On specifying reactivity in robotics

Geoffrey Biggs and **Bruce MacDonald**
Department of Electrical & Computer Engineering, University of Auckland

Email: g.biggs at ec.auckland.ac.nz, b.macdonald at auckland.ac.nz

Abstract

Robot programming infrastructure needs to improve for robots to achieve their potential. A key aspect of the programming component is handling responses to events, both expected and unexpected. This paper presents the current status and trends in specifying reactivity in robotics. Recent and notable work in event handling is examined. The trends, future directions and demands for reactivity in robotics are discussed. The current status of reactive specification methods is found to be less than satisfactory. Recommendations are given on where future research should focus.

1 Introduction

Robots have become increasingly complex and their controllers increasingly powerful, yet robotic programming tools have not advanced as rapidly [?; ?]. Robots must be programmed both at the development stage, to create the functionality of the robot, and in the field, to customise the robot to applications, environments and tasks. It is important that robots become easier to program so that their potential may be fully realised.

Robot researchers face difficulties developing medium to large software systems for robots that are to assist humans in everyday human environments. Robot systems present special demands and as a result much of the software infrastructure is proprietary, much is necessarily targeted at specific hardware, robot software development kits may be limiting, there is a lack of open standards to promote collaboration, code reuse and integration, and there is a lack of techniques for bringing the human in to the robot's programming infrastructure.

The difficulty is the complex interactions robots have in real environments and the complex sensors and actuators that robots use, including:

- A large number of devices for input, output and storage, which far exceed human programmers' familiar senses and effectors, compared to the few devices in a desktop or server.
- Simultaneous and unrelated activity on many inputs and outputs.
- Real time requirements, as the automation system must operate in the real world.
- Unexpected real world conditions.
- Wide variations in hardware and interfaces, as opposed to the highly commoditized desktop.

Programmers of robot arms and other complex articulated automatic devices must also deal with non–intuitive geometry. Programmers of mobile robots must deal with widely varying and unpredictable conditions as the robot moves through its environment. Standard debugging tools give programmers access only to program data. This makes debugging robot programs difficult because program data is at best an indirect representation of the robot and environment.

We believe robot programming systems must be targeted more closely to robotics, paying attention to the nature of typical robot programs, the typical skills of robot programmers, the interactions between humans and robots, and the programming constructs that prevail in robotic applications.

We view robot programming systems as having three important conceptual components that are of interest to their designers:

- The programming component, including designs for programming language/s, libraries and application programming interfaces (APIs), which enable a programmer to describe desired robot behaviour.
- The underlying infrastructure, including designs for architectures that support and execute robot behaviour descriptions, especially in distributed environments.
- The design of interactive systems that allow the human programmer to interact with the programming component, to create, modify and examine

programs and system resources, both statically and during execution. The human programmer may also interact with the infrastructure component, to examine, monitor and configure resources, and directly with robots as they perform tasks.

There are other components that are not of particular concern to designers of robot programming systems, such as the robots themselves, operating systems, compilers, robot hardware drivers and so on. A few aspects, such as real time operating system performance, will be of concern.

1.1 Event handling

In this paper we address a key aspect of the programming component; how a programmer may specify the robot's behaviour in response to events. Two major issues for robot programmers are (a) specifying how a robot is to respond to events as they occur, such as asynchronous sensor events on touch, sonar, laser, and vision channels, and (b) specifying how a robot is to handle unexpected, exceptional situations, such as unplanned obstacles, faulty sensors and motor events.

Consider for example a simple vacuuming robot. Suppose the robot has map of the room it is to vacuum. The robot controller must plan paths to move around room and cover the space with its vacuuming nozzle, and then follow the paths. Many events will occur as the robot follows the path, including moving obstacles, changes in the room layout that do not match the map, inaccurate movements that cause the robot to run into furniture, and so on. The robot program must be easily designed to cope with all these events and those that cannot be thought of in advance.

Vacuuming behaviour is illustrated in Figure 1. The robot plans a path to cover the room, then begins following that path. It may encounter static or dynamic obstacles. Static obstacles require the map be updated and a new path planned, while dynamic obstacles require the robot to wait for the obstacle to move. The robot may also encounter hardware problems (such as a jammed vacuum cleaner).

