Software Infrastructure for an Autonomous Ground Vehicle

Matthew McNaughton[*] , Christopher R. Baker[*], Tugrul Galatali[†], Bryan Salesky[‡] ,
Christopher Urmson[§] , and Jason Ziglar[¶]
Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213

The Defense Advanced Research Projects Agency Urban Challenge required robots to drive 60 miles on suburban roads while following the rules of the road in interactions with human drivers and other robots. Tartan Racing's Boss won the competition, completing the course in just over 4 h. This paper describes the software infrastructure developed by the team to support the perception, planning, behavior generation, and other artificial intelligence components of Boss. We discuss the organizing principles of the infrastructure, as well as details of the operator interface, interprocess communications, data logging, system configuration, process management, and task framework, with attention to the requirements that led to the design. We identify the requirements as valuable reusable artifacts of the development process.

I.   Introduction

IN the Defense Advanced Research Projects Agency (DARPA) Urban Challenge of November 3, 2007, 11 autonomous cars competed to drive 60 miles in under 6 h on suburban roads, interacting with each other and with human-driven vehicles. Three robots crossed the finish line without requiring human intervention. "Boss", entered by Carnegie Mellon's Tartan Racing team finished with the lowest time.

This paper illustrates the software infrastructure of the Tartan Racing Urban Challenge System (TRUCS) as a positive example of how the software infrastructure for a successful mobile robot system may be built. Our presentation is organized around seven key pieces of the infrastructure. As shown in Fig. 1, these are the operator interface, the interprocess communications system, the process management system, the data-logging system, the configuration system, and the task framework and data input and output libraries. The seventh piece is not a distinct component, but rather the organizing concepts of the system that guide the design and integration of the parts. We hope the reader will take away some lessons learned in the design of these pieces and the organizing concepts that were found to be effective.

TRUCS grew out of lessons learned through fielding the Red Team vehicles in the DARPA Grand Challenge races of 2004 [1] and 2005 [2], as well as other systems fielded in the past by the team members. The literature does not have many in-depth analyses of software infrastructure for mobile robot systems, aside from inter-process communications (IPC) packages [3–7]. The study of mobile robot software infrastructure is important because a well-crafted infrastructure has the potential to speed the entire robot development project by enabling the higher

* Graduate Student, Robotics Institute, mmcnaugh@ri.cmu.edu, tallbaker@cmu.edu
† Visiting Scholar, Robotics Institute, tugrul@galatali.com
‡ NREC Commercialization Specialist, Robotics Institute, bsalesky@rec.ri.cmu.edu
§ Director of Technology for Urban Grand Challenge, Robotics Institute, curmson@ri.cmu.edu
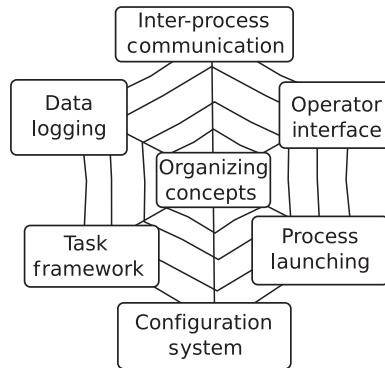¶ NREC Robotics Engineer, Robotics Institute, jpz@cmu.edu

Fig. 1 Seven key pieces of the software infrastructure discussed in this paper.

level architecture. The higher level architecture includes planning, perception, and world modeling algorithms that are carefully designed to carry out the functional task. They are implemented atop a software infrastructure that, if done well, lets their respective developers decouple their efforts from the overall development effort as much as possible. Designing the infrastructure with close attention to the requirements on the system can make the difference between success and failure.

A long-standing goal in the robotics community is to create software components that can be reused by "plugging" them together to create software for new robotic platforms. Part of the motivation for this is the great expense of developing software, especially software for robots. However, it is difficult to realize savings from the opportunistic re-use of software components [8]. Components are not the only reusable artifact of a development project. Requirements, architectural design, analyses, and test plans are a few of the many reusable artifacts created during system development [8]. If all of the requirements of the software system could be anticipated in detail, it would be easy to construct it in one fell swoop of development. To the extent that the same requirements apply to robotics projects, they should be mined for reuse opportunities. Thus, the detailed design of the system components of past projects, as opposed to the code itself, is also a useful resource.

The organization of this paper is as follows. In Sec. II we survey related work in software infrastructure and re-use for mobile robot systems. We also look at work in infrastructure and reuse for other kinds of robot systems, as well as guidance from the field of software engineering on how to approach architectural design and reuse in software systems. Section III discusses the system software qualities and organizing concepts embodied in TRUCS, such as repeatability, flexibility, and statelessness. These qualities are not directly the property of any one module, but guided the development effort towards a principled overall design. Section IV describes the requirements on and implementation of an operator interface, such as the developer's need for diagnostics. Section V discusses why TRUCS was a distributed system, and covers the design of the interprocess communication system devised to enable efficient communication between the dozens of tasks running on multiple computers in the robot. Section VI covers the process launching and management system used to coordinate the startup, monitoring, and shutdown of the tasks responsible for carrying out a mission. Section VII describes the data logging system for retaining and reviewing data generated during test runs. We discuss the organization of the software configuration system in Sec. VIII. TRUCS required complex configuration data for the many sensors, tasks, missions, and hardware platforms the system ran on. Section IX discusses the task framework and data input and output libraries that support the common functions of all tasks. We conclude with a discussion and considerations for future work in Sec. X.

II.   Related Work

In this section we survey efforts to describe, disseminate and re-use robotic software systems. There are several published full-scale software frameworks for robotic systems [7,9–16]. Miro [9] is a task and communications library with device abstractions for wheeled robots. It has been used successfully by a research group unrelated to its creators to construct an outdoor mobile robot [17]. Player/Stage/Gazebo [7] offers a device abstraction interface, communications layer, and simulation framework for wheeled mobile robots. The CLARAty [10] framework from NASA is a

framework for planetary rover software, including motion planning algorithms for common rover platforms. CAR-MEN [16] is a task framework and component library for indoor wheeled robots and SLAM applications. ModUtils [18,19] is a component architecture designed to isolate components from any knowledge of the components they are connected to and enable quick reconfiguration of components into different architectures in the AI view. The Microsoft Robotic Studio [12] is a new contribution of software development tools into the hobbyist market.

