Writing Code in the Field: Implications for Robot Software Development

William D. Smart

Department of Computer Science and Engineering
Washington University in St. Louis, United States wds@cse.wustl.edu

1 Introduction

Running robots on a deployment in the "real world" is very different from running them in a laboratory setting. Environmental conditions, such as lighting, are often completely beyond your control. There is often a limited amount of time on-site to test and tune the robot system, and deadlines are generally very inflexible. Finally, the public is notoriously unforgiving of failure. In this chapter, we identify some key design issues that can affect real world robot deployments. We discuss these issues and propose some solutions, based on our own experiences, designed to ease the pain of sending a robot out into the real world.

1.1 The Deployment Scenario

In this chapter we will consider a particular type of real world deployment, characterized by six main features:

Environment: The deployment takes place in an environment where we do not have complete control over the environmental conditions, such as the lighting level. We also assume that we cannot completely replicate the environmental conditions in the laboratory before arriving at the deployment.
We cannot alter the environment in any substantial way for the deployment. This includes removing furniture and obstacles that the robot has difficulty with, adding brightly-colored navigation aids, and forbidding the public from areas of the environment.

Schedule: The robot is not the most important thing in the environment, and existing schedules cannot be altered to accommodate our deployment. For instance, the opening time of a building cannot be delayed if the robot is not ready on time.

We often have a limited amount of time on-site to make sure that the robot behaves as expected, and to tune it to the specifics of the environment. This limitation generally comes from the policies of the deployment site which may, for example, restrict after-hours access to the venue.

Task: The robot is performing some autonomous task during the deployment, and is not just a static exhibit. It is expected to interact with the environment, and possibly the general public, in an "intelligent" and purposeful manner.

Duty Cycle: The robot must perform its task for a set period of time, often several hours, without intervention. All maintenance and repairs must happen outside of this time period (after the building has closed, for example).

The Public: The public will be observing the robot as it performs its task, and might even be interacting with it directly. We cannot assume that the public knows anything about robots, computer science, algorithms, or the like. Although human "interpreters" can be used to explain the robot's behavior, they should be used sparingly.

Reputation: The public has a low tolerance of failure, and will associate poor performance with our lab, institution, and students. Failures in the laboratory, in front of a knowledgeable audience, can be (somewhat) explained away at a technical level. No such easy escape exists in the real world. If the robot seems broken, or is acting erratically, the public will lose interest and move on. More importantly, they will remember that the robot didn't work as expected, and will tell their friends.

As a concrete example, consider the deployment of a mobile robot in a science museum. The museum has fixed hours, which we have no control over. The robot *must* be ready when the doors open at 9am, and we cannot perform any obvious maintenance until the museum closes at 5pm. The robot is likely to be one of a large number of exhibits. If someone comes by while the robot is not working, they will simply move on to the next exhibit. This means that it is important to keep the system working for the entire duty cycle. This is especially true if it has been advertised, since people might have made a special trip to see the robot in operation.

It is unlikely that we can replicate the conditions of a typical science museum in our lab, although we might be able to approximate some of them. This means that we will need to spend time tuning our system to operate in the museum. The amount of time we can spend doing this will be limited by museum policies. It also must happen at an inconvenient time, in the evenings and very early mornings.

1.2 Typical Deployment Problems

For any non-trivial deployment of the type described above, we will have to adapt our systems on-site. This adaptation will involve the tuning of software parameters to account for environmental conditions, and will usually also include writing new code to deal with unforeseen circumstances. Often, several subsystems must be tuned, which leads to contention a single shared resource: the robot.

It is a fact of life that hardware and software will fail. This is especially true on deployment. Software written on-site, under time pressure, will not be of the same quality as that developed over time in the laboratory. This means that it will fail more often, often catastrophically. Simply shipping the robot to the deployment site can increase the likelihood of hardware problems. Components and connectors can be shaken loose and gears can be knocked out of alignment. Even the most careful pre-deployment inspection cannot be guaranteed to catch all hardware problems.

We must accept that things will fail, and try to manage this failure so as to minimize its effect on the deployment. We must ensure that any failures that do occur result in a graceful performance degradation, rather than stopping the system dead. The robot should continue to operate, perhaps at a reduced capacity, until the end of the duty cycle, when we can perform the needed maintenance.