Recent and notable work in handling expected and unexpected events is reviewed in section 2. Section 3 discusses the directions that reactivity in robotics might take in the future and what the next step should be. Section 4 presents conclusions.

2 Literature Review

Existing methods for handling events can be conveniently divided into those for handling expected events and those designed for handling unexpected events, as shown in Figure 2.

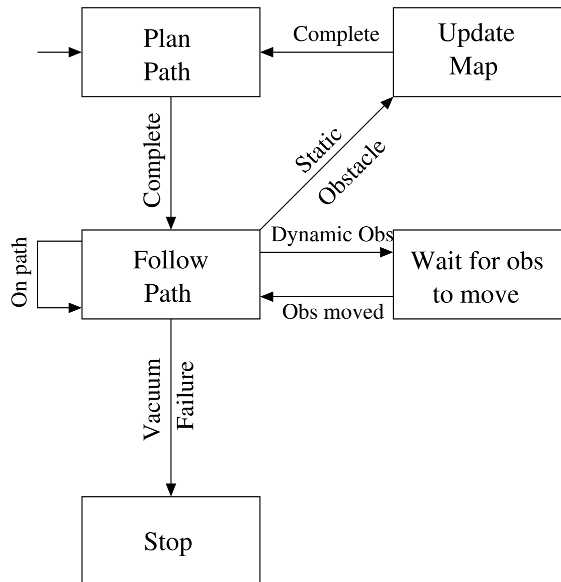

Figure 1: The behaviour of the simple vacuuming robot.

2.1 Expected events

There are four methods for handling expected events. These are event loops, languages with reactive semantics, behavioural systems and hybrid systems.

Event Loops

An event loop is simply an infinite loop that checks once per iteration (or more if necessary) for any pending events and then calls appropriate handlers for each event. Event loops are most commonly found in software programs that use a graphical user interface (GUI), as these are by nature reactive to user input. An event loop calls the correct function to handle an event based on the event's identifier. The function to be called for each event may be coded manually into the loop or specified using a call back. While event loops provide a simple method of simulating reactivity, they do have some drawbacks.

The first problem with an event loop approach is that it is not capable of handling complexity. As the program gets larger and more complex, the number of events also gets larger. This can lead to more complex event loops and increased difficulty in maintaining and expanding the system.

The second problem is handling large computational tasks. Large tasks, such as path planning, cannot be called from within the event loop without causing it to become unresponsive to events until the task is complete. The two common ways to get around this problem are to break up large tasks into smaller parts that can be called each iteration of the loop when there are no pending events, or to utilise a separate thread for the event loop

