1998 IEEE International Conference on Robotics and Automation, Vol. I, pp. 844-849, Leuven, Belgium, May *16-20, 1998*

Message-based evaluation for high-level robot control

Christopher Lee*[†] Yangsheng Xu[†‡]
[†]Robotics Institute, Carnegie Mellon University
Pittsburgh, PA, USA
[‡]Department of Mechanical and Automation Engineering
The Chinese University of Hong Kong, Hong Kong

Abstract

In this paper, we present a method for high-level control of robots whose low-level software is based on dynamically reconfigurable, reusable real-time software modules. Our approach is to use an embedded interpreter for a general-purpose programming language to direct the operation of the low-level modules toward meeting the task-level goals of the robot. To this end, we present RSK, a virtual-machine kernel implementing a Scheme interpreter capable of hard real-time operation, and employing a method of code execution we call "message-based *evaluation" (MBE). MBE* is a novel combination of a traditional code execution model and a message-passing architecture, which simplifies the process of writing code for managing the robot's reconfigurable subsystem.

1 Dynamically reconfigurable real-time software

A major goal of real-time operating systems like Chimera is to enable sensor-based control applications to be built from libraries of reusable software modules. For this purpose, they provide standard interface specifications for implementing reusable real-time software modules, and a library of functions for building and using configurations of these modules [1]. A well-written and debugged library of real-time modules thus facilitates rapid development of reliable sensor-based control systems. In Chimera, these modules or "port-based objects," typically cycle at some fixed frequency and communicate their inputs and outputs through a global state-variable table. A typical configuration of real-time modules for controlling a robot manipulator arm is shown in Figure 1.

A real-time software module is reusable only if it is sufficiently independent of the specific details of the different applications for which it is used. Therefore, an essential focus of developing reconfigurable software is keeping task-level details out of the reusable modules. For example, a PID control module should not care whether it is controlling a joint-angle in a robot-arm, a Cartesian tool-coordinate, or a feature-coordinate in a visual-servoing process. As a result, reusable software modules are most useful for the lowest-level tasks within a robot software architecture—those which do not require explicit knowledge of the task-level details of the robot's operation.

*Supported by a DOE Fellowship in Integrated Manufacturing

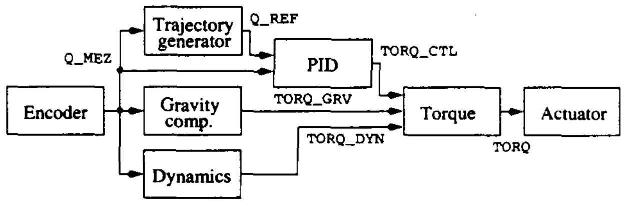

Figure 1: Example configuration of real-time modules

In robotic applications, this specialization results in a need for a higher-level layer of the software architecture which can direct the use of the reusable modules for the purpose of satisfying the robot's task-level requirements. This layer typically initializes all the reusable modules when the robot is booted, sends messages to modules telling them to modify their working parameters (e.g. adjusting controller gains, or sending via-points to a trajectory-generator module), and receives messages from modules to learn of significant events in the operation of the robot (e.g. significant qualitative changes in the readings of robot sensors). Most importantly, when the qualitative nature of the robot's task changes significantly, the high-level layer of the architecture must change the configuration of reusable-modules to match the needs of the task. For instance, when a manipulator arm is moving in a Cartesian control mode and contact is sensed at the end-effector, the robot should switch to a force-control or impedance control configuration. Such a "dynamic reconfiguration" typically involves turning off some modules and turning others on. This must be done on the fly, changing the active control law without disturbing the timing or effectiveness of the overall system. In cases such as the switch from Cartesian to compliant control, this must be performed without delay to avoid unacceptable forces at the end-effector. It is thus essential for the safety of the robot and its surroundings that the high-level controller react to important task-level events in hard real-time.

