Proceedings of the 2000 IEEE
International Conference on Robotics & Automation
San Francisco, CA • April 2000

Architecture, the Backbone of Robotic Systems

Ève Coste-Manière
CHIR Medical Robotics group
INRIA Sophia Antipolis
Eve.Coste-Maniere@sophia.inria.fr
www.inria.fr/chir

Reid Simmons
School of Computer Science
Carnegie Mellon University
reids@cs.cmu.edu

Abstract

Architectures form the backbone of complete robotic systems. The right choice of architecture can go a long way in facilitating the specification, implementation and validation of robotic systems. Conversely, of course, the wrong choice can make one's life miserable. We present some of the needs of robotic systems, describe some general classes of robot architectures, and discuss how different architectural styles can help in addressing those needs. The paper, like the field itself, is somewhat preliminary, yet it is hoped that it will provide guidance for those who use, or develop, robot architectures.

1 Introduction

There have been some remarkable and inspiring success stories for robotics over the past few years. For example, the surgeries performed with robot assistance, vehicles capable of autonomously driving on highways for hundreds of kilometers, the Sojourner rover exploring Mars, robots for hazardous waste cleanup, demining, agriculture, and autonomous museum tour-guides. Such robotic systems, working in challenging applications, rely upon theoretical progress allied to technical advances and the exponentially increasing power of computers.

A common feature of such systems is their complexity, which also grows apace and provides its own challenges. FRAMEWORKS FOR MANAGING THIS GROWING COMPLEXITY is the theme of this symposium on Architectures for Robot Control and Coordination. Modern robotic systems, which need concurrent embedded real-time performance, are typically too complex to be developed and operated using conventional programming techniques. Rather, *managing complexity demands frameworks and tools that embody well-defined concepts to enable the effective realization of systems to meet high-level goals.*

The term *robot architecture* encompasses several different notions. Of particular interest are architectural *structure* and *style*. Architectural structure refers to how a system is divided into subsystems, and how those subsystems interact. This is often represented by the traditional "boxes and arrows" diagrams. Architectural style refers to the computational concepts that underlie a given system. For instance, one system might use a publish-subscribe message passing style of communication, while another may use a more synchronous client-server approach.

All robotic systems embody *some* architectural structure and style (and often a single robot system uses several styles together). However, it is sometimes difficult to determine *post hoc* exactly what architecture was used. The architecture and the implementation are often intimately tied together, in a "build it and make it work" manner. This is unfortunate, as a well-conceived architecture can have many advantages in the specification, execution, and validation of robot systems.

An architecture should facilitate the development of robotic systems by providing beneficial constraints on design and implementation of the desired application, without being overly restrictive. While this criterion is easy to express, it is much harder to operationalize. In particular, different applications have different needs, which can be best satisfied with different architectures. While matching the right architecture to the application is much more of an art than a science, in this paper we endeavor to provide some guidelines and insights to guide that choice.

The paper is organized as follows: section 2 introduces some of the different architectural styles encountered in the literature. In sections 3, 4, and 5, we discuss how the development of complex robotic systems can be facilitated using control architectures, focusing on the specification, execution, and validation phases. Trends for the future of architectures conclude the paper.

0-7803-5886-4/00/$10.00© 2000 IEEE

2 Robot Architectures

Robot systems differ from other software applications in many ways. Foremost are the need to *achieve high-level, complex goals*, the need to *interact with a complex, often dynamic environment, while ensuring the system's own dynamics*, the need to *handle noise and uncertainty*, and the need to *be reactive to unexpected changes.* These needs influence how robotic systems are designed, how they operate, and how they are validated (and even what it *means* to be validated). In addition, many robots belong to the class of *critical systems* [7]. In such systems, errors during operation can have significant consequences and *system safety* issues play an essential role.

Most of the architectural styles described in the technical literature can be classified into three categories: hierarchical, behavioral, and hybrid. The hierarchical style adopts a top/down approach [3]. It highlights the supremacy of high-level control and restricts low-level horizontal communications. It has poor flexibility and has been adapted with difficulty to the control of new generation robots, which have to handle many sensors in reactive and reflex loops.

In contrast, the behavioral architectures adopt a bottom-up approach [8]. This style uses groups of software modules known as "behaviors" which run concurrently and interact through communication and through the environment. With this style, designing high-level control to achieve non trivial objectives is often difficult. Further, composition laws do not have enough semantics to allow safety issues to be easily considered.