There are a few evaluations of these software packages in the literature. Orebäck and Christensen [20] offer an in-depth analysis and evaluation of several architectures. They composed a set of reference requirements for a robot application and implemented three software systems using different robot architectures on the same robot platform. The architectures compared were Saphira [11], TeamBots [14], and BERRA [21]. As a result of their experiments they formulated several design guidelines: that representations of sensor data and map data were an important but overlooked aspect of robot architecture; that modules should be individually addressable from the outside rather than mediating every command through a central deliberative component; that user interface, mission planner, and route planner should not be combined into a single component.

Broten and Mackay [17] document their experiences in selecting an architecture off-the-shelf for use in developing a nontrivial fielded mobile robot system. They compared Player [7], Miro [9], and CARMEN [16] in detail. The platforms were compared on the basis of documentation, available tutorials, support for multiple platforms, available sensor drivers, localization and mapping capabilities, and simulation capabilities. CARMEN was rejected for its lack of documentation at the time, although this improved by the time they published their paper. They found Player simple to use but lacking in multiprocessing capabilities. Miro was selected for its powerful set of abilities. They found that the greatest barriers to reuse of robot frameworks was software complexity, poor documentation, and rigid system architecture.

Robotic software systems of significant size are almost always distributed systems. A few reasons are that the system may have multiple communicating robots, have CPU demands too high for one computer, or need to firewall between soft real-time and hard real-time requirements by putting them on separate computers. The literature on distributed systems outside of robotics is extensive. Waldo et al. [22] describe the essential differences between local and distributed computing. More specifically, robots are typically distributed real-time systems. Tsai et al. [23] describe the major challenges and some solutions for this type of system, although they give most of their attention to the development of custom computing hardware to support the demands of real-time response, which is out of scope for many mobile robotics applications, including TRUCS.

The IPC system is a core asset of a distributed robotic system, and there are many publications about them. The robotics literature offers some basic re-usable distributed communications components. These include IPC [4], IPT [3], CORBA [24], RT-CORBA [5], and NML (part of RCS) [25]. As with all components, architectural mismatches must be identified and mitigated. Gowdy compared IPT [3], RTC [26], NML [25], NDDS(now RTI-Data Distribution Service [27]), MPI [28], and CORBA [24] for their suitability to robotics applications. He evaluated them in terms of portability, popularity, reconfigurability, ability to support different types of data flow models (e.g. query/response vs. broadcast), and ease of use. He found that IPT and RTC were most suitable for short-term projects with clearly defined goals, and that CORBA might be suitable for long-term projects expected to evolve significantly in purpose. Further afield are software architectures for other types of distributed systems, such as Fielding's dissertation [29] on architectures for network software and the Representational State Transfer messaging paradigm.

Areas of robotics besides mobile robotics have distinct efforts to reuse software. Chetan Kapoor's dissertation [13] develops OSCAR, an architecture and component library for real-time control of advanced industrial robots. Orocos [15] is a research platform for component-based approaches to real-time robot control software. The related project Orca [30] is a component-based approach to re-usable robot software.

We focus in this paper on the software infrastructure rather than the artificial intelligence (AI) software, but we should be aware of reuse efforts in this aspect. An example of these concerns is the way planning and perception functions should be divided into components and how communications between them should be structured. This perspective is important to the software infrastructure as it drives the facilities the software infrastructure should provide. A major architectural decision from the AI perspective is whether to synthesize sensations into a coherent world model as CODGER did [31], or to hard-wire sensor readings directly into behavioral reactions as Brooks' subsumption architecture [32]. Another thrust of research effort is towards the ability to easily coordinate large-grained robot behaviors. Examples in the literature are the Task Description Language [33] and Colbert [34], used for

specifying and ordering execution of robot behaviors. In terms of design lessons, Dvorak [35] claims that conventional notions of modularization in robotic systems creates difficulties in reasoning about resource usage.

We draw ideas from a rapidly maturing discipline for developing and analyzing software architectures in the mainstream software engineering research community [8,36]. Software architecture rests on a discipline of articulating requirements, documenting the architecture, and consistently connecting the architecture to the requirements. A software architecture promotes or inhibits *quality attributes* such as "modifiability", "reusability", and "availability", that describe properties of the system besides the actual functional results it computes. They are often not readily apparent from the structure of the code. *Quality attribute scenarios* [8] are a documentation format used to make these quality attribute requirements specific and testable. There exist typical approaches to achieving quality attributes. Bass et al. [8] term sets of such approaches *tactics*. They provide a checklist for the architect to ensure that all major approaches to a design problem have been considered and that their side effects have been anticipated and mitigated. In our research efforts we are moving toward greater uptake of these design ideas and disciplines to increase the quality of our software.

In the remainder of this paper we describe seven key pieces of the software infrastructure for the Tartan Racing Urban Challenge System software infrastructure.

III.   System Quality Attributes

In this section we describe some of the major quality attributes that motivated the design of the TRUCS software infrastructure. A quality attribute is an adjective such as "modifiability", "reusability", and "availability", that describes properties of a software system that are not directly related to the functional results it computes [8]. Articulation and elaboration of the quality attributes is an important part of the design process. Quality attributes must be explicitly "designed-in". They can require collaboration throughout the system and attempts to hack them into the implementation after the architecture is fixed are usually expensive and unsatisfactory. We look at how the TRUCS infrastructure achieved, and in some cases did not achieve, some of these quality attributes.

A. Flexibility

The software infrastructure is designed to support the AI functionality such as perception and planning. A major requirement of the AI on the infrastructure is that it should be flexible in its ability to direct information flow. Capabilities desired of the robot can require that information be available for decision making at unexpected places in the software structure. This can require radical redesigns of information flow and representation. ModUtils [18] is an architectural framework for infrastructure that is aimed specifically at providing this quality attribute, by allowing components to be configured at run-time to connect to their inputs and outputs using shared memory, a TCP/IP socket, a file, or some other means.

TRUCS provides flexibility of module interconnections using a method similar to ModUtils. There is a code-level interface that the process uses to poll for new input from a stream of formatted messages. The messages retrieved through the interface could be coming from a file, the TRUCS IPC library, or any other method devised. Different message sources are implemented in individual shared object libraries which are dynamically loaded at run-time according to settings in the task configuration file.

B. Independence

The functionality of TRUCS was implemented by dozens of distinct tasks. The system should be robust to crashes in individual tasks. If a task crashes, the rest must continue to function. When the task is restarted, it must be able smoothly to resume its role in the system. This means that tasks must maintain an *independence* of each other.

To achieve task independence, the architecture must eschew synchronous communication between tasks. A task cannot make a request to another task and wait for it to respond. If the recipient task of a request were unavailable, the requesting task would freeze waiting for a response. All tasks in the system could potentially freeze as a result. This indicated our choice of a publish/subscribe style to the interprocess communications system rather than a call-return style.