Several strategies have previously been used for managing such dynamically reconfigurable subsystems, including on-line state machines, and separate high-level programs running on host workstations. In several Chimera-based robot architectures [2, 3], the high-level process reconfigures the real-time subsystem based on an on-line state machine interpreter responding to messages sent from modules in the reconfigurable subsystem. ControlShell [4] for the VxWorks operating system also uses a state machine for managing dynamically reconfigurable real-time subsystems. Implementing interpreters for state machines is fairly straightforward, and state-machines are well understood and amenable to design through graphical user interfaces. Synchronous languages such as Esterel [5] which is used in the ORCCAD [6] robot application development system, may also be useful for this purpose.

Another approach for managing reconfigurable subsystems of real-time control modules is represented by Onika, a visual programming environment for designing control systems as configurations of modules, and for controlling the reconfiguration of these control systems during execution of Chimera applications [7]. Onika's visual programming language is limited in terms of the algorithms it can represent, however, and because it manages dynamic reconfiguration of the low-level Chimera modules from a non real-time workstation, it is inappropriate for managing reconfigurations which must occur in hard real-time.

In this paper, we present the approach of using an embedded interpreter for a general-purpose programming language for high-level control of reconfigurable subsystems. This approach has a number of advantages: (a) sufficient expressive capability for most high-level task specifications can be guaranteed by using a suitably powerful interpreted language, (b) a general-purpose programming language can specify robot-tasks using traditional structured-programming or object-oriented methods, (c) hard real-time response times to events can be achieved through careful implementation of the embedded interpreter, and (d) an interpreted language (in source-code form or compiled to virtual-machine code) is a convenient way for remote operators to send general-purpose commands to a robot while it is running (e.g. for remote teleoperation).

Development of a robot architecture for the Dual-use Mobile Detachable Manipulator, $(DM)^2$, motivated our adoption of this strategy. $(DM)^2$, shown in Figure 2, is a mobile robot consisting of a mobile base and a detachable manipulator ann [8]. The manipulator is a symmetric 5-DOF ann with a gripper at each end, and may either grasp the mobile base with one gripper to become a mobile manipulator system, or detach from the base and walk hand-over hand by grasping special handles with its grippers. The software for this robot is built upon the Chimera 3.2 operating system, It uses configurations of real-time modules for controlling the motion of the mobile base and manipulator arm, and requires the ability to dynamically change these configurations as the robot changes hardware configurations (i.e. from mobile manipulator to walking arm) or performs different tasks (e.g. switching from walking to grasping and then lifting an object).

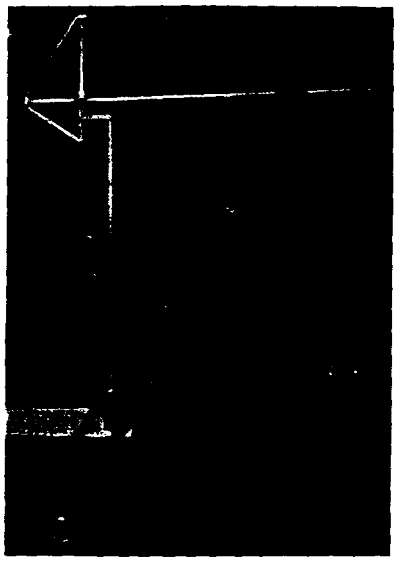
Figure 2: $(DM)^2$

$(DM)^2$ requires high-level software which can not only perform the necessary reconfigurations of its low-level software in hard real-time, but which is intelligent enough to manage the overall operation of a mobile robot. Some examples of what the high-level software for $(DM)^2$ must do include: using an internal map of its environment to keep track of the angle of inclination of the surface the arm is walking on (to adjust the gravity vector for calculating gravity-compensation torques in the joints); allowing multiple attempts at grasping handles or the mobile base before admitting failure (possibly perturbing the set-point slightly each time); switching between different controllers during different subtasks (i.e. using an adaptive controller when picking-up an object of unknown mass); following procedural descriptions of arm motions for walking and mobile base movements from on-line or off-line path-planners: and accepting commands from a remote operator. In all these cases, we need to specify alternative actions to be taken if any individual operation fails.