1.3 Software Architectures for Deployments

We have deployed our mobile robot in a number of real world situations, and observed many instances of the problems outlined above [SDM03, BDG03, BDSG04]. This has led us to modify our software architecture [MS04] and development process to ease the deployment experience. Our modifications have been aimed at two things: reducing software development time on-site and increasing robustness to the inevitable hardware and software failures that occur. In this chapter, we describe some of the problems encountered on deployment, and the modifications that they caused us to make in our control architecture.

2 Reducing Software Development Time On-Site

Performing some amount of software development in the field, while on a deployment, is almost inevitable. Thus, we need to make the on-site development process as efficient as possible. Although we should always strive for efficient software development, there are some additional pressures on a deployment, and often severe penalties for missing deadlines. For example, consider a robot that is part of a larger exhibition in a science museum. The museum has a hard deadline for opening in the morning, and will not postpone it if the robot is not ready. If the public arrives to see the robot surrounded by puzzled-looking students, there is a real price to pay. Our public reputation, and that of our institution, can suffer. Perhaps more importantly, our credibility with the museum will be hurt, which can affect plans for subsequent deployments.

It is vital to have an effective development and testing process in place, well before we arrive at the deployment. Although we cannot eliminate on-site tweaking of code, we can certainly minimize it by planning ahead. Arriving at a deployment without a full, working application is simply asking for trouble. Time in the field should be spent adapting a system, not implementing it. Even if we cannot replicate the conditions of the deployment in the laboratory, we can, at least, come armed with a system that works under *some* conditions.

While it seems obvious to say that good software engineering practices are useful, this is often forgotten in a university or research lab environment. A good deal of the code written in such an environment can be considered to be proof-of-concept code, designed to test a theory and then to be discarded. Code is often modified incrementally, without a formal specification, leading to undocumented behavior. We can often get away with this for internal deployments, since the person who wrote the code is probably there to answer questions. However, this can cause severe problems while on an external deployment.

All code that is to be used on a deployment should be maintained under a source code control system, such as CVS. When a new version of some part is written, unit and integration tests should be carried out before it is checked into the official code base. Documentation and test code should also be checked in at the same time as the main code.

We have found that it is extremely useful to have a student act as the configuration manager for the robot. While any student can check code out of the repository, only the configuration manager (CM) may check in new versions. This allows us to easily make sure that the other students do not "forget" to include documentation and test routines with their new code.

The role of configuration manager is doubly important for a robot, since hardware can also be changed. Just as no-one is allowed to add to the code base without the permission of the CM, no-one can alter the hardware configuration. The removal or addition of hardware can cause many problems, and a huge amount of frustration. We have had a student spend several hours trying to debug the speech synthesis software on our robot, only to find that someone had removed the sound card without telling anyone. While one could argue that the hardware should be the first thing that we check, we tend to make assumptions about the sources of failure. Computer science students tend to assume that the software is at fault.

Adding hardware to a robot can be just as bad. Everything on a robot is powered from a shared set of batteries. If we have too many devices, there may not be enough power to go around, leading to strange hardware behavior. We have observed this problem first-hand (with another team) at a major robot competition. A student plugged in a new camera system just before the start of the run. It drew enough power from the batteries to cause the laser rangefinder to reset, and stop returning values. Since the obstacle-avoidance system relied on the laser, the robot was essentially blind, and drive straight into the first table it encountered. A large group had gathered to watch the robot, making it a high-visibility failure.

Perhaps worse than the failure itself was the time needed to debug the problem. All of the symptoms were of a faulty laser range-finder. However, this device was fine, as all testing in isolation confirmed. It is a testament to the experience of the deployment team that the problem was found at all.

Suggestion 1
Designate someone as the Configuration Manager (CM). No code or hardware is changed on the robot without the explicit approval of the CM. The CM is also responsible for enforcing all coding standards and procedures.

Most of time on-site is spent tuning parameters in the robot software, to adapt it to environmental conditions. We will begin by discussing how this process can be optimized. We will then cover how to minimize the effects of any on-site software development that may be necessary. In all of the following, we assume that we have a single robot (or a small number of robots) and multiple developers, each with a computer. The robot is a shared resource, and one of our primary goals is to reduce the contention for this resource.

2.1 Tuning Parameters

The most obvious way to reduce the time spent tuning parameters is to have fewer parameters to tune. Although it is not always possible to remove parameters from an algorithm, it might be possible to set reasonable default values that are "close enough". These default values might be set according to some context given by the developer. For example, there might be one set of vision algorithm defaults for a bright environment with direct sunlight, and another for a window-less room with artificial lighting. The parameters can be properly tuned if there is sufficient time, but the default values will prevent catastrophic failure if we run out of time. If we do tune the parameters for a specific algorithm, the settings should be saved, to be reused in similar environments in the future.