C. Statelessness

A task is *stateless* if its output depends only on recently received inputs. Statelessness helps to ensure that the function of a task in the system is not negatively affected if it is restarted.

If a task evolves an internal state over time that affects how it interacts with other tasks, a crash in that task could damage the integrity of communications between it and other tasks even after it restarts.

TRUCS had two tasks that were inherently stateful: a mission planner, which kept track of the current goal, and a road blockage detector, which tracked modifications to an earlier road map owing to road blocks. Their state was synchronized to disk to be recovered in case of a crash. Several other tasks accumulated state on a short-term basis, which was not retained in case of a crash. For example, when the robot arrives at an intersection, it must take its turn with other traffic. No effort was made to retain the relevant state variables in case of process crash. The restarted process assumed that the robot was the last of the waiting vehicles to arrive at the intersection.

D. Visibility

It is desirable for messages sent between tasks to be *visible*, that is, for it to be possible to understand the information being transmitted in a single message taken out of context. Visibility of communications allows the operator interface (discussed in Sec. IV) to be able to snoop on communications between tasks. If the tasks communicate in terse terms that depend an on existing shared state between them, their messages may be impossible to interpret. Thus visibility and statelessness are related. Roy Fielding [29] identifies scalability, simplicity, modifiability, visibility, portability, and reliability as key attributes of a network communications protocol.

Avoiding synchronous communications and incorporating statelessness mitigates the risk of partial failure, one of the four major architectural risks of distributed computing as opposed to local computing [22]. TRUCS avoided as much as possible having tasks maintain long-term state, and avoided patterns of communication that required tasks to query each other for that state. If necessary, state information was broadcast at regular intervals, whether requested or not. This was inefficient in terms of network bandwidth used when the information was not needed, but had the advantage that the communication was visible. Regularly transmitting all shared state, rather than only incremental changes, ensured that new observers could join at any point in time. A promising direction for future designs of the messaging paradigm should consider a message as an incremental transfer of state between tasks and provide additionally for explicit representations of the whole state.

E. Repeatability

Repeatability means that running the same program on the same input gives the same result. For typical single-threaded programs, repeatability is a necessary part of functional correctness. For a real-time distributed system such as TRUCS, repeatability is not necessary for correctness. Variations in IPC message arrival times or thread scheduling, owing to nondeterminism in the underlying hardware or operating system, can cause the system to behave differently, though still correctly. A simpler form of repeatability would allow the developer to debug an individual task by re-running it on logged data, presenting logged messages to a task in the same order in which they were received. The TRUCS infrastructure did not provide this capability, but we have identified it as a worthwhile addition to future systems. Without the ability to rerun tasks on logged input data exactly as it ran originally, developers must log excess diagnostic data manually to trace the execution of misbehaving processes.

IV. Operator Interface

The operator interface used by TRUCS was called the Tartan Racing Operator Control System (TROCS). It was the face of the system, used for defining, selecting, and launching missions, viewing logged data, modifying the world during a simulation and viewing the vehicle state while it was running. TROCS includes a 3-D graphical view superimposed with geometric and textual diagnostic notations, and tabs containing text fields and 2-D graphics displaying diagnostic information gleaned from throughout the system.

The need for developer diagnostics drives the design of the operator interface. A TRUCS developer might need to view any of dozens of pieces of data. These could be sensor data at any level of processing such as raw laser hit locations or identified moving obstacles; AI quantities such as predicted future positions of moving obstacles, or of hypothesized obstacles behind occlusions; or robot motion plans under consideration. The size and flux of this list argues for a framework that makes it easy for the TROCS developer to acquire data from the rest of the system and

to create new views of data. It also suggests a need for the TROCS user to be able easily to select just a few views to display at once.

The following sections delve deeper into the general requirements on the operator interface and the implementation of TROCS.

A. Viewing

TROCS displayed data in three ways. The first was a simple 2-D textual display of elementary quantities published by system tasks. Second was scrolling 2-D graphs of time-varying numerical quantities such as the velocity controller's setpoint and response. These types of display would typically show data that did not need to be monitored in real-time, or that did not have a good spatial representation. Third was a real-time 3-D display of the world and the robot's position in it. Data were represented in this display with colored lines, points, and other geometric or textual symbols.

The TROCS display (Fig. 2) consists of a menu, a toolbar with buttons for loading, starting, and stopping mission scenarios, a 3-D view panel showing the annotated world and vehicle, and a tabbed set of panes containing mostly textual diagnostic information. Some space is reserved for vital messages from the process launching system (described in Sec. VI). Users of TROCS interacted mainly with the 3-D view of the vehicle and road network. Developers created lightweight 3-D "drawables" to display data whenever possible. A drawable produces a set of points, lines, and other graphical elements rendered with OpenGL, representing a specific piece of data. Drawables in TROCS were such things as boxes representing the predicted future positions of a moving obstacle, red squares representing the locations of lethal non-moving obstacles, rotating diamonds representing the lane that has precedence at an intersection, or multiple curves extending forward from the robot representing candidate paths to follow over the next 1 s. There were 23 different drawable classes providing 173 individually selectable data items for display. Little code was required to create a new drawable. During testing, the vehicle can switch between multiple modes of operation, for example from driving on a known road map to driving on an unknown dirt road that must be detected as it is driven on. The set of drawables of interest then also switches abruptly. This motivates the ability to save and switch between multiple configurations of drawables, though this was not implemented in TROCS.

