Code Reusability Tools for Programming Mobile Robots

Carle Côté, Dominic Létourneau, François Michaud,
Jean-Marc Valin, Yannick Brosseau, Clément Raïevsky, Mathieu Lemay, Victor Tran
LABORIUS – Department of Electrical Engineering and Computer Engineering
Université de Sherbrooke, Sherbrooke (Québec) CANADA, J1K 2R1
Email: {Carle.Cote, Dominic.Letourneau, Francois.Michaud}@USherbrooke.ca

***Abstract*— This paper describes two initiatives aiming at improving code reusability for programming mobile robots: RobotFlow/FlowDesigner, a data-flow programming environment; MARIE (Mobile and Autonomous Robotics Integration Environment), a programming environment allowing multiple applications, programs and tools, to operate on one or multiple machines/OS and work together on a mobile robot implementation. RobotFlow/FlowDesigner's objective is to provide a modular, graphical programming environment that will help visualize and understand what is really happening in the robot's control loops, sensors, actuators, by using graphical probes. MARIE aims at avoiding making an exclusive choice on particular programming tools, making it possible to reuse code and applications.**

I. Introduction

The intelligence of a system depends on its sensing, acting and processing capabilities, not taken individually but as a whole. Over the last two decades, we have seen the field of autonomous mobile robotics progress rapidly with the technological evolutions of computers, sensors and actuators, allowing the elaboration of various algorithms and approaches for sophisticated decision-making processing applicable to such systems. Obstacle avoidance, navigation, localization, mapping, planning, modeling, recognition, searching, tracking, interaction, cooperation, decision-making, just to name a few, are good examples of such algorithms developed on different robotic platforms, to validate research hypothesis and to learn about the challenges to overcome in making a mobile robot act as an intelligent and useful agent in the real world.

While good progress has been made, a lot remains to be discovered about how to accomplish such goal. A key obstacle to overcome is the issue of integration. When the Behavior-Based Subsumption Architecture was introduced in 1986 [1], one of the most fundamental contribution was to demonstrate the importance of integration in mobile robotics. Yes it is critical to work on specific capabilities such as SLAM, vision, planning, speech, etc., but making them work as a whole is a necessary step for increased intelligence manifested by autonomous machines in real life settings. Without a doubt, the task is certainly a difficult one: no standards in mobile robotics exist, with everybody developing capabilities on their own (on different robots, using different operating systems and programming environments) and trying afterward to combine it with others. This is where the design team of the robot GRACE put some of their efforts in the 2003 AAAI Mobile Robot Challenge [2]. We had observed the same difficulties with our robot entry in 2000 [3], making the robot going through the entire steps of the challenge using symbol recognition, a touch screen interface, emotional expression, autonomous recharging and HTML reports. From this experience we identified two critical issues: programming tools to manage the complexity in programming sophisticated decision-making algorithms for robots, and programming environments that facilitate code reuse. Note that we make a distinction between a software architectural methodology for decision-making of an autonomous agent (such as [1], [4], [5]), and programming environments which are used to implement architectural methodologies (such as Saphira, Mobility, etc.). Our focus in this paper is on the latter.

The mobile robotic community recognizes the need for the evolution of standards in the field, to allow the industry to re-exploit what have been shown to work and to make progress using these elements. Currently a great variety of programming environments and tools for robotics exist (see http://www.orocos.org/related.html for a list). Initiatives also exist to produce a functional basis for robotic software system (project OROCOS), device abstractions for robotic programming (Player/Stage [6]), navigation toolkits (CARMEN [7]), or low-level robotics building blocks [8]. Most of these initiatives are developed independently of the others, driven by specific applications and objectives in mind. One solution is to choose the one with the most diverse capabilities and the biggest community of users, guaranteeing somewhat that support will be provided over the years. But it may still be too soon to make such choice, with the field having to explore a great variety of ideas, application areas (each one having its own set of constraints, e.g., space, military, human-robot interaction, etc.) and to cope with continuously evolving computing technologies.