For some parameters, there may be no good default value. In this case, a procedure for determining a good setting should be included in the documentation. If this procedure is straightforward and algorithmic, it means that the tuning can be done by any member of the deployment team, not just those with an intimate knowledge of the algorithm.

Suggestion 2
Reduce the number of parameters to tune, and provide defaults where possible. Provide guidelines for determining a good setting for parameters without a default.

If it is reasonable to have parameters optimized automatically, then this is almost certainly the best approach. Even if this optimization is not complete, it might provide a better starting point for a human programmer. We make extensive use of this approach with the vision system on our robot. We calibrate the stereo vision system by taking a picture of a known target (a checkerboard pattern), and running code to estimate the camera parameters. The stereo vision calibration is not entirely automatic, and requires some human input. However, this requires no skill on the part of the human, and can be done by any of the deployment team. We can also correct for local lighting conditions by taking a picture of a target with a (known) variety of colors.

Suggestion 3
Write code to automatically tune as many parameters as possible. If human input is required for any of these calibrations, have explicit instructions that do not assume any knowledge of the underlying algorithm.

Parameters settings should not be defined in the code. If they are, then every parameter setting change requires that the code be re-compiled, which will cost valuable time (see below for more details). Each subsystem should save its parameter settings in a configuration file, in some human-editable form, such as XML. If the robot has been successfully deployed in a similar setting before, all of the parameter settings for that deployment should be saved in a single place. This will potentially reduce the amount of time spent tuning parameters in similar settings, by providing a better starting point.

In our architecture, parameters are stored in configuration files, but can also optionally be provided by a configuration server. This service, based on a similar system in CARMEN [TMR06] allows us to change parameter values while the robot is running, using a graphical interface. This lets us see the effects of a parameter setting immediately, without having to stop and start the code. This saves time, and we have found that it is often easier to set some parameters, like how close to come to obstacles, interactively. Additionally, being able to change parameter values while the robot is running allows us to recover from some failures transparently (see section).

Suggestion 4
Save parameter settings in a configuration file, in some human-editable format, such as XML. Provide a mechanism for modifying the parameter values while the robot is running.

It is painful to watch programmers waiting in line to be able to use the robot, while someone tunes *their* set of parameters. If we design our applications as monolithic systems, parameter changes to one sub-system can affect other sub-systems. Even if they do not interact at run-time, if we have a monolithic controller, we cannot run two copies of it at the same time, meaning that only one programmer can be occupied.

If, however, the architecture is modular, and allows sub-systems to be run independently, we can parallelize the process of calibration. The vision programmer can tune the color tracking model, while the sensor programmer calibrates the laser range-finder. Better yet is the ability to run sub-systems on a workstation that is not directly connected to the robot, from the command line, using data logged to files. This allows, for example, the vision programmer to capture a sequence of images, then use these to tune the color tracking parameters on a workstation while the robot is being used for other tasks.

To address this, our software architecture is composed from many small, independent modules. Each of these modules is a single process (in a Linux environment), and is designed to be run as a service (the basic compositional unit of our architecture), a pipe element (in the Unix sense, reading from the standard input, writing to the standard output), and a command-line program (reading from and writing to files), depending on the arguments used. This allows us to use the same code for each mode of operation, ensuring that tuned parameter settings for one mode will be valid in another.

This arrangement allows us to quickly prototype prototype simple sequences of processing steps on the command-line. A saved image can be piped into the color blob finder, with the output being saved to a file. Looking at this file allows us to quickly debug problems with the blob finder. Once it is working as expected, we can pipe the output into the face detector, which again outputs to a file that can be examined, and so on.

Suggestion 5
Provide as much modularization as possible in the system. Allow sub-systems to be run independently of each other, and to deal with input and output from a variety of sources.

2.2 Writing New Code

In addition to reducing the time it takes to tune parameters, we must reduce the time it takes to write new code while in the field. Regardless of how well-prepared we are, it is almost inevitable that we will have to write *some* new code.