In TROCS, a 3-D view of the world is displayed through a camera. A camera is defined by a coordinate frame, which can be attached to the world or to the robot. The position of the camera can be moved within the coordinate frame by the user. If the camera is attached to the coordinate frame of an object that has a slow position update rate, the display may appear jerky and disorient the user. TROCS used an interpolated object position in these cases.
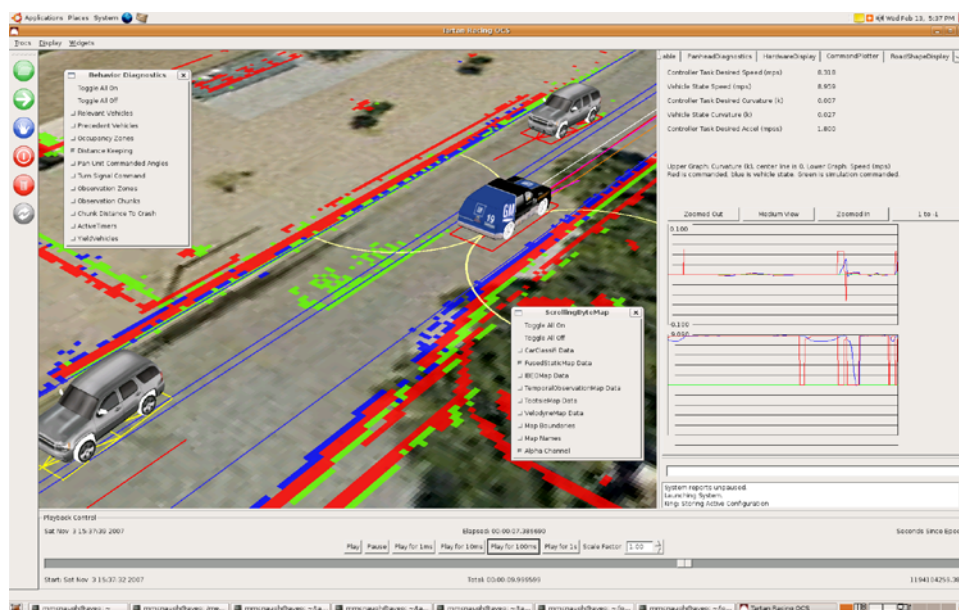

Fig. 2 The TROCS display.

TROCS rendered the majority of 3-D diagnostic information by subscribing to messages produced by tasks and translating them into graphical representations. Another method used was for tasks to publish serialized OpenGL drawing calls directly on a designated channel shared by all tasks. The advantage of this approach was that it was easier for developers to write ad-hoc visualizations while debugging functionality without adding a new message type and drawable that would eventually be obsolete. We identified several caveats against using this method beyond the early task debugging stage of development. Tasks using the direct drawing method could disrupt TROCS in mysterious ways as any OpenGL command could be sent. It could be difficult for operators to distinguish between graphical elements produced by different tasks, as sources were not individually selectable as built-in TROCS drawables were. Extensive display code embedded in tasks could cause performance problems, slowing their operation to unacceptable levels. This also violated our guideline that robot control tasks should be unconcerned with display issues. Finally, diagnostic information represented directly as drawing commands could not be played back from logs with new and better renderings later.

B. Manipulation

The user should be able to interact with the 3-D display by translating and rotating the view. Additional tools may be necessary to interact with the 3-D display to get information about the world. For example, TROCS provided tools to measure the distances between locations in the world, and to display the lowest cost to traverse road elements.

In TROCS, user interaction in 3-D space was provided by cameras. Interactions relevant to the road model were confined to the Road Model Camera, a world-fixed camera. Interactions relevant to the simulation, such as placing obstacles, were confined to the Simulation Camera, a robot-fixed camera. The Road Model Camera offered several interactions, which had to be disambiguated using combinations of the Shift and Control keys and left- and right-clicks. Tools were always active when using a particular camera, so clicking to move the camera could accidentally invoke a tool, which could have undesired side effects on the world. The idea of factoring out a controller (tool) from a view (camera) is not new, but our experience confirms that future operator interfaces should incorporate this principle.

C. Movies

The operator interface is a natural place to make movies. Developers may need to discuss the system's behavior in a certain situation. Researchers need to promote and disseminate their work by giving academic lectures, sending progress reports to sponsors, and soliciting new grants. Videos support all of these efforts. The requirement to create movies of the operator interface has architectural consequences that should be taken seriously.

Third-party video screen-capture applications are available. However, they may be unreliable and unavailable or difficult to get working for less common platforms. They are often built on the assumption that the window to be captured has large areas of uniform color and changes infrequently. As a result, they may show unacceptable performance on large screen areas with detailed dynamic graphics. Playing back data at high rates through the operator interface to record a smooth-playing movie introduces timing problems that push the operator interface and the frame capture software to be integrated.

A movie has a characteristic frame rate and frame size. The frame rate is the number of frames per unit time displayed by the movie when played back at its intended speed. The replay rate is the number of real seconds represented by each second of the movie. The user of TROCS could control the replay rate of the data playback, and the frame rate of the recorded movie. However, movie recording happened in real time and could not be modulated to compensate for hiccups in the data playback.

Slow disk accesses during playback caused TROCS to sometimes be choppy during replay. To remedy this in a future operator interface, four requirements follow. First, that the operator interface can directly control what time sequence of data is replayed. Second, that it knows what time is represented by the played-back data. Third, that it can pause data playback until it is ready for more data. Fourth, that it knows when it has received all the data representing a particular range of time. This implies that it has to know which channels are actually playing back and that all the data has arrived from those channels. The testing and simulation facilities, not described in this paper, could benefit from similar enhancements. However, there is an architectural mismatch between these requirements and the anonymous publish/subscribe communications style used in the TRUCS IPC system (discussed in Sec. V). Future work would have to consider these issues carefully.

TROCS used COTS (common off the shelf) video codec libraries to encode frames into MPEG movies. There are architectural risks when integrating COTS software into a system. We chose to link the libraries directly into TROCS. Instabilities in the libraries occasionally disrupted movie-making. Future designs should firewall the video encoding into a separate process.

D. Data Acquisition

TROCS did not have special status with the IPC system. It obtained data by the same mechanism as all other tasks. Drawables in TROCS subscribed to channels published by other parts of the system for the data they drew. As TROCS functioned as one task within the framework of the IPC system, messages could only be consumed by one drawable. Most messages were only of interest to one drawable, which displayed every aspect of information it contained. For multiple drawables to subscribe to a channel, messages would have to be read by one drawable and communicated to other drawables in an ad-hoc fashion, for example through a global variable. However, developers expected that multiple drawables could subscribe independently to the same messages, just as tasks could, so this was a source of bugs during development of some drawables. We note that this design broke the conceptual integrity of the publish/subscribe system. A future operator interface should have an internal anonymous publish/subscribe system to mirror the design of the IPC system.

V.   Inter-Process Communications

Robotic software systems are frequently distributed systems. That is, the software runs as a set of separate processes, possibly on separate computers. Performance is a key motivation. Many mobile robot solutions require too much processing power or memory to run on a single CPU core, or even a single computer. Distributed systems can be more robust to program errors, and support parallel development efforts by multiple developers.