So we asked ourselves the following question: how can we come up with an efficient way not to reinvent the wheel every time we have to program a robot, without imposing tools that would have to suit the needs of the very broad robotic community? The challenge is to come up with a solution that helps the science in mobile robotics progress, allowing everybody to benefit from the works of others. Our answers to this question led to two initiatives related to programming mobile robots. RobotFlow/FlowDesigner

is a modular data-flow programming environment that facilitates visualization and understanding what is really happening in the robot's control loops, sensors, actuators. MARIE (Mobile and Autonomous Robotics Integration Environment), a programming environment allowing multiple applications, programs and tools, to operate on one or multiple machines/OS and work together on a mobile robot implementation. RobotFlow/FlowDesigner and MARIE are described in the following sections, along with the justification behind the design choices made for these tools.

II. ROBOTFLOW/FLOWDESIGNER

A programming environment, designed to visualize decision blocks, reusable buildings blocks and their interconnections, surely helps to structure systems and to understand mechanisms. This idea has been used for many years now with scientific and engineering software like Matlab/Simulink, LabView and the LEGO Mindstorm programming environment. These environments, however, do not fulfill all of the needs associated with the programming of sophisticated intelligent decision-making algorithms for autonomous mobile robots. With the great variety of mobile robotics platforms and their programming environments (proprietary or not), code reusability is limited by not being able to easily import code from one platform to another. Debugging is a critical part of robot programming, and having the platform move in the world makes it difficult to see what works and what does not in various dynamic situations. Tools that facilitate debugging and data inspection are therefore very important. At the same time though, with limited processing power onboard robots, computational overhead has to be limited to a minimum in order to allow good response time when running complex algorithms.

In order to address these challenges, the FlowDesigner (http://flowdesigner.sourceforge.net) project was initiated in 1999. FlowDesigner is a C++ data-flow processing library coupled with a graphical programming environment. The graphical interface is shown in Figure 1. With data-flow networks, two interaction mechanisms can be implemented: push and pull [9]. Pushing is when an interaction between processing elements is initiated by the data sender (producer); pulling occurs when an interaction is initiated by the data receiver (consumer). Push connections are appropriate for communication triggered by asynchronous events, while a pull network instructs the source element to send data only when the destination element is ready to process. FlowDesigner was originally designed for image and audio signal processing (DSP), having to deal with synchronous data processing. That explains why we chose a pull mode architecture. The pull mechanism also provides the simplicity of designing processing elements that do not have to be aware of the others and where everything is self-scheduled. Self-scheduling happens when blocks are asked to output their results: each output block (sink blocks) call their input blocks to compute recursively in order to be able to obtain the input data required for calculation. This kind of computation does not require to have a specific scheduler that tells when a block has to process its input data. This simple implicit scheduling mechanism makes it possible to build networks from smaller functional blocks without running into efficiency problems caused by scheduling overhead.

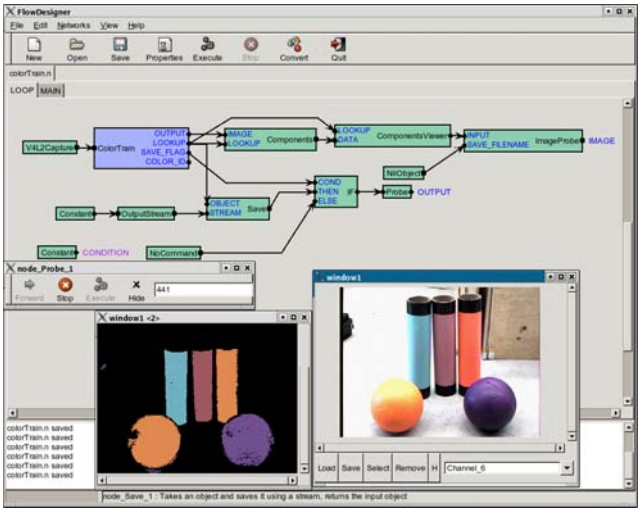

Fig. 1. Color extraction using FlowDesigner and RobotFlow

FlowDesigner allows to create reusable software blocks, and to link them together using a standardized mechanism in order to create a network of blocks. The blocks are connected dynamically at run time. A network composed of multiple blocks can be inserted as a super-block into other networks. The composition mechanism is inspired by the Composite object pattern [13]. Blocks and super-blocks can then be treated the same way, which help preserve uniformity of the blocks interface (inputs, outputs, parameters, scheduling). Super-blocks allow the creation of user-defined processing blocks that can be reused and modified easily with the GUI. This makes the global system easier to maintain since everything is grouped into smaller functional blocks. Once the blocks are properly connected, the user can execute the program directly from the GUI interface or from a shell terminal, which parses the textual description of the network. Allowing FlowDesigner programs to run in a shell terminal provides flexibility and requires less ressources.