Instead of writing completely new code in the field, we should aim to compose existing (compiled) software into new applications. This achieves two important goals: it avoids introducing bugs due to hastily-written code, and it avoids having to recompile code. Software written under (often extreme) time-pressure will almost certainly be less reliable than that written and tested in a more controlled environment. Re-compiling code (often many times, as incremental changes are made and tested) is not only time-consuming, but increases the stress level of the programmers. As time becomes short, staring at a long compile is a harrowing thing.

If we are forced to re-compile code, we want to compile (and link) as little as possible. Although a build management system will help with this, we can reduce the size of the compile even further if our system is composed of small, independent modules.

Our approach is to have a large number of small, independent modules. Each module provides a small number of interfaces, over which information flows. Typically, each interface (such as "distance") can be provided by more than one service (laser range-finder, sonar range-funder, stereo vision, *etc.*). Applications request interfaces from a broker agent, which selects appropriate services to provide the interface (which may request interfaces themselves, and so on). There is considerable flexibility in the selection of actual services, which means we can adapt to different environments quickly, by selecting appropriate ones. For instance, we have several services that supply the "obstacle-avoider" interface, each of which use a different algorithm. Depending on the specifics of the deployment, we can easily select the most appropriate algorithm, without having to recompile any code. We can also be sure that it will work with the rest of the system, because all interactions with it are performed through the generic "obstacle-avoider" interface.

Suggestion 6
Build applications by composing small, independent, previously written (and tested) modules.

3 Guarding Against Hardware and Software Failures

Despite our best efforts, hardware and software will fail in a variety of ways. This is especially true when we are running the robot out of the laboratory. Transporting the robot to the deployment site might have caused some hardware components to shake loose. Software written during the deployment is not going to be as reliable as that written and tested in the laboratory. Although we should always try to avoid hardware and software failures, we must also recognize that we cannot realistically eliminate them all. This means that we must plan for these failures, and try to minimize their impact on the deployment.

The most basic safeguard is to have a complete backup of the *entire* baseline system. If the hard drive in the robot fails, we need to be able to replace it with a new one, and restore the baseline system as quickly as possible. On deployments, we actually have two identical disks, so that we can restore the system by physically installing the backup. Any changes to the baseline system made on-site are under source code control on a remote workstation, and can be applied to the baseline system in the case of failure.

Suggestion 7
Have an up-to-date backup of everything that you can. Be able to restore the software to its baseline state quickly in the case of failure. Commit on-site changes to a source code control system, so that they can be easily applied to the baseline system.

When running robot applications in the laboratory, we want to know about problems with the hardware and software immediately. When they occur, we stop the robot, diagnose and fix the problem, and restart the application. However, when we are on deployment in the real world, our goals are different. In this case, if some subsystem fails we want the robot as a whole to continue operating as well as possible until the end of the current duty cycle. At that time, we can intervene to fix the problem. Ideally, members of the public watching the robot will never even be aware that something failed.

Our robots must show graceful performance degradation in the face of hardware and software failures. This means that we must identify failures, and switch to some backup system quickly enough that the overall behavior of the robot is not noticeably affected. Moreover, this failure tolerance should be built into the software architecture (as much as is possible), so that application developers do not have to consciously think about rare failure conditions while writing their code.

To sequence applications at a higher level, we have a scripting language that describes augmented finite state machines. This lets us quickly assemble high-level applications by composing lower-level behaviors. It removes the need to re-compile these low-level behaviors, and also gives us strong encapsulation, and crash protection, which we discuss in the next section.

A key element of our approach is to provide strong encapsulation for subsystems. The failure of a subsystem should not cause other systems to fail. One way of accomplishing this is to use the process encapsulation mechanisms of the underlying operating system (Linux, in our case), and make each subsystem an independent process. If one process fails catastrophically, the others will be insulated from this failure. This is especially relevant on deployment, where we might be running code that was hastily-written, and not well tested.

Suggestion 8
Provide strong, operating system level encapsulation between subsystems where possible.

Our approach to adding robustness to failures draws heavily on ideas from Recovery Oriented Computing (ROC) [PBB02], and in particular from the idea of micro-reboots [CF01]. If a subsystem fails, restarting it will often cause it to work again, for a while at least. If this reboot is quick, the overall performance of the system will not suffer. We extend this idea by allowing new services (that provide the same interface) to be started in place of the one that failed.