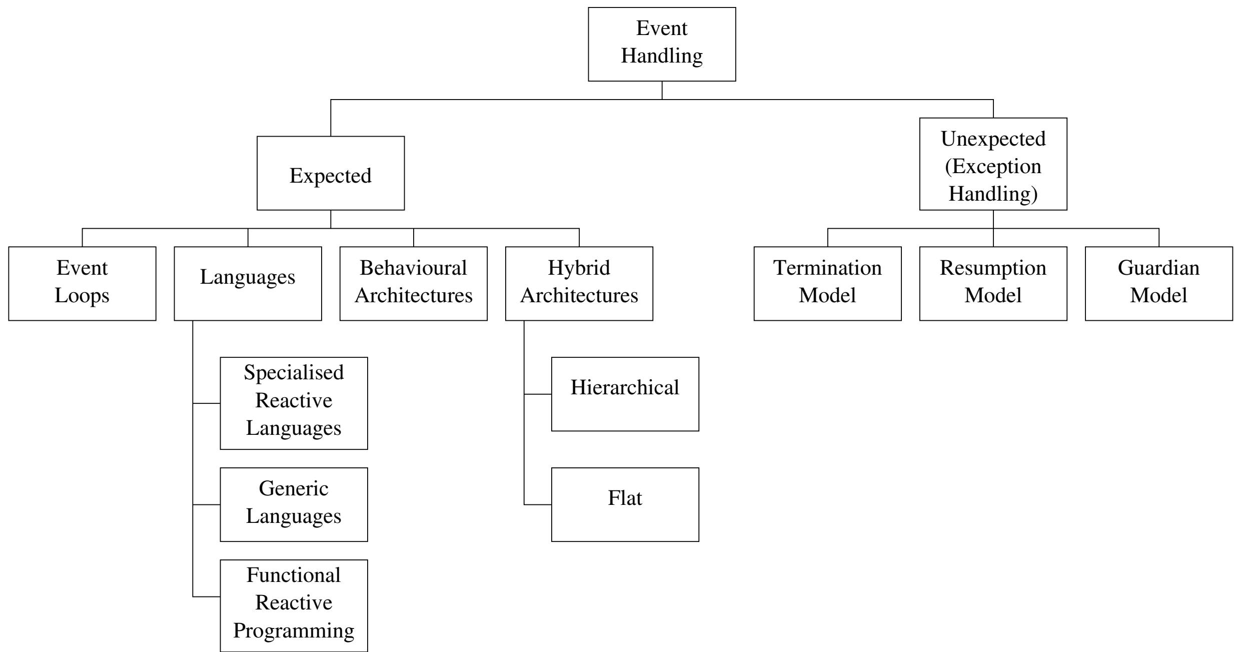

Figure 2: Event handling systems can be divided into two major cateogries.

from other parts of the application. In both cases the program becomes harder to write, debug and maintain.

Example The vacuum robot would begin by planning the path to take around the room. Then it would begin following this path and start an event loop watching for the three relevant events shown in Figure 1. Upon encountering the static obstacle event, there are two options. Firstly, the robot could wait for the planner to replan the entire path before resuming the event loop. Secondly, a well-designed planner could plan the initial stages of the path first, allowing the robot to begin moving again quickly while it replans the rest of the path as necessary. Such a planner would be run in a separate thread, probably posting an event back to the event loop when enough of the path is planned for the robot to begin moving again. The difficulty here is the need to manage a separate thread for the planner.

Languages for reactivity

There are a number of languages available for creating reactive systems. A good example of such a language is Esterel [Esterel, 2005]. Esterel provides special syntax for specifying inputs and outputs, signals between concurrent threads of execution, and even the threads themselves. It divides time into ticks, and in each thread of execution a single statement takes a single tick. Esterel cannot be executed directly; it is compiled into another language such as C, which is then compiled to produce a binary.

The drawback of these languages is that they are specific to reactive systems. While they are good for specifying, for example, an automatic teller machine, they are not as useful for robotic systems where reactivity is not the only concern.

To improve on this, there have been some attempts to allow the specification of reactivity in general purpose languages. The creators of these systems hope to improve the use of reactivity in more general systems.

For example, Boussinot [1991] adds reactive Esterel-like concepts to C. The author aims to show that reactive programming can be done in C in a natural way with only a few new concepts. The concept of ticks is mimiced here. This preserves determinism while allowing parallelism, and ensures that the termination time of a process can be known (in other words, the end of the current instant). Some useful statements are provided for thread management. A "watching" statement is provided to allow a body to be killed in the instant the condition being watched evaluates to true. The system also provides "last will" statements to provide a way to cleanup when a body is terminated but not when it exits normally.

Functional Reactive Programming is based on functional programming, which follows an evaluative approach rather than focusing on order of execution [Tucker, 2004]. It is well suited to creating reactive systems, as it works on the concept of producing outputs by evaluating inputs. Yampa [Hudak *et al.*, 2003] and Frob [Hager *et al.*, 2001; 2002] are two similar examples

of FRP languages designed specifically for robotics.

Frob adds pre-defined controller and sensor interfaces, a communications infrastructure a simulator, visualisation, and specialised versions of FRP services such as tasks to FRP. It is a thin layer over FRP, the user sees much of the underlying FRP system. "Tasks" are used to create sequentiality. They produce a continuous control output while running and another value on termination. Reactive components are treated as first class entities in the FRP framework (i.e. objects), which the authors say allows for a very flexible and adaptible system structure.

GRL [Horswill, 2000] is an FRP language aimed at behavioural systems. It can be used for many kinds of behavioural architectures, including the subsumption architecture and motor schemas, showing the versatility of this approach.

[Dai *et al.*, 2002] describes the implementation of Functional Reactive Programming (FRP) in C++ using templates and macros. The developed system is simpler to use than Haskell. However, it is still very complex and an extensive knowledge of C++ template programming and functional programming is required to use it, showing one of the common problems with the functional programming approach; they are difficult to use.