FlowDesigner's buffering mechanism allows blocks to compute their outputs only once per iteration for better efficiency. During a given iteration, if block A has calculated its outputs which are requested by block B, block A just returns the results stored in its output buffers, without propagating the request recursively to its input blocks. Buffer size is managed by the system, enabling blocks to request outputs over the N previous iterations, enabling the creation of feedback blocks. FlowDesigner also defines standard data types and operators that can be used by blocks. Creating new C++ blocks does not require knowledge of FlowDesigner's internal processes, but only the procedure to define inputs, outputs, parameters, and the desired processing function for calculation by the block.

Automatic type checking and type conversion are provided by the data-flow library. Objects creation and destruction are handled by the system, avoiding dealing with memory allocation. Using standardized data types and conversion operators reduces complexity of the C++ blocks, improves code readability and helps uniformize blocks. When linking blocks together with the GUI, users are automatically notified when a link between two blocks is invalid, which prevents errors and misuses of a block. User-defined data types and operators can easily be added in new toolkits.

Multiple toolkits are already available, e.g., for audio processing, artificial neural networks, fuzzy logic, visualization probes, vector quantization (VQ), and Gaussian Mixture Models (GMM). Visual and textual debugging probes are available in these toolkits, providing a flexible and useful way to visualize the information transiting in the data-flow network. Probes can be inserted as standard blocks in the network, without modifying the data content. For instance, built-in probes can show data either on the terminal or on the screen. More complex graphical probes can plot the data (2D) in real time. Interactive probes can also be created with buttons, text areas, graphics, keypad, etc. They can interrupt the processing of a block, allowing step-by-step debugging. Once everything works as planned, probes can be removed without recompiling, an useful capability while debugging mobile robots.

RobotFlow (http://robotflow.sourceforge.net) is the mobile robotics toolkit based on FlowDesigner. Using the RobotFlow toolkit, programmers do not have to worry about low level drivers for robots and devices and are able to build programs with sufficient hardware abstraction. The system also provides enough flexibility to create higher level intelligence modules without forcing the programmer to use a pre-determined mechanism for behavior arbitration and selection. Behavior blocks, Pioneer2 robots interface blocks and vision processing blocks are already available for programmers to create complex systems. For instance, as shown in Figure 1, extracting color components from an image and displaying it in a graphical image probe require the use of just a few building blocks and links. A color training probe makes it possible to select membership to color channels directly on the image. RobotFlow also has Player/Stage [6] client interface blocks which allows the control of all Player supported platforms and devices, along with interface to Stage and Gazebo simulators. Using these links, RobotFlow provides a convivial interface for creating higher level programs using a powerful and flexible client/server robot and device interface like Player. Blocks in RobotFlow could be made to make it compatible with Mobility, Saphira, etc.

For comparison, Evolution Robotics (http://www.evolution.com) offers a similar programming environment based on a Graphical User Interface (GUI) to connect software components. The layered software architecture provides higher level modules for vision, navigation and interaction on top of a Task Execution Layer (TEL), a Behavior Execution Layer (BEL) and a Hardware Abstraction Layer (HAL). The HAL provides low level operating system (OS) interfaces and hardware interfaces. The BEL provides mechanisms for creating behaviors, which are basic building blocks on which software applications are built. Behaviors coordination is performed automatically by the system. The Behavior Composer GUI helps connecting behaviors inputs and outputs. The TEL provides infrastructure for creating goal-oriented tasks, enabled by an event-driven mechanism. With RobotFlow, our goal is to let programmers decide how behaviors are connected, selected and arbitrated, what kind of software architecture and intelligence mechanisms to use, and which data types to exploit. A lot remains to be discovered about how to program intelligent autonomous mobile robots, and in our opinion flexibility is more important than user-friendliness or automatic and hardcoded mechanisms at this point. RobotFlow's graphical user interface, mostly incorporated into probes, is somewhat minimal, but programmers have all the liberty of building much better GUIs that are suited for their needs.