Hybrid architectures are the most recent [6, 20, 19, 5, 22, 7, 1]. The hybrid style combines both reactive and deliberative control in a heterogeneous architecture. It facilitates the design of efficient low-level control with a connection to high-level reasoning. The connection between the two levels can be tricky, however, and must be carefully designed and implemented to provide the right mix of reactivity and deliberation.

We now turn to discussions of how the use of robot architectures can address the needs of robotic systems, focusing on the processes of system specification, runtime execution, and overall validation.

3 Specification

Perhaps the foremost issue in designing robotic systems is the *need to manage the complexity of interactions* – both interactions between the system and its environment, and interactions between individual components of the system.

One way of dealing with this complexity is through *modularity within a given structure* – overall system complexity can be reduced by decomposing it into smaller components with well-defined abstraction levels and interfaces between them. Architectural styles such as data flow support this by making upstream components independent from downstream components. This contrasts, for instance, with a composition style based on function calls, where the invoking modules depend on the return values of the callees. Similarly, publish-subscribe message passing encourages the use of independent modules [23], whereas a client-server style may increase dependency and lead to more complex interactions.

Often, system decomposition is hierarchical – modular components are themselves built with other modules. Architectures that explicitly support this type of decomposition can lead to more modular systems, which has a postive impact on the execution phase. However, while hierarchical decomposition of robotic systems is generally regarded as a "desirable quality", debates continue over the dimensions along which to decompose. For instance, the RCS architecture [2] advocates decomposition along a temporal dimension – each layer in the hierarchy operates at a characteristic frequency an order of magnitude slower than the layer below. Many architectures such as TCA [23], RAPs [13], PRS [16], Laas [1], ORCCAD [7], and Subsumption [8] advocate decomposition based on task abstraction, with more or less well defined semantics. In some situations, decomposition based on spatial abstraction may be more useful, such as when dealing with problems involving both local and global navigation. The main lesson is that different applications need to decompose problems in different ways, and architectures need to be flexible enough to accommodate different decomposition strategies.

Another way of dealing with complexity is to *provide expressive languages and tools.* Such languages and tools, when based on solid theories, can provide constructs that constrain design in certain ways, while hiding the complexity of the underlying concepts. Architectures should enable different languages and representations to be used when and where appropriate, integrating them seamlessly. For instance, typical robotic applications involve both continuous and discrete control [7]. Architectures should address the problem of bridging the gap between those two domains – finding some interface that respects both representations, yet facilitates information and control flow between them.

Researchers have developed many high-level languages tuned to particular robotic tasks. Languages for real-time behaviors include the Subsumption language [8], ALPHA [14], and Skills [6]. Other approaches are to provide graphical programming environments for specifying control strategies. Code is then automatically generated from the graphical descriptions. Architectures that use this technique include ControlShell [22], ORCCAD, and Labview.

The use of high-level languages is even more prevalent at the symbolic, discrete control level. In particular, languages for specifying task-level control are quite common. They include RAPs [13], TCA [23], PRS [16], Propice [1], ESL [15], MAESTRO [9] and TDL [24]. A significant distinction between these languages is whether it is self-contained, a language extension, or a library. A self-contained language, such as RAPS or PRS, is often not very expressive and difficult to integrate with other code unless abstract views are provided to facilitate connection between modules, such as in MAESTRO. A language extension is a superset of an existing language that includes explicit task-level control constructs. For instance, ESL is an extension of Lisp and TDL is an extension of C++. The advantage here is that it is easy to incorporate task-level control along with other computations, since it is all specified in the same language, and in the same basic way.

Finally, an important aspect of robotic systems is the detection and handling of exceptional conditions. Such situations tend to be numerous and can arise at any time, but with low probability. Architectures can help in specifying appropriate reaction strategies by distinguishing nominal from exceptional behavior, and by enabling developers to specify such behaviors in a modular and incremental fashion. In some architectures, such as Subsumption, there are no exceptions – every state is treated in the same way. In other architectures, exceptions are treated specially and are specified in a different fashion from "nominal" behavior. In particular, such architectures often have the notion of "jumping" to an exception handler and then returning back to resume normal operations after the exception has been handled. To support this, architectures such as ESL, ORCCAD, ControlShell, TCA and TDL allow named exceptions to be specified hierarchically. This enables designers to specify both specific exception handlers that are applicable to restricted situations, such as retrying a move, as well as more general exception handlers that have wider scope and more global effects, such as aborting the task altogether [12].