In developing a software architecture for $(DM)^2$, we initially built an interpreter for a simple, custom-designed scripting language to manage the dynamic reconfigurations of the low-level real-time subsystem [9]. After some experience programming this system, however, we decided that a more powerful, general-purpose language would be better suited to our needs and chose Scheme. Scheme is a Lisp dialect with a concise specification for which small, efficient interpreters can be written. It is also a powerful language commonly used for writing artificial-intelligence algorithms and for programming in a functional style [10]. It is simple to use for writing descriptions of the operations necessary for high-level control of our robot, and we felt it easier to write more complex approaches to such task-level needs with a general-purpose programming language than with a state-machine description. Scheme, in particular, has continuations as first-class objects, and these play an important role in our method of executing high-level robot code (as discussed in Section 2). We thus developed the Robot Scheme Kernel **(RSK)**, which can respond to events in real-time, and which works cooperatively with real-time code written in a system programming language (such as C) within an existing multi-threaded, multiprocessor robot architecture. RSK satisfies these requirements through real-time memory management strategies and a novel execution model which is designed specifically for controlling robots.

2 Message-based evaluation

Task-based management for supervision and dynamic reconfiguration of the low-level subsystem requires a very different style of coding than that for which traditional system programming languages are designed. Two of the main challenges in writing such high-level robot code are that (a) there may be a high degree of functional parallelism in the normal operation of the robot's hardware, and (b) its operations involve physical processes that occur much more slowly than the elementary software operations which are used to manage them. General-purpose programming languages (especially system programming languages such as C), excel at data manipulation and logic-based control of execution flow. However, they are less appropriate for specifying temporal relationships between subexpressions such as those demonstrated by the code in Figure 3 (the details of which will be discussed later in this section). RSK executes code like this by employing a method we call ***message-based evaluation*** (MBE), which is designed to allow the structure of high-level robot control code to reflect the structure of the tasks whose execution it supervises.

```
; Move arm in direction dir with speed speed until
; contact is detected at the end-effect06 but stop
; if the motion lasts longer than 5 seconds.
(define (move-to-contact dir speed)
 (race
  (lambda ()
   (move-arm dir speed))
  (lambda ()
   (detect-contact) 'contact)
  (lambda ()
   (pause 5.0) 'no-contact)))
; If moving the arm achieves contact, switch to
; a configurationfor compliant control
(case (move-to-contact <down> <slow>)
 ((contact) (start-compliant-control))
 (else (GUI:error
        "Contact was not detected")))
```

Figure 3: Robot code for a guarded move

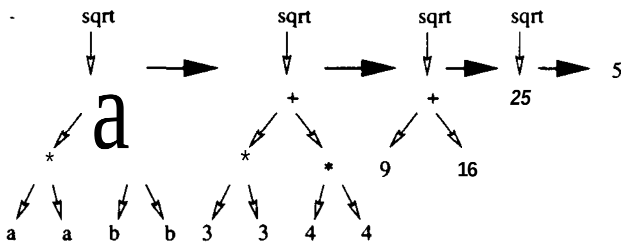

Figure 4: Evaluation by graph reduction

In functional programming languages, the evaluation of an expression is often modeled as a process of "graph-reduction." An expression is an acyclic graph, and evaluation is a process whereby the graph is simplified in a step-by-step fashion to a single node representing the value of the expression. For instance, the evaluation of the expression $\sqrt{a^2+b^2}$, coded in Scheme as **(sqrt (+ (* a a) (* b b)))**, could be represented (for $a = 3, b = 4$) as the graph simplification shown in Figure 4.