Overall, FlowDesigner and RobotFlow are mostly useful when dealing with sequential (synchronous data-flow) processing. Processing with mobile robots are often synchronous: it starts with sensing, perception, deciding what to do and taking action using the effectors. RobotFlow also allows reuse of networks of blocks, or parts of networks of blocks, for multiple mobile robotics projects. However, the pull scheduling used in FlowDesigner policy is not well suited for asynchronous processing. Other scheduling policies could be developed with FlowDesigner (push, events) to better fit robotics needs and are to be addressed in the future. With asynchronous processing, finite state machines and petri nets become easier to do because each block in a network can switch to different state according to events. Nevertheless, RobotFlow allows to create finite state machines that are incorporated into one block controlling every state, and outputing commands to other blocks. Processing threads can be created within blocks or networks of blocks, allowing parallel calculations using different update rates and events. The GUI also runs in a different thread, making GUI events possible. Being able to make parallel computation (distributed computing) would probably require to run multiple instances of FlowDesigner/RobotFlow using different update rates. Such issues in system communication are addressed by MARIE.

III. MARIE

Reusability in robotics can be seen at a functional level where small reusable blocks of functionalities are created and linked together to build a robotics system, as done by RobotFlow/FlowDesigner. It can also be seen at a system level where applications, tools and programming environments are created and linked together. Those two levels can be used concurrently in the same system as they offer different integration possibilities. As indicated in [10], many programming environments based on one or both levels exist, proposing different approaches for robotics

system development and integration. Unfortunately, many of these programming environments are not compatible with each other and cannot be easily integrated together. MARIE's first goal is to create a programming environment at a system level, facilitating reusability of applications, tools and programming environments in an integrated and coherent system.

Distributed system must integrate many applications that communicate with each other either locally or on separate processing nodes. Creating direct application-to-application communication links is not always possible since applications do not necessarily share the same communication protocols (TCP, UDP, shared memory, etc.) or communication mechanisms (push, pull, event-based, etc.). Many programming environment, such as Player [6], CARMEN [7], MIRO [11] and CLARaty [12], have solved this problem by choosing specific communication protocols and/or mechanisms that need to be implemented by all applications to be linked together. This solution has some limitations knowing that it might not be possible or desirable to modify the source code for an application. Furthermore, it limits coexistence of multiple communication protocols and mechanisms interacting together. Another solution is to import the functionalities of various applications by re-implementing them in a common framework, and then adding what is required for communication. This solution typically involves an error-prone work that requires time, effort and knowledge to assure that functionalities are not deteriorated or modified during the migration process. Unless for compatibility, optimization or performance issues, re-implementing functionalities in a common programming environment should probably be avoided whenever possible.

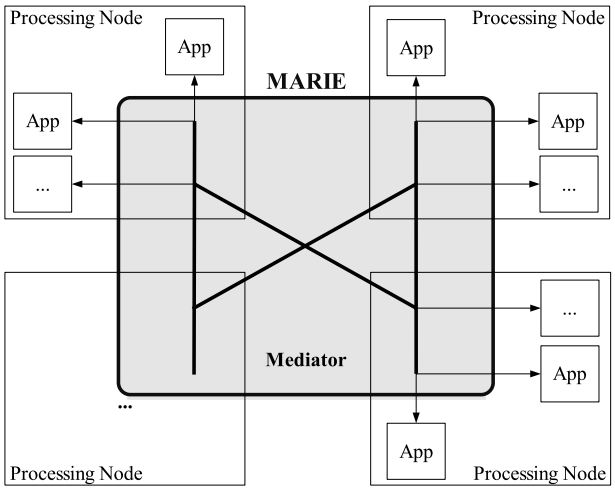

Fig. 2. MARIE's adaptation of the mediator pattern for distributed system

MARIE proposes another solution, illustrated in Figure 2, by adapting the mediator design pattern [13] for distributed systems. The mediator design pattern primarily creates a centralized control unit (named mediator) which interacts with each colleague (application) independently, knowing how to coordinates global interactions between colleagues to realize the desired system. Here are the five consequences identified for the mediator design pattern, adapted to the distributed mobile robotics context:

1) *It limits subclassing*. Changing overall system behavior following the mediator pattern means having to only change mediator's interactions between applications, leaving applications intact. This is a good consequence because otherwise it would probably be necessary to modify application source code to change behaviors, reducing application reusability.
2) *It decouples colleagues*. Having all applications linked to the mediator only promotes loose coupling between them. Applications do not need to be aware of others applications, limiting interactions only through the mediator. Using better decoupling between colleagues also allow to reuse mediator configurations and applications independently, and to easily switch applications offering the same services.
3) *It simplifies object protocols*. A mediator replaces many-to-many interactions with one-to-many interactions between mediator and its colleagues. In a distributed context, it means that each application can use its own communication interface as long as the mediator knows how to communicate with each communication interface. Application reusability is enhanced by being not limited to specific communication interface, and by getting compatibility issues handled by a centralized control unit instead of being the responsibility of each application to integrate.
4) *It abstracts how objects cooperate*. It can be easier to create an abstraction level over each application, with each seen as a service provider. This abstraction level can use robotic domain representation without influencing the functional level of each application.
5) *It centralizes control*. However, by centralizing control, the complexity of the mediator increases in order to manage adequately all interactions between applications and implement all of the system functionalities. This can lead to specific implementations that can be hard to understand, maintain and extend.

According to these consequences, we have identified four functional components (control and logic) necessary for the interaction of heterogeneous applications together via a centralized control unit based on the mediator design pattern. Figure 3 illustrates these functional components allowing the integration of distributed applications. They form the main elements composing the base of MARIE's communication framework. *Application Adapters* are responsible for sending service requests and communications from the centralized control unit to the applications, and vice versa, using an application proxy. Each application to be integrated must have its own *Application Adapter* that encapsulates communication mechanisms, the services it provides and the specific configurations it needs. The centralized control unit then communicates directly with the *Application Adapters* to interact with the applications, and

vice versa. *Communication Adapters* are responsible for translating information between different communication protocols and mechanisms. For example, they can provide an interaction mechanism that lets two applications using a pull communication mechanism exchange data correctly. They can also be responsible for creating a certain level of communication abstraction by providing communication tap points that hide data source from data sink. *Communication Managers* are responsible of creating and managing communication links between *Application Adapters* that need to be connected together. This means that they can be used to interconnect *Communication Adapters*. One *Communication Manager* must be available for each processing node present in the distributed system. *Application Managers* manage and control the entire system by coordinating system states, achieving system coherence and stability, and configuring and controlling all components available in the system. One *Application Manager* must be available for each processing node present in the distributed system.

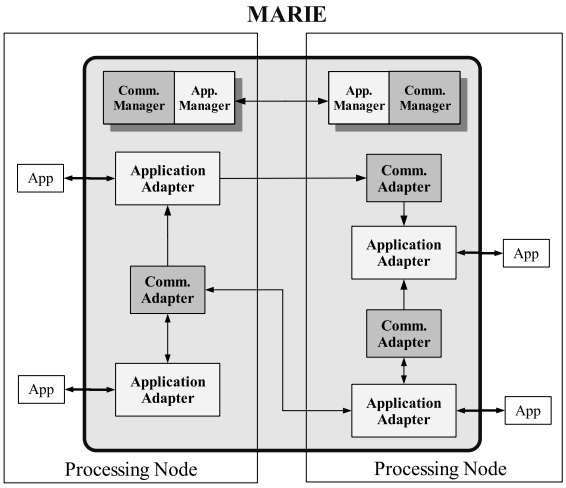

Fig. 3. Example of a distributed system with a detailed view of MARIE centralized control unit

Explicitly decoupling functionalities using functional components enhances modularity and reusability of those components in different integrated systems. By using a solid and generic communication framework, MARIE aims to create a very flexible system that will support a wide variety of applications, and let robotics developers build on top of MARIE's higher level of abstraction to design systems based on their own integration needs. In order to reduce development complexity associated with implementation of the centralized control unit, MARIE's solution is to have many development and debugging tools that will help create a stable and coherent system. MARIE's design also follows standardized software engineering methodologies to ensure good software design practices according to requirements in mobile robotics [14].