Many IPC systems have been devised for robotics, as discussed in Sec. II. They are usually created for a specific project and its needs, and do not find widespread use. The design of the IPC system is often shaped by requirements directed at other parts of the overall system. As important system requirements vary between robotics projects, there is reason to expect that IPC systems would not be re-used, even if they have well-factored implementations that are highly reusable at the level of static code modules. Robotics IPC packages often use the anonymous publish/subscribe communication style [36] (also called implicit invocation or event bus) between processes rather than a call-return style, or remote procedure call. This communications style supports flexibility as tasks need not know explicitly about each other, so they can be added and removed from the system relatively easily. The Tartan Racing team developed its own IPC package, named SimpleComms, that was based on the anonymous publish/subscribe communications style.

The implicit invocation style in an interprocess communication context can be analyzed in terms of several design decisions. How those decisions are made depends upon the requirements of the overall system. In Section 7.3 of their book, Shaw and Garlan [36] discuss design decisions that must be made in an implicit invocation architecture. Table 1 summarizes the major design choices made for SimpleComms.

Figure 3 shows a sketch of the communication paths between processes in the TRUCS IPC system. Each machine has a SimpleComms server called `scs` that handles all communications between machines. Tasks participate in the IPC system by connecting to `scs` through a Unix domain socket. The `scs` processes find each other by broadcasting on the local network. The method of using a single `scs` process on each computer responsible for all network transmissions ensures that a message originating at one computer is transmitted to each other computer at most once, minimizing network bandwidth consumption. This enables scalability in the IPC system, so that it can operate unchanged as the number of processes and computers in the system increases. TRUCS transferred approximately 20 MB/s of data over the network. The volume would have been too much for a central hub that received and copied messages to all subscribers. The `scs` tasks on each computer copies messages to the local subscriber tasks.

Data are marshalled into message structures explicitly, using the `Boost::serialization` package. An encoding should be quick to encode and decode and be concise. The `Boost::serialization` default binary encoding has these properties; we use it to encode messages for transport between computers. A desirable quality attribute for a network protocol is *visibility*, which means it easy to sniff the communications stream and display the data being sent over the network. For TRUCS it was convenient enough to display message contents in the operator interface.

Table 1 IPC design decisions

| Design axis	Design choices
When communications channels may be created.	SimpleComms supported creation of channels at any time, but the task configuration system required tasks to declare their channels at start-up.
How participants are identified.	SimpleComms required tasks to identify themselves with a unique string formed from host name and task name. This prevented the developer from accidentally starting two instances of the TRUCS processes on the same event bus.
How participants discover the event bus.	Tasks connected to a local SimpleComms server through a named unix pipe. SimpleComms servers on different machines discovered each other by broadcast on the local subnet.
How data is marshalled from the process into messages, and unmarshalled when messages arrive.	SimpleComms used Boost serialization [37]. Another popular choice is IDL [24].
How message queue overflows are dealt with.	Tasks using SimpleComms had a fixed length queue. New messages that would overflow the queue were discarded.
How messages are sent to multiple recipients.	SimpleComms sent one copy of each message over the network to the SimpleComms server on each machine with a message subscriber. Other choices are to use broadcast or multicast, but this constrains the protocol that can be used and the reliability of message delivery.
Whether messages are delivered reliably.	SimpleComms used TCP to deliver all messages reliably. Another choice is to use UDP to send messages that may be dropped without causing errors.
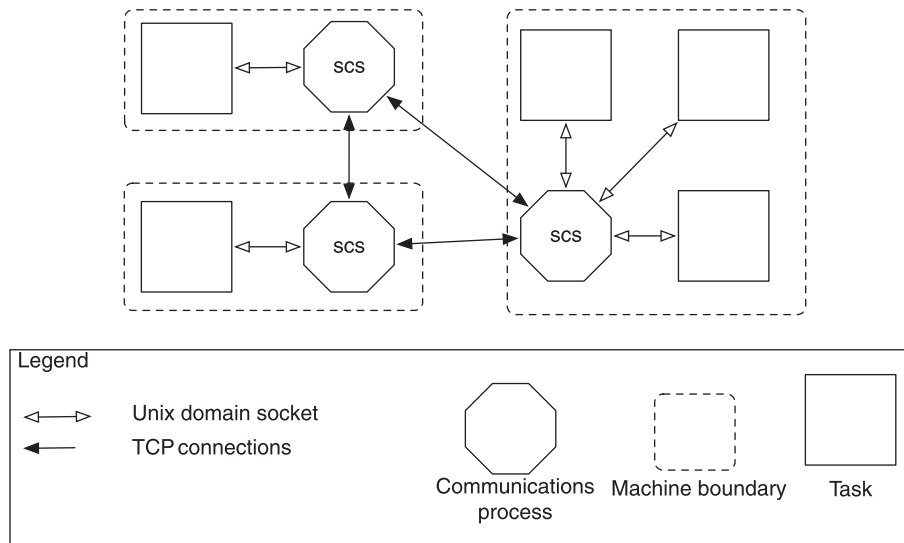
Fig. 3 The process view of the TRUCS IPC system.

The IPC system assumes that messages are guaranteed delivery and hides any packet loss in the TCP layer. When transport links are lossy, it could be beneficial to use a transport protocol such as UDP that does not insist on delivering all messages, if the message semantics for the channel are such that only the latest message is useful.

An important part of an IPC system is how the IPC libraries and its control flow interfaces with the task code. We discuss this aspect of the IPC system further in Sec. IX.

VI.   Process Launching System

TRUCS had dozens of distinct tasks running on 10 computers. In such a large system it is necessary to automate the procedure of launching all of these tasks with the same configuration. In TRUCS, one process per machine, known as the `vassal`, is responsible for launching robot tasks for a specific mission. The `vassal` is started when the developer begins to use the robot and continues to run until he is finished. There is one `king` process that is responsible for mediating between the user at the operator interface and the vassals. It maintains a representation of the user's desired system run state, monitors the health of tasks, and distributes task configuration information to the vassals. It generates a name used to identify the run. This name is used to specify where logs generated during the mission should be stored. Data generated by the process launching system when no mission is running are not logged.

The process launching system must have access to the configuration files to determine a list of available missions, and learn which tasks are to be started. The `king` reads the configuration files, sends a list of available configurations back to the operator interface, and accepts the name of a mission selected by the user. The `king` extracts the configuration files needed for the given mission and sends them to the `vassal` tasks using `rsync` [38]. The `vassal` tasks run the mission tasks designated to run on their computers, and kill them when signalled by the operator. The `vassal` tasks are responsible for logging data related to program crashes, such as core files.

Figure 4 shows the states of the process launching system, maintained by the `king`. The Stop state has two substates, where there is either a configuration loaded or not. The transition from the Startup state to the Stop/No Config Loaded state is automatic. Other transitions are driven by the user from the operator interface.