While FRP provides a highly formal approach for programming reactive systems, functional programming in general has difficulties with exceptions. Govindarajan [1992; 1993] states that this is because traditional exception handling is incompatible with the evaluative approach of functional programming. To solve this, they discard the control flow concept of exception handling and instead focus on the objects involved in the exception. There are also difficulties with the functional programming model used in FRP, as it is significantly different to the imperative model currently popular.

Behavioural Architectures

Behavioural based architectures are robot architectures that use purely reactive systems to control robots and achieve goals. Reactive systems maintain an ongoing relationship with their environment, constantly responding to changes in it [Boussinot, 1991]. The important aspect of a behavioural system is how it selects which behaviour will control the robot; the coordination method. Robot programs are broken down into a set of behaviours that co-operate or compete to control the robot. These behaviours can commonly be represented as simple Finite State Machines (FSMs).

There have been many behavioural architectures developed, beginning with Brooks' subsumption architecture in 1986 [Brooks, 1986]. A robot program is broken up into a series of simple behaviours divided horizontally into layers, as shown in Figure 3. At the lowest level are the simplest layers. In each successive layer are more complex behaviours that subsume those below them by supressing their inputs and outputs in order to control them. The behaviours are typically implemented as Augmented Finite State Machines (AFSMs).

The advantage of this approach is that it is highly parallelised and custom behaviours can be created easily and implemented directly in hardware. It also strongly supports incremental design due to its layered architecture. However, it can be difficult to adapt behaviours for use in other systems since higher layers usually depend on the lower layers.

The main alternative behavioural architecture is motor schemas, based on schema theory [Arkin, 1998]. The outputs of many concurrent schemas are combined into a single output, usually using a vector addition model. The vector addition model is the coordination method. Motor schema architectures are typically quite modular and easily adaptable to other systems. They are also easily reconfigured at run-time.

Other behavioural architectures developed over the years include:

1. Action-Selection, using an "activation level" to decide which behaviour is in control [Maes, 1990].
2. Colony Architecture, a simplified version of the Subsumption architecture that allows for ordering of behaviours in a tree rather than a layer [Connell, 1989].
3. The Distributed Architecture for Mobile Navigation (DAMN), which uses a voting system to coordinate between behaviours. Behaviours can cast their votes in various ways, for example different statistical distributions [Rosenblatt, 1997].

Many architectures provide a special language for defining behaviours. For example, in the Subsumption architecture, behaviours are typically implemented as AFSMs, and there is also a specially-designed Behaviour Language for specifying these AFSMs that is compiled down to the actual AFSMs that are implemented in hardware or software.

One of the biggest advantages of behavioural architectures is that they are highly parallelised. Many of them, notably the Subsumption architecture, can also be implemented directly in hardware, due to the state machine nature of their behaviours. Another advantage of this style of architecture is that they can provide very rapid responses to changes in a robot's environment and status.

However, because of their use of simple behaviours there is considerable debate over whether behavioural architectures can achieve long term and complex goals. There may also be issues with the flexibility and learning capability of these systems.