These advantages come with a cost. First, system performance can be affected by the overhead caused by the additional software for the functional components. Second, allowing to create systems using many heterogeneous communication interfaces and applications increase the complexity of making MARIE a stable and coherent system. Third, each application must have a clear method of interactions (through an API or communication links for example) that MARIE can use for integration. Finally, a more subtle but critical issue is in the way applications access system resources (drivers, memory, hardware, etc): if two independent applications tries to access the same resources at the same time or in an incoherent way, it can result in an unstable and unpredictable system. Controlling system stability can be difficult if integrated applications do not give access to how they manage system resources.

A. Prototypes

MARIE's implementation is done through implementations of prototypes with increasing levels of complexity and sophistication, with the objective in mind of not *reinventing the wheel* and study how different software elements can be used and integrated in MARIE.

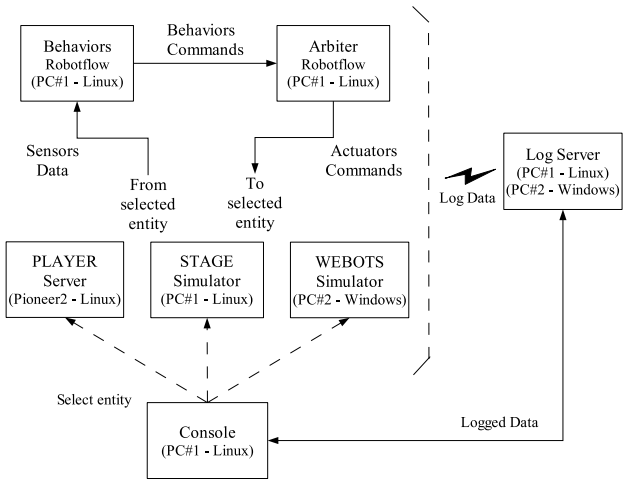

Fig. 4. MARIE's first prototype system

The first implementation of MARIE (http://marie.sourceforge.net/) was developed for proof of concept. It allows to change at run-time the robotic entity that is controlled using the same decision modules, as illustrated in Figure 4. Three entities are available, each one running on a different computer: a simulated robot using Stage on Linux, a simulated robot using Webots (http://www.cyberbotics.com/) on Windows, and a real Pioneer 2 robot running Player server on Linux. The decision modules consist of a behavior-based architecture using sonars to wander around while avoiding obstacles in the environment. The behavior-based architecture is divided in two independent applications: one for the behaviors, and the other for arbitration (subsumption). Both applications are implemented using RobotFlow. The *Behaviors* application receives sensors information from MARIE's centralized control unit without knowing what entity provides them. It then provides commands to MARIE's centralized control unit, which sends them to the

Arbitration application. The *Arbitration* application, like the behaviors application, does not know which module provides behaviors commands, since it receives them from centralized control unit. Once arbitration is done, the *Arbitration* application provides actuator commands back to MARIE's centralized control unit, which sends them to the selected entity. All these interactions are done asynchronously. Using this prototype, it is possible to change on the fly the entity controlled by the decision modules. The same implementation of decision modules can be validated on different entities, without having to re-implement it in multiple programming environments. Adding on-line a module to teleoperate the robot would be very easy: it would require just to add a new module implementing a virtual joystick that sends directly actuators commands, leaving everything else untouched. Although it is a simple implementation, MARIE's first prototype demonstrates many promising possibilities.