Even code that has been well-tested cannot be guaranteed to be free of bugs. However, if a piece of software fails, it can often be successfully restarted. This is a reasonable strategy for keeping a robot active during a deployment, since software becomes more unreliable the longer it runs (due to slow memory leaks, and the like). We can often make quite strong empirical claims about the short-term behavior of software. For example, we can be confident that a piece of code will run for one hour if we repeatedly run it for one hour without it failing. We cannot do so well if we want to claim that it runs for a whole day. However, if we can seamlessly restart the program every hour (or when it fails), we can approach this stronger claim. To make this possible, individual modules in the software architecture should have as little internal state as possible. If we must have internal state, it should be stored in some way that allows it to persist through a process restart (in shared memory, or stored in a file, for example).

Suggestion 9
Have as little internal state as possible in a process. If the subsystem absolutely requires internal state, store in a persistent way, that can survive a restart.

The broker agent in our architecture monitors every service (process) in the system. Those that fail catastrophically are restarted up to a certain number of times. If a service fails too often, it is removed and replaced by another service that provides the same interfaces. If no such interface is found, then the system enters a failsafe mode, and requests human assistance.

Each process also has a watchdog timer that is monitored by the broker agent. If a process seems to be inactive, it is restarted or replaced. The broker can also optionally observe the output from sensor services, trying to identify failure. By their nature, sensor readings have measurement error in them. This means that a healthy sensor will rarely return exactly the same value every time it is read. If a sensor does return the same value every time, this is an indication that the hardware has failed, and that the service should be shut down.

Suggestion 10

Provide automatic monitoring of processes and (where possible) hardware systems to automatically detect and correct failures.

We can perform this replacement without notifying the application because of the abstractions used in the interfaces. For example, to determine the location of objects in the world, an application might request the "distance" interface, with a preference for data generated by a laser range-finder. The broker agent performs the steps necessary to connect the application to the laser range-finder service, and all is well. If the laser fails, however, another service that supplies the "distance" interface, such as the sonar range-finder, can be substituted in, without the application being aware. The data generated by the sonar range-finder will, of course, be different from that generated by the laser. However, the hope is that it is "good enough" to keep the application from failing catastrophically.

There is a price to be paid for using such abstractions. We can no longer ask for data from a particular sensor, such as the laser range-finder. This also means that we cannot make assumptions about, for instance, the failure modes of the data being supplied. Our applications must be written in more generic terms, requesting the specific characteristics of sensors before they are used, rather than making the assumptions that are common in current robot software. This makes writing software more time-consuming, in our experience. However, we believe that the level of added robustness is well worth it. We can, of course, still directly request laser range-finder data, but by doing this we lose any graceful degradation. If we require the laser to be working, and it breaks, then our application must deal with this failure in some intelligent manner.

Abstractions should not be limited to sensor data. Our architecture using abstractions for common tasks such as path-planning. When requesting the "path-planner" interface, the application can request a particular algorithm, with a preference level ("required" to "don't really care"). If a path-planner repeatedly fails, it can be replaced by another (often using a different algorithm). This allows us to provide robustness against poorly-written code.

Suggestion 11
Provide abstractions for commonly used things like distance measurements and path-planning, and write all applications in terms of these abstractions. This allows backup services to be substituted for primary ones that have failed.

One of our goals on deployments is to hide hardware and software failures from the general public. While the techniques described above have proven to be extremely useful, they are not foolproof. There are still failures that will cripple our system. Some of these, such as the primary computer failing, cannot be covered without implementing multiply-redundant parallel hardware systems. However, for other unforeseen problems we have implemented an option of last resort. If the robot needs an interface that cannot be provided, often due to multiple failures, it launches a graphical interface on a nearby workstation, and asks for human help. A human operator can intervene, extricate the robot from its dilemma, and then give control back to the autonomous system. The operator can also instigate this recovery mode. Although this is technically cheating, and not appropriate in all situations, it has saved us embarrassment in a few situations.

Suggestion 12
Have an emergency recovery mode that allows a human to take direct control of the robot, extricate it from trouble, and then give control back to the autonomous system without the general public being aware of the intervention.

4 Some Concluding Thoughts

Most robot architectures are designed to make robot programming easier [OC03, MAR06, USEK02], to provide a framework for particular application research [TMR06], or to provide support for experiments [UMK04]. Our architecture is the only one we are aware of that is designed with deployments in mind. Although we have a complete architecture, the ideas discussed in this chapter could be applied to (almost) any existing robot architecture to provide robustness, and to make deployments easier.

