to achieve without these concepts and technologies.

A. Advances in Scalability

a) Abstract interfaces: The abstract service interfaces allow control of different robots through the same interface. Furthermore, they facilitate the replacement of the robot controller with a robot simulator without changing anything on the client side.

b) Publisher/subscriber architecture: The publisher/subscriber architecture used for telemetry distribution decouples the suppliers of information from the consumers. This enables easy replacement of input streams for sensor-centric processes, such as pose estimation, with logged data streams for development and evaluation purposes.

c) Link time dependencies: The use of abstract service interfaces resulted in a tremendous reduction in link-time dependencies to other subsystems. Where a former client application communicating with a high-level interface drew in more than 40 conceptually unnecessary library dependencies to other robot software modules, the abstract interface design is limited to about half a dozen. These libraries contain code actually used by the client implementation.

d) Remote inspectability: An important side effect of network transparent high-level interfaces is that they add remote inspectability for each individual service. This is an important entry point for scripting, unit testing, and online-supervision of the system in operation. In case of a failure, the individual components can be analyzed as part of the running system, tremendously reducing the time to locate the culprit. This interaction can even be used to work around some of the problems the autonomous system encounters by human intervention.

B. Advances in Flexibility

Component-based architectures provide a very high level of flexibility by allowing for extensive configurability. This however requires proper support to stay manageable. The different components need to access other components for pulling in additional information from sensors and subsystems, as well as for pushing results. These links need to be easily reconfigurable to ensure the flexible applicability of a component.

e) Run-time configuration: For the Component Configurator pattern [14] we rely on the implementation provided by the ACE library, the Service Configuration framework. It provides a meta-server concept for services and uses run-time linking to draw in services from dynamic libraries in a plug-in architecture like manner. The configuration of the meta-server can be specified in a configuration file on startup. Additionally, it can be altered at run-time. There are two different grammars available for the configuration file syntax: A custom designed one as well as an xml-based version. This setup allows us to add science instruments or other robot payloads to the software system without having to alter or re-link any part of the base-controller.

f) Service configuration: For the configuration of the individual services, the Service Configurator framework only provides command-line equivalent capabilities. Therefore, for service configuration, the parameter framework provided by Miro is deployed. The tool support of parameter framework (code generation, GUI editors) saves development time and reduces error sources in the configuration process, such as syntax errors or typos in configuration files.

V. CONCLUSION AND FUTURE WORK

In this paper we analyzed the way in which software methods and technologies contributed to our ongoing task of mastering the intrinsic complexity and scalability issues of the robotics domain. We concentrated on four practices to increase the level of abstraction of our robotic software: 1)reuse existing control frameworks to leverage acquired expertise while minimizing development effort; 2) use robotics middleware to allow the distribution of components and simplify communication; 3) create high-level interfaces to guarantee a consistent and extensible architecture while supporting parallel development; 4) decouple the components of the system using a service oriented design and advanced middleware features to increase flexibility and reusability. The result of this work is a ”loosely coupled, highly cohesive” system providing a reconfigurable software architecture adaptable to different robotic application scenarios.

The benefits in productivity, reliability, and maintainability of our new software architecture convince us that there is much to leverage from the software methodologies for the robotic field. We hope that this paper will encourage the robotic community to utilize more of the practices and technologies readily available from the software engineering field.

The work in this paper was mostly centered on a single robot system. Our ongoing research targets, among others, multi-robot applications with respect to team fault-tolerance against individual or temporary robot dropouts. In this context we continue leveraging advanced software development methods and technologies.

VI. ACKNOWLEDGMENTS

This work was supported by the NASA Exploration Technology Development Program and the Mars Technology Program.

REFERENCES

[1] T. W. Fong, I. Nourbakhsh, C. Kunz, L. Flückiger, R. Ambrose, R. Simmons, A. Schultz, and J. Scholtz, “The peer-to-peer human-robot interaction project,” in *AIAA Space 2005*, Long Beach, California, September 2005.

[2] R. Hirsh, J. Graham, K. Tyree, M. Sierhuis, and W. J. Clancey, “Intelligence for human-assistant planetary surface robots,” in *Intelligence for Space Robotics*, A. M. Howard and E. W. Tunstel, Eds. Albuquerque, NM: TSI Press, 2006, pp. 261–279.

[3] W. Bluethmann, R. Ambrose, M. Diftler, S. Askew, E. Huber, M. Goza, F. Rehnmark, C. Lovchik, and D. Magruder, “Robonaut: A robot designed to work with humans in space,” *Autonomous Robots*, vol. 14, no. 2, pp. 179–197, March 2004.

[4] J. J. Biesiadecki and M. W. Maimone, “The mars exploration rover surface mobility flight software driving ambition,” *Aerospace Conference, 2006 IEEE*, pp. 15 pp.–, 2006.

[5] P. Kroll and P. Kruchten, *The Rational Unified Process made easy*. Addison-Wesley, 2003.

[6] P. Kroll and W. Royce, “Key principles for business-driven development,” *Rational Edge*, 2005. [Online]. Available: http://www-128.ibm.com/developerworks/rational/library/oct05/kroll/

[7] Object Management Group. (2006) CORBA FAQ. [Online]. Available: http://www.omg.org/gettingstarted/corbafaq.htm

[8] I. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin, “CLARAty and challenges of developing interoperable robotic software,” in *Proceedings of the 2003 International Conference on Intelligent Robots and Systems (IROS 2003)*. Las Vegas, Nevada: IEEE/RSJ, October 2003.

[9] H. Utz, S. Sablatnög, S. Enderle, and G. K. Kraetzschmar, “Miro – middleware for mobile robot applications,” *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, vol. 18, no. 4, pp. 493–497, August 2002.

[10] Object Management Group. (2006) OMG IDL. [Online]. Available: http://www.omg.org/gettingstarted/omg_idl.htm

[11] (2006) Mircroraptor web site. [Online]. Available: http://gs295.sp.cs.cmu.edu/brennan/mraptor

[12] D. C. Schmidt and S. Vinoski, “Object interconnections: The CORBA component model: Part 1, evolving towards component middleware,” *C/C++ Users Journal*, February 2004.

[13] D. C. Schmidt and S. D. Huston, *C++ Network Programming*. Addison-Wesley Longman, December 2002, vol. 1.

[14] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley & Sons, 2000.