4 Execution

While architectures have an essential role in the design and specification of a robotic application, they also play a significant role in the *run-time execution of robotic software.* For robotic systems, run-time execution typically includes issues such as *real-time response, appropriate goal-directed behavior, and reliable reactivity to environmental changes.*

While all robotic systems must be reactive, some applications demand more in terms of strong real-time response. Some architectures enable designers to specify the frequency at which behaviors should run, and then schedule execution to meet the constraints. The CIRCA architecture does this by using an explicit scheduler in conjunction with a real-time operating system [20]. ORCCAD and ControlShell also provide run-time support for scheduling execution at different frequencies. In contrast, many behavior-based architectures, such as Subsumption and the skills of the 3T architecture, operate without any real-time guarantees. Basically, the hope is that response will be "fast enough" for the domain. Whether real-time guarantees are needed or "fast enough" is sufficient depends on the characteristics of the robotic application itself.

With respect to goal-directed behavior, architectures can provide mechanisms for managing tasks and behaviors, such as task decomposition and behavior arbitration, and for managing interactions between tasks and behaviors, such as resource management, multi-tasking and temporal sequencing. The idea is to provide constructs that are commonly needed for task-level control. This alleviates the implementer from the responsibility of ensuring that the code performs according to specifications.

The most basic task-management capability is facilities for concurrent execution, either by multi-tasking in a single process (such as ControlShell provides) or by a collection of distributed processes (such as provided in TCA). Some architectures (such as RAPs and the Skill Manager of 3T) manage concurrency internally, through agenda queues. However, these tend to be less general solutions and do not easily allow distribution across multiple processors.

In addition to supporting concurrency, many robot architectures provide more direct task management capabilities. For instance, behavioral-style architectures typically include functionality for arbitrating amongst concurrent, potentially conflicting, behaviors. Some architectures handle this by blocking the output of a subset of the behaviors, either through overrides, as in Subsumption, or by dynamically selecting which be-

haviors will run, as in 3T. Other architectures address this problem by combining the outputs of multiple behaviors to form a single output command, as in Aura [4] and DAMN [21]. The tradeoffs are that blocking outputs typically leads to more predictable and computationally efficient systems, but combining outputs tends to be more flexible, since all behaviors have a potential effect on the chosen output.

At the task-control level, the main problem addressed is how to schedule tasks. This typically involves specifying temporal constraints between tasks, such as "task A cannot run until task B is finished", "C must start 10 seconds after D starts", "E must terminate when F completes", etc. Many architectures provide languages (RAPs, ESL, TDL, MAESTRO, PRS, Propice) that support such constraints and run-time environments to enforce them. While they differ in detail, most operate by maintaining some sort of agenda queue as well as a hierarchical representation of the task decomposition. An added degree of expressiveness on top of the temporal constraints is provided when architectures enforce restrictions on resource utilization amongst tasks.

With respect to reliable reactive behaviors, an architecture can provide software support for monitoring the environment and invoking exception handlers, when appropriate. Clearly, the ability to detect and react in a reliable manner depends on the real-time capabilities of the system, but there is more to the management of such exceptional conditions. This includes support for cleanly terminating existing tasks, and if necessary invoking recovery strategies, then, if possible, resuming the original tasks.

Architectures may also provide run-time support for arbitrating between applicable recovery strategies, such as organizing and invoking exception handlers in a hierarchical fashion. This type of non-local control flow is difficult to program, so existing support is very beneficial. Many of the existing robot architectures implement the type of hierarchical "catch and throw" exception mechanism found in languages such as Lisp, Ada, C++, and Java. The differences are mainly that in the robot architectures, these capabilities are tightly integrated with other task-level execution constructs, such as task suspension or task termination.

5 Validation

By "validation" we mean *both testing and formal verification.* Testing includes both unit and system testing. Since complex robotic systems are typically designed and implemented in a modular fashion, it is important to be able to unit test a component before the entire system is completed. However, the response of a component typically depends on the behavior of other system components. Thus, testers often need to *stub out* components by replacing them with functionally identical, yet simpler modules in order to see how the component being tested reacts in that context. Architectural support for this capability includes the use of anonymous publish-subscribe messaging.

Also, in testing it is often desirable to replace the actual robot hardware with a simulation. Since the hardware and simulation typically run at different speeds, it is often desirable to have the architecture maintain an internal clock that differs from the real-time clock, in order to ensure that the *relative* timings of events is the same for both simulation and actual robot.