Our architecture provides a basic level of fault tolerance. However, explicitly building in fault detection and recovery systems into the application will result in a vastly more robust system [MNPW98, WN96]. This is especially true if advanced, model-based reasoning techniques are used [DWH04, VGST04]. However, this required detailed knowledge of the application, and is hard to make generic.

In this chapter, we have discussed some of the observations that we have made during several real-world deployments of a mobile robot system. These observations have caused us to change our perspective on robot software architectures and the middleware provided by them. The underlying ideas of our current architecture are

1. Support for efficient on-site adaptation of applications.
2. Graceful degradation under various failures.

Our observations are directed at robot deployments where the public are present, and there is a cost to stopping the system to fix a problem. They are not meant to cover every possible deployment, although we believe that many of the lessons will also have value in other deployment scenarios.

We are currently working on evaluating and extending our system. It was successfully demonstrated during the Mobile Robot Competition and Exhibition held at AAAI 2004, and successfully recovered from an unscripted sensor failure during one of the runs in the Challenge event. We are currently focusing on more advanced failure recovery and prediction mechanisms.

References

[MAR06] *Mobile and Autonomous Robotics Integration Environment (MARIE)*, http://marie.sourceforge.net/, 2006.

[TMR06] Sebastian Thrun, Michael Montemerlo, and Nicholas Roy, *Carnegie Mellon robot navigation toolkit (CARMEN)*, http://www.cs.cmu.edu/carmen/, 2006.

[UMK04] Hans Utz, Gerd Mayer, and Gerhard K. Kraetzschmar, *Middleware logging facilities for experimentation and evaluation in robotics*, September 2004.

[USEK02] Hans Utz, Stefan Sablatnog, Stefan Enderle, and Gerhard K. Kraetzschmar, *Miro Middleware for mobile robot applications*, IEEE Transactions on Robotics and Automation 18 (2002), no. 4, 493497.

[SDM03] William D. Smart, Michael Dixon, Nik Melchior, Joseph Tucek, and Ashwin Srinivas, *Lewis the graduate student: An entry in the AAAI robot challenge*, AAAI Mobile Robot Competition 2003: Papers from the AAAI Workshop, 2003, Available as AAAI Technical Report WS-03-01, pp. 4651.

[MS04] Nik A. Melchior and William D. Smart, *A framework for robust mobile robot systems*, Proceedings of the SPIE: Mobile Robots XVII (Douglas W. Gage, ed.), vol. 5609, December 2004, pp. 145154.

[CF01] George Candea and Armando Fox, *Recursive restartability: Turning the reboot sledgehammer into a scalpel*, Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), May 2001, pp. 125132.

[PBB02] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Mertzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft, *Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies*, Computer Science Technical Report UCB//CSD-02-1175, Department of Computer Science, University of California at Berkeley, March 2002.

[BDG03] Zachary Byers, Michael Dixon, Kevin Goodier, Cindy M. Grimm, and William D. Smart, *An autonomous robot photographer*, Proceedings of the IEEE/RSJ International Conference on Robots and Systems (IROS 2003), vol. 3, October 2003, pp. 26362641.

[BDSG04] Zachary Byers, Michael Dixon, William D. Smart, and Cindy M. Grimm, *Say cheese!: Experiences with a robot photographer*, AI Magazine 25 (2004), no. 3, 3746,

[OC03] Anders Oreback and Henrik I. Christensen, *Evaluation of architectures for mobile robotics*, Autonomous Robots 14 (2003), 3349.

[DWH04] Richard Dearden, Thomas Willeke, Frank Hutter, Reid Simmons, Vandi Verma, and Sebastian Thrun, *Real-time fault detection and situational awareness for rovers: Report on the mars technology program task*, Proceedings of the IEEE Aerospace Conference, March 2004.

[MNPW98] Nicola Muscettola, Pandu Nayak, Barney Pell, and Brian Williams, *Remote agent: To boldly go where no AI system has gone before*, Artificial Intelligence 103 (1998), no. 12, 547.

[VGST04] Vandi Verma, Geoff Gordon, Reid Simmons, and Sebastian Thrun, *Particle filters for rover fault diagnosis*, IEEE Robotics and Automation Magazine (2004), Special Issue on Human-Cenetered Robotics and Dependability (In Press.

[WN96] Brian C. Williams and P. Pandurang Nayak, *A model-based approach to reactive self-configuring systems*, Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), AAAI Press / The MIT Press, 1996, pp. 971978.