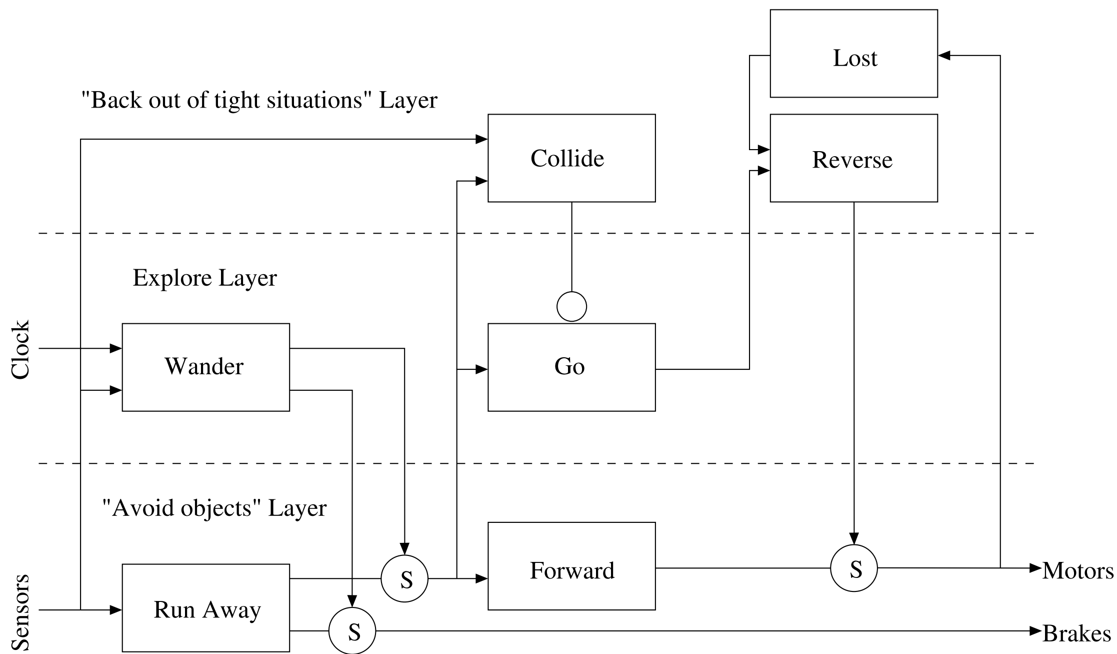

Figure 3: A simple three-layered robot in the Subsumption architecture [Arkin, 1998].

The specification of the behaviours is also an issue. A robot program design must be broken down into small component behaviours to be implemented in a behaviour architecture. While many behavioural architectures provide a language for defining behaviours, these are architecture specific. As mentioned in [?], architecture specific languages are not suitable for modern robotics.

Example The vacuuming robot as described in section 1.1 would not be implementable in a behavioural system because of the inability to plan. Instead, a behavioural vacuuming robot would simply move at random around the room, triggering different behaviours when it encounters certain events. A good example of a behavioural vacuuming robot is the Roomba robot from iRobot [iRobot, 2005]. This robot has various behaviours such as backing up and turning to avoid an obstacle, wall following, and boucing off walls randomly.

Hybrid architectures

As mentioned in section 1.1, behavioural architectures may not be suitable for achieving complex or long term goals. One proposed solution is to use hybrid architectures, which attempt to combine the best parts of long term planning and short term reactive systems. In this way, roboticists hope to take advantage of reactive systems' advantages in dealing with dynamic environments while still being able to plan ahead. The behavioural and planning components are each written in the language suited for them, for example a behavioural language and a cognitive system.

There are two typical structures for a hybrid architecture, hierarchical and flat. A hierarchical structure typically uses high–level planners and low–level behavioural components. Communication is between layers. A flat structure resembles a network of components communicating with each other directly.

The Autonomous Robot Architecture (AuRA) was one of the first hybrid architectures [Arkin, 1998]. It uses a deliberative heirarchical planner and a schema theory based reactive controller. The planner converts paths into a sequence of behaviours, then sends these to the reactive component for execution.

By contrast, the Atlantis architecture developed at the Jet Propulsion laboratory relies on the deliberative component being called only when necessary by a sequencing component [Gat, 1991]. It is an example of a hierarchical hybrid architecture.

The Planner-Reactor architecture uses a planning component to continuously reconfigure a reactive component as it is executing, allowing it to respond rapidly to the environment and changes in plans [Lyons and Hendriks, 1992; 1995]. The Procedural Reasoning System, on the other hand, uses a least-commitment strategy to delay creating complex plans for as long as possible. Plans are thus created in response to the immediate state of the robot. [Georgeff and Lansky, 1987]

A recent development, the BERRA architecture, is aimed at service robotics [Lindstrom *et al.*, 2000]. This system uses a deliberative layer and a reactive layer. The

deliberative layer is also responsible for communication with an operator. A third layer, the task execution layer, acts as a bridge between these two layers.

The benefit of hybrid architectures is that they can allow for both long term goal planning and quick reaction to the changing state of the robot. They can also be seen as more closely mimicing the concepts of high level intelligent control and low level instinctive reactions seen in nature. However, they do have some drawbacks. Firstly, the natural barrier between the two controllers raises real–time issues when a more complex response is needed quickly. They also generally require the use of separate development and programming models for the separate controllers; a high-level AI language for the planners and a behavioural language or some other form of specification for the behavioural components, increasing development difficulty.

Example The vacuuming robot could be implemented quite naturally in a hybrid architecture by using a behavioural component and a path planner component. The path planner would be called once at the start to provide an initial path to follow. The behavioural component could take over and follow the path, responding as necessary to the events depicted in Figure 1. When a static obstacle is encountered, the path planner component would again be invoked to provide a new path.