Architectures can also aid in testing (and debugging) by providing methods for instrumenting robotic systems that do not (unduly) affect real-time performance [26]. One such method is logging significant system events. For instance, ControlShell, through its Stethoscope tool, enables users to view internal variables of an executing real-time system. TCA enables run-time logging of all message traffic. Similarly, the Remote Agent architecture [19] has extensive support for logging events and state changes, where the logger runs as a background process so as not to interfere with other processing.

Still more useful is the visualization and/or analysis tools that operate over such log data. For instance, Stethoscope enables users to graphically display, in real-time, continuous variables. Tools developed for the Remote Agent allow users to visualize message traffic and task execution [25]. Further development of automatic analysis and diagnostic tools is definitely warranted.

Robotic systems typically have huge state spaces – much larger than can be tested by trial and error. Formal verification provides a mean of guaranteeing certain system properties, such as safety, liveness, and absence of resource conflicts. The basic idea is to formally model aspects of the robotic system's behavior and to determine if that behavior meets some given specifications. Architectures can facilitate this in two ways. First, the behavior of the architecture can be formally specified, which facilitates modeling the behavior of any robotic system which uses that architecture. Second, the architecture may provide languages that are constrained in a way that makes formal verification tractable. For instance, if an architecture represents behaviors using finite state automata, it may be tractable to guarantee liveness and safety for a system

built using that architecture.
For instance, aspects of the task executive component of the Remote Agent were modeled using temporal logic and verified using model checking [17]. ORCCAD promotes the use of the synchronous modeling of the discrete-event parts of the system in conjunction with the description of continuous time aspects. ORCCAD also uses TIMED-ARGOS and the KRONOS symbolic verification tool to model the quantitative temporal characteristics of real-time systems [11]. The systematization of the verification process makes such analyses accessible to non-specialists, immediately integrable in a complete programming environment and thus less error prone. Two main types of properties are identified: the first ones are related to critical operating issues such as *safety* and *liveness* properties; the second ones are operational properties related to the completion of the desired objective such as the *coherency of the specification with the requirements* of the *application*. In addition, verification methods can be used to create *abstract views* to help the user during the specification phase.

6 Future Trends

The area of robot architectures has seen much good research, and several commercial products exist that embed at least some of those ideas. However, there remains much more to be done. In particular, systems integration and debugging remain two big open areas for research. More work is needed to provide standards for components, communication, and system decomposition.
In particular, it is important to view these architectures not as monolithic entities, but as parts of an overall solution. In this view, the best aspects of different architectural styles need to be identified, isolated, and "packaged", so that they can seamlessly "play along" with other architectural styles. We are still far from having this. One reason is that the abstraction levels used in the existing architectures are far from being compatible. The community needs to define (and agree upon!) precise and eventually standardized "interfaces" between levels, independent of the underlying structures and styles.
In the area of system debugging, much work is needed to formalize different architectural styles. This is important so that developers can understand precisely what their system will be doing. Formalization also aids in automatic test generation and verification. More visualization and analysis tools need to be developed, so that users may track down and correct problems. In addition, it would be useful to develop general replay mechanisms that enable developers to record all relevant data from a given run and play it back in non-real time [26].
Another open problem is how to compare different architectures and architectural styles. Research papers on architectures are typically descriptive – describing what the system did and how it was organized – but rarely provide metrics that would enable readers to determine what effect, if any, the architecture itself had on system performance. While we offer no ready solutions to this problem, one observation is that, since architectures can address needs in specification, execution and validation, it is legitimate to compare architectures along any of those axes [10]. For instance, one might report on how a particular architectural style made the specification of a system easier, or how it provided increased guarantees of real-time performance, or how it reduced the time to field a validated system. It is important to develop (and use!) quantitative criteria for evaluation of integrated systems (c.f. evaluation measures of haptic devices [18]). Even if quantitative comparison is not possible, reporting on the whole development cycle, rather than just on the final product, can help in qualitatively assessing the effects of different architectural styles and structures.

7 Summary

This paper has tried to present some of the difficulties in developing complex robotic systems, and discussed how architectures can help alleviate some of those difficulties. While architectures alone cannot create complete working systems, good architectural styles and structures can go a long way in making the specification, execution, and validation easier and more reliable. We hope that in providing some observations on the problems and potential solutions, we can provide guidance to those who want to use, or develop, robot architectures for their systems.

References

[1] R. Alami, R. Chatila, S. Fleury, M. Herrb, F. Ingrand, M. Khatib, B. Morisset, P. Montarlier, and T. Siméon. Around the lab in 40 labs... In *IEEE International Conference on Robotics and Automation, ICRA'00*, San Francisco, 2000 (this issue).