A second implementation integrating more of MARIE's concepts has been completed June 2004. It contains Application Adapters for Player, CARMEN and RobotFlow/FlowDesigner, which are used to build a semi-autonomous tele-operated mobile robot application. Navigation and localization are done with CARMEN; remote control and behavior control are done with RobotFlow/FlowDesigner; Player device abstraction allows to use the same system configuration to run in simulation (with Stage and Gazebo) and on a real robot (Magellan Pro). MARIE's functional components and communication layer are programmed in C++ using ACE (http://www.cs.wustl.edu/~schmidt/ACE.html) as a generic communication framework. By using ACE instead of more specialized communication systems, it is possible to implement multiple communication mechanisms and protocols, and support many communication systems used in robotic such as IPC (http://www-2.cs.cmu.edu/afs/cs/project/TCA/www/ipc/ipc.html), NIST/RCS (http://www.isd.mel.nist.gov/projects/rcslib/) and CORBA (http://www.corba.org/). A detailed description of this implementation is available at http://marie.sourceforge.net.

IV. Conclusion

In this document we have described two software development initiatives currently underway, aiming at facilitating code reusability for programming of autonomous mobile robots. Our goal is to make such tools robust, efficient, well-documented, and to share them with others to make the field progress. They are all available for free, under an open-source license.

In our view, programming tools for code reusability in mobile robotics are fundamental elements to optimize scientific breakthroughs in the field. They become a mean to communicate knowledge and implementation results through readable, reusable and well-documented code. They allow exchange of ideas, sharing of approaches, and communication of implementation results. In addition of having these exchanges through articles, conferences and workshops, the tools will make these exchanges through working code (source code under open source license, another way of communication, or through executable) and common working practices. It will also allow the realization of innovative scientific contributions, by making possible to integrate interesting capabilities on mobile robots in novel ways. In that regard creating these tools are more than just an engineering effort: it must be part of the scientific process of studying intelligence in autonomous systems. Our most fundamental hope is that tools such as RobotFlow and MARIE become interesting enough that others will join the initiatives in an effort that shows great potential in addressing the integration issues facing the emerging industry of mobile robotics.

Acknowledgment

F. Michaud holds the Canada Research Chair (CRC) in Mobile Robotics and Autonomous Intelligent Systems. This research is supported financially by CRC, the Natural Sciences and Engineering Research Council of Canada (NSERC), AUTO21st Network of Centre of Excellence and the Canadian Foundation for Innovation (CFI).

References

[1] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. RA-2, no. 1, pp. 14–23, March 1986.

[2] S. Hedberg, "IJCAI-03 conference highlights," *AI Magazine*, vol. 24, no. 4, pp. 9–12, 2003.

[3] F. Michaud, J. Audet, D. Létourneau, L. Lussier, C. Théberge-Turmel, and S. Caron, "Experiences with an autonomous robot attending the AAAI conference," *IEEE Intelligent Systems*, vol. 16, no. 5, pp. 23–29, 2001.

[4] J. S. Albus, "Outline for a theory of intelligence," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 21, no. 3, pp. 473–509, May/June 1991.

[5] F. Michaud, "EMIB – Computational architecture based on emotion and motivation for intentional selection and configuration of behaviour-producing modules," *Cognitive Science Quaterly, Special Issue on Desires, Goals, Intentions, and Values: Computational Architectures*, vol. 3-4, pp. 340–361, 2002.

[6] R. T. Vaughan, B. P. Gerkey, and A. Howard, "On device abstractions for portable, reusable robot code," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2003, pp. 2421–2427.

[7] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: The Carnegie Mellon navigation (CARMEN) toolkit," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2003, pp. 2436–2441.

[8] M. Hattig, I. Horswill, and J. Butler, "Roadmap for mobile robot specifications," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2003, pp. 2410–2414.

[9] Y. Zhao, "A model of computation with push and pull processing," Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Sciences, December 2003.

[10] E. Woo, B. A. MacDonald, and F. Trépanier, "Distributed mobile robot application infrastructure," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2003, pp. 1475–1480.

[11] H. Utz, S. Sablatnög, S. Enderle, and G. Kraetzschmar, "MIRO – Middleware for mobile robot applications," *IEEE Trans. on Robotics and Automation*, vol. 18, no. 4, pp. 493–497, 2002.

[12] I. A. D. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin, "CLARaty and challenges of developing interoperable robotic software," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2003, pp. 2428–2435.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[14] A. Orebäck and H. I. Christensen, "Evaluation of architectures for mobile robotics," *Autonomous Robots*, vol. 14, pp. 33–49, 2003.