A reliable system must have a process health monitoring system capability to restore normal operation when a bug in a process causes it to quit unexpectedly or become unresponsive to input. If the process has died, it may be relaunched according to some policy. If the process is in an infinite loop, it should be killed before being restarted. Processes may have different time allowances before being assumed to be stuck in an infinite loop. TRUCS processes reported on a `ProcessHealth` message channel which the `king` listened to and summarized on a `HealthDisplay` channel, which was rendered by TROCS. A process which was not heard from for a set period of time was presumed to be stuck in an infinite loop. The `king` would send a command to the vassals to kill and restart these processes.

VII.   Logging System

Data generated by the system during a mission must be recorded for later review and analysis. It is a special challenge to handle the volume of data generated by large systems with many sensors and tasks. TRUCS had 84 publish/subscribe channels, of which 67 were logged at an aggregate rate of approximately 1 GB/min. A major impediment to logging large volumes of data is the rate at which hard drives can write the data. Space in Boss' equipment rack is at a premium, and hard drives in the hot moving vehicle wear quickly. Boss logs to two hard drives, each attached to a different computer designated for data logging. Once a test run is over, the data must
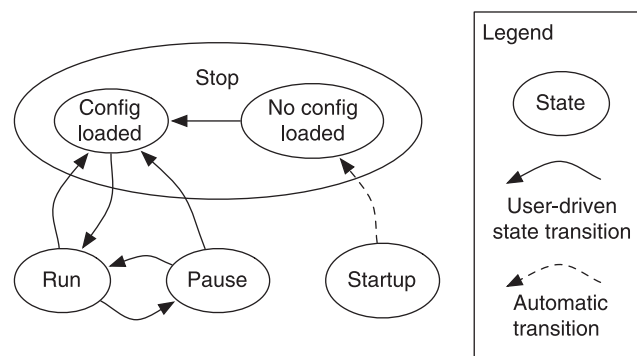

Fig. 4 State chart of the process launching system.

be copied from the robot to long term storage servers, and from there to developer laptops for them to run their analyses.

Given the limited amount of storage space available on a laptop and the limited speed at which the hard drive can read data, it is necessary to excerpt only the data necessary for a diagnostic task before it is copied from long term storage to the developer's laptop. For most diagnostics tasks, only a subset of logged data channels are relevant. For example, if an oncoming car is not properly perceived, messages generated by planning tasks are not needed to diagnose the reason. Following the discipline of using stateless communications between tasks is an advantage here, as data generated long before the incident are not needed to intepret later messages.

TRUCS logging tasks were each responsible for logging one channel. Messages on a channel were logged to a single Berkeley DB [39] file, keyed on the time at which the logging task removed the message from its incoming SimpleComms messages queue. Note that these times were not exactly the same as the times at which other tasks processed the same messages, which introduced some uncertainty during debugging.

Logs are replayed by a dedicated task that reads messages from a given set of Berkeley DB log files. It publishes the messages back on the SimpleComms event bus at a controlled rate. As publishers on the bus are anonymous, tasks that use the messages are unaffected by the fact that a different source is producing them. TROCS can be used to step forward through time and display the messages using the 2-D and 3-D drawables. This makes it easy to correlate events on different channels. New drawables can even be written to display new views on the same data, which could not be done if data were logged in the form of `printf` statements. Tasks besides TROCS can also use the data being replayed over the event bus. For example, logged sensor data can be used to test new perception algorithms.

VIII.   Configuration System

TRUCS used a configuration system based on the Ruby [40] scripting language to manage the variety of task configurations that could run on Boss. Considerations in the design of this system are that there are many tasks, each with many tuning parameters. The tuning parameters may be correlated between tasks. They may also vary by environment or the mission undertaken in the environment. Selections of tasks to be run may differ across environments or missions, and many missions may be mostly the same but for a few small changes.

The organizing concept for the TRUCS configuration system is *inheritance*. Inheritance is used to generate variations on the configuration in two ways. First, "derived" configuration files can include "base" configuration files and then modify selected values. Second, files in a subdirectory corresponding to a mission replace files with the same name in parent directories. This allows the developer to generate a new mission that varies slightly from an existing mission by creating a subdirectory and filling it with configuration files that would replace the existing ones that sit at a higher level. Table 2 describes the configuration information that defines a mission.

A mission was represented as a leaf in a directory hierarchy containing Ruby script files. Figure 5 shows the typical levels at which configuration resources were kept. Each task read a Ruby configuration file named after the task. The Ruby file defined a tree of key-value pairs in the form of a nested hash table, which could then be read by the C++-based main task code.

Table 2 Information required to define a mission

| Resource name	Content	Purpose
Route Network Definition Format (RNDF)	A road map containing roads, their coordinates, number and extent of lanes, location of intersections, parking lots, stop lines, and checkpoints available for use in missions.	To define where the robot is permitted to go and how it is to behave.
Mission Definition Format	A sequence of checkpoints in an RNDF.	To define the sequence of locations that the robot must visit to complete its mission.
Robot configuration information	The tasks to be run on the robot, such as the particular planners and sensor map fusers to use; which message channels to log; tuning parameters for individual tasks	To customize the behavior of the robot for the mission.
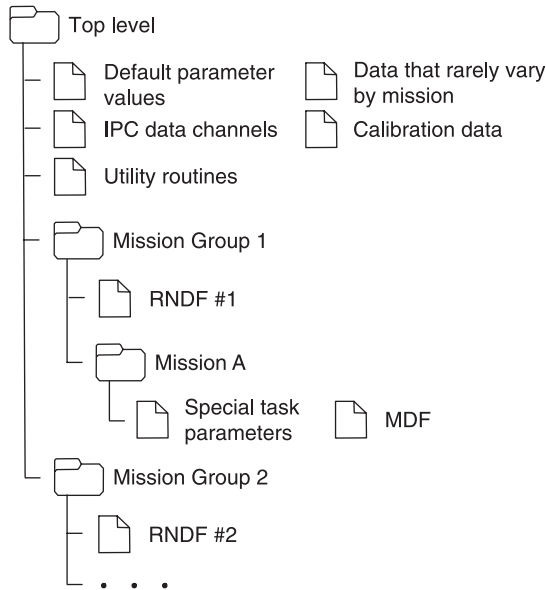
Fig. 5 A sample tree of configuration data.

IX.   Task Framework and Interface Libraries

Every task needs to read configuration data, manage initialization and shutdown, and set up IPC channels, among other things. These functions are done in the same way by all tasks, so a common library serves all tasks. This library provides a task framework, that is, a main program skeleton with specific places for a developer to write in the code unique to that task.