This evaluation could be accomplished by a conventional stack-based computation such as (PUSH a, PUSH a, APPLY '$*$', PUSH b, PUSH b, APPLY '$*$', APPLY '$+$', APPLY 'sqrt'). Such a method is efficient for conventional computers and does not require a literal graph-based representation of the expression to work. A very different evaluation method could also be used—one based on message-passing between nodes of an explicit graph representation of the expression. In such a method, each node of the graph is represented by an object which may receive messages from and send messages to its parent and child nodes, and which knows how to compute its own value when given the value of each of its child nodes. The evaluation process is triggered by sending a message to the root node commanding it to evaluate the graph. The evaluation occurs through each node implementing the following procedure:

1. For each child node (if you have any), send a message to that child telling it to evaluate itself and to reply with a message containing the result of this evaluation.

2. Once all child nodes have replied, evaluate yourself and return the result.

In the case of the expression graphed in Figure **4,** the "variable nodes" *a* and *b* immediately look-up their values and send them to the "multiplication nodes" which in turn calculate their products and send these to the "addition node", which sends the sum of these products to the "square-root node", which returns the final result *(5).*

Although this method is obviously inefficient for the example computation, it has some interesting characteristics:

- For each node in the graph which has more than one child node, the order of evaluation of the child nodes is unspecified, and the child nodes could even compute their results in parallel.

- If the underlying messaging system were to support the necessary communication (see Section 3), each node could be on any CPU of a multiprocessor system or even on a separate computer. The evaluation process would be exactly the same in these cases.

If we extend the evaluation process so that each node controls when and if each of its child nodes is evaluated, we can build expressions which explicitly represent temporal (as well as logical) relationships between the execution of their subexpressions. We can, for instance, implement "async nodes'' which evaluate their child nodes sequentially (equivalent to the stack-based evaluation strategy); "conditional nodes" which evaluate some child nodes depending on the results returned by others (e.g. a node implementing an "if-then" operation); "sync nodes" which evaluate all their child nodes in parallel: and even "race nodes" which tell all their child nodes to evaluate themselves and return the result of the first child node to finish (aborting the evaluation of the other child nodes). Figure 3 shows an example of the use of a race node. Writing an equivalent expression in a system programming language would be much more difficult. typically requiring the use of explicit polling mechanisms, or a combination of a state-machine description and a state-machine implementation.

MBE combines a standard expression evaluation technique, similar to the stack-based method, with an implementation of the message-passing method. This results in an interpreter with both the efficiency of the standard method and the ability of the message-passing method for executing code representing explicit temporal relationships between subexpressions. **MBE** extends the model of the message-passing evaluation architecture by specifying that a node must either return its result to its parent node "immediately" or tell its parent node that it is "not done yet." The interpreter can thus use the standard evaluation method to evaluate an expression until a call to a function within the expression raises a "not-done''exception. When this exception is raised, the interpreter creates a child-node object representing the incomplete function call, and a parent-node object representing the remainder of the (as yet unfinished) computation. At the appropriate time, the child-node can cause the interpreter to resume the evaluation by sending its value in a message to its parent node. Such an event is typically triggered by a message sent from another branch of the evaluation tree or from a low-level module indicating that a gripper has been closed, a manipulator motion completed, or an obstacle detected.

When **MBE** switches from its standard evaluation strategy to its message-passing strategy, it need only create a single object to represent the remainder of the incompletely reduced graph rather than an explicit graph representation of the entire unfinished computation. This is because the object representing the graph above the child node contains a *continuation*. A continuation is a representation of the entire default course of a given computation, and as such is a full representation of the incompletely reduced graph of an expression. Continuations are typically used to implement co-routines, threading, and throw-catch style exception handling. In Scheme, continuations are first class objects. If a Scheme implementation is based on a "continuation chain," or a chain of "incomplete continuation" objects, rather than on a C-style stack, then creating such a continuation object is roughly equivalent in speed and memory cost to a function call. This allows the switching between the two styles of evaluation to be very efficient. Thus, **MBE** works quickly and cheaply for interpreting the Scheme language.