[2] J. Albus. Outline for a theory of intelligence. *IEEE Transactions on Systems, Man and Cybernetics*, 21(3):473-509, May/June 1991.

[3] J. Albus, R. Lumia, and H. McCain. Hierarchical control of intelligent machines applied to space station telerobots. *Transactions on Aerospace and Electronic Systems*, 24:535–541, September 1988.

[4] R. C. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112, Aug. 1989.

[5] J. G. Bellingham and T. R. Consi. State configured layered control. In *1st Worshop on Mobile Robots for Subsea Environments*, pages 75–80, October 1990.

[6] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. A proven three-tiered architecture for programming autonomous robots. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2), 1997.

[7] J.-J. Borrelly, E. Coste-Maniere, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The orccad architecture. *International Journal of Robotics Research*, 17(4):338–359, April 1998.

[8] R. A. Brooks. A robust layered control system for mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[9] E. Coste-Manière and N. Turro. The MAESTRO language and its environment: Specification, validation and control of robotic missions. In *IEEE/RSJ Intl Conf on Intelligent Robots and Systems, IROS'97*, volume 2, pages 836–841, Grenoble, France, September 1997. Maestro WEB page:"http://www.inria.fr/icare/maestro/".

[10] E. Coste-Manière, H. H. Wang, S. M. Rock, V. Rigaud, A. Peuch, M. Perrier, and M. J. Lee. Joint evaluation of mission programming for underwater robots. In *IEEE Intl Conf on Robotics and Automation*, pages 2492–2497, April 1996.

[11] B. Espiau, K. Kapellos, and M. Jourdan. *Verification in Robotics: Why and How?* Robotics Research, the Seventh International Symposium, Giralt and Hirzinger editor, Springer Verlag, October 1995.

[12] J. L. Fernandez and R. Simmons. Robust execution monitoring for navigation plans. In *Proceedings International Conference on Intelligent Robotics and Systems*, Vancouver, Canada, Oct. 1998.

[13] R. J. Firby. An investigation into reactive planning in complex domains. In *Proc. National Conference on Artificial Intelligence*, pages 202–206, Seattle, WA, July 1987.

[14] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Tenth National Conference on Artificial Intelligence*, San Jose, CA, July 1992. AAAI.

[15] E. Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. In L. Pryor, editor, *Procs. of the AAAI Fall Symposium on Plan Execution*. AAAI Press, 1996.

[16] M. Georgeff and A. Lansky. Reactive reasoning and planning. In *Proc. National Conference on Artificial Intelligence*, pages 972–978, Seattle, WA, July 1987.

[17] M. Lowry, K. Havelund, and J. Penix. Verification and validation of AI systems that control deep-space spacecraft. In *Proceedings Tenth International Symposium on Methodologies for Intelligent Systems*. Springer-Verlag Lecture Notes in Artificial Intelligence, Vol. 1325, Oct. 1997.

[18] M. Moreyra and B. Hannaford. A practical measure of dynamic response of haptic devices. In *IEEE International Conference on Robotics and Automation, ICRA'98*, pages 369–374, Leuven, Belgium, 1998.

[19] N. Muscettola, P. Nayak, B. Pell, and B. Williams. The New Millenium Remote Agent: To boldly go where no AI system has gone before. In *Proc of the 15th Intl Joint Conf on Artificial Intelligence*, 1997. Extended Abstract of Invited Talk.

[20] D. Musliner, E. Durfee, and K. Shin. CIRCA: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1993.

[21] J. Rosenblatt and C. Thorpe. Combining multiple goals in a behavior-based architecture. In *Proc. Conference on Intelligent Robots and Systems*, Pittsburgh, PA, Auguest 1995.

[22] S. Schneider, V. Chen, G. Pardo-Castellote, and H. Wang. Controlshell: A software architecture for complex electro-mechanical systems. *International Journal of Robotics Research, special issue on Integrated Architectures for Robot Control and Programming*, Spring 1998.

[23] R. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, Feb. 1994.

[24] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proc. International Conference on Intelligent Robots and Systems*, Vancouver Canada, Oct. 1998.

[25] R. Simmons and G. Whelan. Visualization tools for validating software of autonomous spacecraft. In *Proc. of i-SAIRAS*, Tokyo, Japan, July 1997.

[26] J. Tsai and S. Yang, editors. *Monitoring and Debugging of Distributed Real-Time Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1995.