The TRUCS task framework runs the main task loop at a fixed rate set in the task configuration file. The task developer manually inserts a check in the task loop for a new message on each message interface the task subscribes to. The advantage of using a synchronous task loop is that it is simpler to code than an event-driven model where task code is run in response to message arrival. A significant disadvantage is that it can increase overall system latency, for example, between sensing an obstacle and reacting to it.
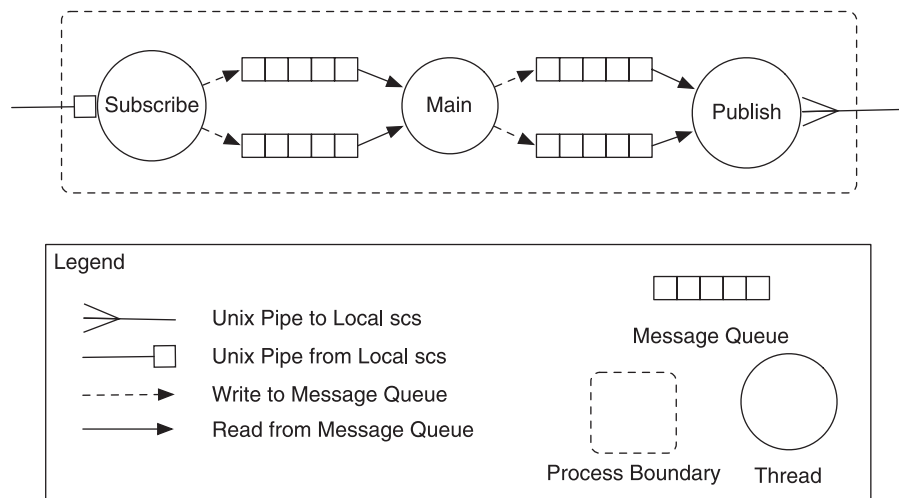

Fig. 6 Thread view of the IPC message library in a TRUCS task.

The message interface can read messages from the IPC system, directly from sensor hardware, from a file, or any other conceivable source. Each source is implemented in a shared object (`.so`) file which is loaded by the task library. An available source is selected in the task configuration file. Message sources can also be changed out at run-time.

IPC system message sources create a set of threads to communicate with the SimpleComms server process on the local machine. Figure 6 shows the threads in the task library and how they interact with the input and output channels. The main thread reads from incoming message queues and writes result messages to the outgoing queue. Read/write threads push messages on and off the read and write queues through the local Unix pipe.

X.   Conclusion

In this paper we looked at seven key pieces of the software infrastructure deployed on Tartan Racing's Boss vehicle for the DARPA Urban Challenge. These were the operator interface, the interprocess communications system, the process launching and management system, the data logging system, the configuration system, and the task framework libraries, as well as the organizing concepts and defining quality attributes of the system.

Fred Brooks contended [41] that "conceptual integrity is *the* most important consideration in system design" (italics original). Discovering the right organizing concepts for the system and their best realization is an ongoing effort. A clear conceptual structure staves off architectural decay, and helps to educate new developers more quickly in how to use the system. For example, in TRUCS we identified inheritance as an appropriate organizing concept for the configuration system, implemented using a broad interpretation in terms of programming language constructs. We also affirmed the event bus architecture as a good fit for interprocess communications in mobile robot systems. We found it beneficial to maintain the independence of tasks by enforcing properties of visibility and statelessness in communications.

Use of the event bus architecture can complicate the implementation of features that require components to know the identities of other participants on the bus, or to know when all messages published before a given time have been received. We showed that the need to make movies in TROCS is one example of this architectural mismatch. Monitoring process health is another. The designer should strive to become aware of architectural risks as early as possible in the design phase. Attempts to graft on features that the architecture does not naturally provide for can cause architectural decay unless plans are made early in the design phase.

We found that the design of TROCS was sufficient for its purpose of providing online and offline diagnostics and simulation control. We would make two minor changes: factor out interaction tools from camera views, and create an internal event-bus mechanism to distribute data to drawables.

We have identified a number of interesting directions for our future work. We shall continue to compile design requirements that can be used to evaluate the fitness of a software infrastructure for its purpose. We believe that an important measure of what has been learned from a software project is the list of new requirements that were discovered during its development and that can be used for future systems. One can also compile a base of candidate features to include in the mobile robot software infrastructure, and anticipate their design implications based on previous experiences, as Bass et al. [8] demonstrate with *tactics*. An infrastructure that supports process-level repeatability, as discussed in Sec. III.E, could greatly ease debugging and smooth the development of future systems, as well as potentially decreasing demands for storage space and network bandwidth. We are also interested in developing approaches to managing long term state for complex missions in changing environments while retaining the benefits of stateless communications between tasks, and the independence of tasks discussed in Sec. III.C. Finally, we are interested in strengthening the reusability aspects of the individual system components discussed in this paper.

Acknowledgments

References

[1] Urmson, C., Anhalt, J., Clark, M., Galatali, T., Gonzalez, J. P., Gowdy, J., Gutierrez, A., Harbaugh, S., Johnson-Roberson, M., Kato, H., Koon, P. L., Peterson, K., Smith, B. K., Spiker, S., Tryzelaar, E., and Whittaker, W. R. L., "High Speed Navigation

of Unrehearsed Terrain: Red Team Technology for Grand Challenge 2004," Technical Report CMU-RI-TR-04-37, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, June 2004.

[2] Urmson, C., Anhalt, J., Bartz, D., Clark, M., Galatali, T., Gutierrez, A., Harbaugh, S., Johnston, J., Kato, H., Koon, P. L., Messner, W., Miller, N., Mosher, A., Peterson, K., Ragusa, C., Ray, D., Smith, B. K., Snider, J. M., Spiker, S., Struble, J. C., Ziglar, J., and Whittaker, W. R. L., "A Robust Approach to High-Speed Navigation for Unrehearsed Desert Terrain," *Journal of Field Robotics*, Vol. 23, No. 8, August 2006, pp. 467–508.
doi: 10.1002/rob.20126

[3] Gowdy, J., "IPT: An Object Oriented Toolkit for Interprocess Communication," Technical Report CMU-RI-TR-96-07, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, March 1996.

[4] Simmons, R., and James, D., "Inter Process Communication," [online] www.cs.cmu.edu/~IPC [retrieved 10 March 2008].