Note that while this model of evaluation allows parallel operations to be represented by multiple "not-done'' child-nodes below a parent node, this does not mean that MBE itself is performing any kind of multi-threaded operation. The "not-done" nodes represent processes occurring outside the main thread of the interpreter, typically in the reconfigurable modules. These processes may include things such as grippers opening and closing. and manipulator arms executing motion commands. A "not-done'' node may also be waiting for command-messages from the teleoperation console, or for a panic-message from anywhere in the robot architecture indicating that the robot needs an immediate shutdown for safety reasons. Thus, RSK is usually waiting for messages rather than running Scheme code, and its job is to react to these messages without delay. The state of the current evaluation tree indicates what actions the high-level system should take when it receives messages from the reusable modules or the host workstation.

3 Messaging infrastructure

In discussing the process of code evaluation by message-passing, we noted that if the underlying messaging system were to support the necessary communication, each node in the graph of the expression could be evaluated on a different CPU or computer. This motivated us to design a system for message-passing that is optimized for speed in the local delivery of messages, but which is also able to use whatever operating-system communication mechanisms are available for passing messages to and from remote domains.

Each RSK message contains a 'To" address for directing message delivery and a "From" address so that it may be easily replied to. Each address has a local component and a domain name. Two nodes are in the same "domain" if they are able to use valid memory pointers to one another for their local addresses. Message passing within a domain is thus an inexpensive operation, consisting of adding a message to the priority queue "in-box" of its recipient, and adding the recipient node to a prioritized list of nodes in the domain which have pending messages. A function written in C may also register a local address with the messaging system, allowing it to receive messages via a call-back mechanism. Among other things, this allows high-level Scheme code to send parameters such as gains and via-points to low-level control modules in a reconfigurable subsystem. If a message is sent to a node in a different domain, the message is converted from its local representation (a Scheme object) to a binary representation which may be sent via operating-system communication mechanisms to a process in the appropriate destination. This remote process converts the message back to its original form and performs the local delivery. Since Chimera is a multiprocessor operating system, and because it is hosted by a UNIX workstation, RSK's messaging mechanism enables it to deliver commands and information between CPUs of the real-time computer, and to any computer on the host-workstation's network (e.g. the Internet).

The messaging infrastructure thus allows RSK interpreters running in each CPU of the real-time computer to cooperate with one another, and provides a mechanism for cooperation between the high-level control process of the real-time computer and off-line resources such as remote teleoperation consoles and planners. An additional benefit is that this communication mechanism allows RSK to offload some of the work of Scheme interpretation to the host workstation. The host workstation can parse Scheme code and compile it to a virtual machine-code representation (the compiler is actually a Scheme program running on a UNIX implementation of RSK), and then send a message containing the resulting virtual-machine code to the real-time computer for execution. This allows the high-level process on the real-time computer to focus its resources on managing the operation of the robot rather than on parsing and compiling Scheme code.

4 Memory management

In Lisp-like languages, explicit management of dynamically allocated memory is infeasible. These languages rely on "garbage collection", which is a mechanism for automatically determining what memory a program is no longer using and recycling it for other use. Because this determination involves a global analysis of the interpreter's memory pool, most commonly used garbage collection algorithms require that the interpreter be stopped during the collection in ways which are incompatible with real-time operation. Rees and Donald [11] use an embedded Scheme interpreter for control of small mobile robots. This interpreter is appropriate for their work, but garbage collection pauses make it inappropriate for use in real-time applications. If a robot were unable to react quickly to end-effector contact during a guarded move because the high-level was paused for garbage collection, damage to the robot and its environment could result.

In the initial version of RSK, we addressed this problem by simply using reference-counting for garbage collection. This is a local strategy rather than a global strategy for analysis of memory usage, and can be done in small increments which preserve the overall responsiveness of the interpreter. Although this strategy can be made to work for managing the interpreter's own use of data structures [12], and though this strategy has been successful for controlling our robot without memory leaks, reference-counting cannot reclaim data-structures which point to themselves even when they are not referenced by any data structures in use by the interpreter. Fortunately, there are now methods which allow garbage collectors to run efficiently on stock hardware in hard real-time [13], in addition to those which run on specialized hardware [14]. To allow programmers to use RSK for code which may use cyclic data structures, and to simplify the interpreter, we are implementing a real-time garbage collector based on the "write-barrier" strategy used in Wilson and Johnstone's real-time collector [15].

5 Conclusion

We have described dynamically reconfigurable subsystems for sensor-based control of robot systems, and presented the Robot Scheme Kernel (RSK), an embedded Scheme interpreter designed for high-level management of these subsystems. To allow RSK to evaluate Scheme expressions which represent temporal relationships between their subexpressions, we have developed "message-based evaluation" (MBE). MBE allows the structure of high-level robot control code to reflect the structure of the robot's intended task performance. The messaging infrastructure underlying MBE helps to simplify operation in multiprocessor environments, providing a mechanism for the robot to interact cooperatively with remote processes such as teleoperation consoles and off-line planners. Real-time garbage collection strategies allow RSK to respond in hard real-time to important events during the course of robot operation. Future work will focus on a more formal characterization of this method and its use in robot systems, and a more complete comparison between it and other currently available methods for high-level robot control.

References

[1] D. Stewart, R. Volpe, and P. Khosla, "Integration of real-time software modules for reconfigurable sensor-based control systems," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 325–332, 1992.

[2] J. D. Morrow, *Sensorimotor primitives for programming robotic assembly skills.* PhD thesis, Robotics Institute, Carnegie Mellon University, April 1997.

[3] A. Douglas and Y. Xu, "Real-time shared control system for space telerobotics," *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 13, pp. 247–62, July 1995.

[4] S. Schneider, V. Chen, J. Steele, and G. Pardo-Castellote, "The ControlShell component-based real-time programming system, and its application to the Marsokhod Martian rover," in *ACM SIGPLAN 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, vol. 30 of *SIGPLAN Notices*, pp. 146–55, June 1995.

[5] F. Boussinot and R. D. Simone, 'The Esterel language," *Proceedings of the IEEE*, vol. 79, pp. 1293–1304, 1991.

[6] D. Simon, B. Espiau, K. Kapellos, and R. Pissard-Gibollet, "ORCCAD: software engineering for real-time robotics; a technical insight," *Robotica*, vol. 15, no. 1, pp. 111–5, 1997.

[7] M. Gertz. D. Stewart, and P. Khosla, "A software architecture-based human-machine interface for reconfigurable sensor-based control systems," in *Proceedings of 8th IEEE International Symposium on Intelligent Control*, (Chicago, IL), pp. 75–80. IEEE, August 1993.

[8] Y. Xu, C. Lee, and H. B. Brown, Jr., "A separable combination of wheeled rover and arm mechanism: $(DM)^2$," in *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*. vol. 3, pp. 2383–8, IEEE, April 1996.

[9] C. Lee and Y. Xu, "$(DM)^2$: A modular solution for robotic lunar missions," *International Journal of Space Technology*, vol. 16, no. 1, pp. 49–58. 1996.

[10] H. Abelson *et al.*, "Revised^4 report on the algorithmic language Scheme," *ACM Lisp Pointers IV*, vol. 4, July-September 1991.

[11] J. Rees and B. Donald, "Program mobile robots in Scheme." in *Proceedings of 1992 IEEE International Conference on Robotics and Automation*, (Nice, France), pp. 2681–8, IEEE, May 1992.

[12] D. Friedman and D. Wise, "Reference counting can manage the circular invironments of mutual recursion." *Information Processing Letters*, vol. 8, pp. 41-45, Janary 1979.

[13] P. Wilson, "Uniprocessor garbage collection techniques," in *International Workshop on Memory Management*, no. 637 in Springer-Verlag Lecture Notes in Computer Science, (St. Malo, France), September 1992.

[14] K. Nilsen, "Reliable real-time garbage collection in C++," *Computing Systems*, vol. 7, no. 4, pp. 467–504, 1994.

[15] P. Wilson and M. Johnstone, "Real-time non-copying garbage collection," in *ACM OOPSLA Workshop on Memory Management and Garbage Collection*, (Washington D.C.), ACM, September 1993.