[5] Schmidt, D. C., and Kuhns, F., "An Overview of the Real-Time CORBA Specification," *Computer*, Vol. 33, No. 6, June 2000, pp. 56–63.
doi: 10.1109/2.846319

[6] Gowdy, J., "A Qualitative Comparison of Interprocess Communications Toolkits for Robotics," Technical Report CMU-RI-TR-00-16, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, June 2000.

[7] Gerkey, B., Vaughan, R. T., and Howard, A., "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," *Proceedings of the 11th International Conference on Advanced Robotics (ICAR 2003)*, June 2003, pp. 317–323.

[8] Bass, L., Clements, P., and Kazman, R., *Software Architecture in Practice*, 2nd ed., Addison-Wesley, Longman, Reading, MA, 2007.

[9] Utz, H., Sablatnög, S., Enderle, S., and Kraetzschmar, G., "Miro-Middleware for Mobile Robot Applications," *IEEE Transactions on Robotics and Automation*, Vol. 18, No. 4, August 2002, pp. 493–497.
doi: 10.1109/TRA.2002.802930

[10] Nesnas, I. A., Simmons, R., Gaines, D., Kunz, C., Diaz-Calderon, A., Estlin, T., Madison, R., Guineau, J., Shu, M. M. I.-H., and Apfelbaum, D., "CLARAty: Challenges and Steps Towards Reusable Robotic Software," *International Journal of Advanced Robotic Systems*, Vol. 3, No. 1, March 2006, pp. 23–30.

[11] Konolige, K., and Myers, K., "The Saphira Architecture for Autonomous Mobile Robots," *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, MIT Press, Cambridge, MA, 1998, pp. 211–242.

[12] Microsoft, "Microsoft Robotics Studio," [online], msdn.microsoft.com/robotics [retrieved 25 February 2008].

[13] Kapoor, C., *A Reusable Operational Software Architecture for Advanced Robotics*, Ph.D. Dissertation, University of Texas at Austin, 1996.

[14] Balch, T., "TeamBots," [online], www.teambots.org [retrieved 15 April 2008].

[15] "The Orocos Project," [online], www.orocos.org [retrieved 25 February 2008].

[16] Montemerlo, M., Roy, N., and Thrun, S., "Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, October 2003, pp. 2436–2441.

[17] Broten, G., and Mackay, D., "Barriers to Adopting Robotics Solutions," *Second International Workshop on Software Development and Integration in Robotics*, 2007 [available online].

[18] Gowdy, J., "ModUtils," [online], modutils.sourceforge.net [retrieved 10 March 2008].

[19] Gowdy, J., *Emergent Architectures: A Case Study for Outdoor Mobile Robots*, Ph.D. Dissertation, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, November 2000.

[20] Orebäck, A., and Christensen, H. I., "Evaluation of Architectures for Mobile Robotics," *Autonomous Robots*, Vol. 14, No. 1, January 2003, pp. 33–49.
doi: 10.1023/A:1020975419546

[21] Lindström, M., Orebäck, A., and Christensen, H., "Berra: A Research Architecture for Service Robots," *International Conference on Robotics and Automation*, IEEE, 2000.

[22] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. C., "A Note on Distributed Computing," *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems: Towards the Programmable Internet*, Springer-Verlag, Berlin/New York/Heidelberg, 1997, pp. 49–64.

[23] Tsai, J. J. P., Bi, Y., Yang, S. J., and Smith, R. A. W., *Distributed Real-Time Systems: Monitoring, Visualization, Debugging and Analysis*, Wiley-Interscience, New York, NY, 1996.

[24] "CORBA Middleware," [online], www.omg.org [retrieved 25 February 2008].

[25] Gazi, V., Moore, M. L., Passino, K. M., Shackleford, W. P., Proctor, F. M., and Albus, J. S., *The RCS Handbook: Tools for Real Time Control Systems Software Development*, Wiley, New York, NY, 2001.

[26] Pedersen, J., *Robust Communications for High Bandwidth Real-Time Systems*, Master's Dissertation, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 1998.

[27] "RTI Data Distribution System," [online], www.rti.com/products/data_distribution/RTIDDS.html [retrieved 15 April 2008].

[28] "The Message Passing Interface (MPI) Standard," [online], www-unix.mcs.anl.gov/mpi [retrieved 15 April 2008].

[29] Fielding, R. T., *Architectural Styles and the Design of Network-based Software Architectures*, Ph.D. Dissertation, University of California at Irvine, 2000.

[30] Brooks, A., Kaupp, T., Makarenko, A., Williams, S., and Orebäck, A., *Software Engineering for Experimental Robotics*, chap. Orca: a Component Model and Repository, Springer-Verlag, Berlin/New York/Heidelberg, 2007, pp. 231–251.
doi: 10.1007/978-3-540-68951-5_13

[31] Thorpe, C., Hebert, M., Kanade, T., and Shafer, S., "Vision and Navigation for the Carnegie-Mellon Navlab," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 10, No. 3, May 1988, pp. 362–373, Other versions appeared in *High Precision Navigation* (Springer-Verlag, 1989) and in *Annual Reviews of Computer Science*, Vol. 2, 1987.
doi: 10.1109/34.3900

[32] Brooks, R., "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, Vol. 2, No. 1, March 1986, pp. 14–23.

[33] Simmons, R., and Apfelbaum, D., "A Task Description Language for Robot Control," *Proceedings of Conference on Intelligent Robotics and Systems*, IEEE, October 1998.

[34] Konolige, K., "COLBERT: A Language for Reactive Control in Sapphira," *KI '97: Proceedings of the 21st Annual German Conference on Artificial Intelligence*, Springer-Verlag, London, UK, 1997, pp. 31–52 [online], http://portal.acm.org/citation.cfm?id=731754&dl=GUIDE&coll=GUIDE#.

[35] Dvorak, D., "Challenging Encapsulation in the Design of High-Risk Control Systems," *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM, November 2002.

[36] Shaw, M., and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Upper Saddle River, NJ, 1996.

[37] "boost C++ Libraries," [online], www.boost.org [retrieved 16 April 2008].

[38] "rsync," [online], samba.anu.edu.au/rsync [retrieved 20 April 2008].

[39] "Berkeley DB," [online], www.oracle.com/technology/products/berkeley-db/db [retrieved 21 April 2008].

[40] "The Ruby Language," [online], www.ruby-lang.org [retrieved 11 March 2008].

[41] Brooks, Jr., F. P., *The Mythical Man-Month*, chap. 4, Addison-Wesley, Longman, Reading, MA, Anniversary ed., 1995.

Christopher Rouff
Associate Editor