2.2 Unexpected events

Even in standard software programming, it is not possible to think of all possible situations. Because robots operate in the unpredictable real world, the problem is compounded significantly in robotics. As such, it is very important that unexpected events are handled just as gracefully as expected events by robotic systems.

Exception handling is the method for handling unexpected events in programming. The programmer can specify once for a block of code how to handle an error, rather than manually checking the return value of each function as it is called. This can improve code clarity and reduce programming time. Exceptions can also be used to provide more information about an error. Most exception mechanisms allow for entire objects to be propagated rather than just a signal, thus allowing information about the error that occured to be carried in the object and used in the exception handler.

Exception handling methods can be divided into those that use the termination model, in which the block that raises the exception is terminated in favour of the handler, the resumption model, in which the block that raises the exception is resumed once the handler has completed, and those that use a relatively new concept called a guardian. In the resumption model, the raising block may be resumed at the point at which it raised the exception, presumably with the cause of the exception corrected, or it may be restarted from the beginning.

In robotics, the issue of handling exceptions is often considered a problem for a planner to deal with. The planner must recreate or alter its plan to achieve a goal when a problem arises. There have been some pieces of work on adding specialised exception handling to robotics. Some examples are given below.

Cox and Gehani [1989] present an early work on the use of exceptions in robotics using Exceptional C. The authors state that there are two important aspects of exception handling in robotics. The first is the need to handle errors in real–time, the second is the need for a heirarchy of handlers, in that solutions that take a small amount of time should be tried first (for example, wait for an obstacle to move) before trying more drastic actions (replan a path around the obstacle). They also state that the alternative style of error handling, that of checkpointing (a form of resumption), is not always suitable to robot applications because it may not be possible to return the real world to a previous state.

Simmons and Apfelbaum [1998] describe a "Task Description Language," in which exceptions are associated with a given node in the architecture and a reason for their occurance. Propagation is accomplished by searching up the tree to find a matching node for the exception. That node then becomes the handler, and can reorganise the tree to try and fix the error because the call stack is not popped when the exception is propagated.

Gluer and Schmidt [2000] advocate the use of relational algebra to specify rules for exceptions rather than the traditional rule set approach. The reason for this is because rule sets can become very large and cumbersome in complex systems. As the system grows, hidden interdependencies between rules can develop that make it difficult to manage the list, including adding, deleting and re–ordering rules. The authors propose representing knowledge in tables of relations instead to give it a structured form, similar to a knowledge database.

Bruccoleri *et al.* [2003] discuss the need to handle unexpected events in reconfigurable manufacturing systems. They focus on using reconfiguration to handle exceptions in an agent-based system. They find that it improves global system performance under certain conditions.

The standard exception handling mechanism of throwing an object and catching it elsewhere is, while syntactically neat, semantically messy. It breaks control flow, which can lead to difficult program maintenance if good documentation practices are not followed. In addition, it can leave objects or the system in an inconsistent state. This can be particularly bad in robotics because they deal with the real world. Consider, for example, what would happen if an exception occured while a manipu-

lator was carrying an object, and the handler, unaware that an object was being carried, opened the manipulator. It would be essential that a cleanup procedure be followed, so that the object is put down for example, whether the carrying procedure were executed normally or via an exception.

There is also work in developing exception handling systems for distributed systems. As robotics often involves distributed systems, these can be considered to be relevant. The difficulty with handling exceptions in a distributed system is that there are usually many processes running concurrently and involved in a single task. When an exception occurs, these processes must all be notified and must work together in a coordinated fashion to handle the error. A common approach is the use of Coordinated Atomic (CA) actions to divide the work load up into specific chunks that can then be handled individually. The processes involved in each of these actions are known and so they can be coordinated effectively when the CA action fails. Xu *et al.* [1998] and Romanovsky *et al.* [1998] describe a system using this concept. For example, in robotics a coordinated action could be gripping an object with a manipulator.

Fetzer *et al.* [2004] discuss atomicity of methods in terms of if they leave objects in an inconsistent state or not when throwing an exception. They classify methods as failure atomic or nonatomic depending on whether or not they preserve state, and say that exception handling is atomic if it ensures failure atomicity. This methodology could work well in robotics, where actions could be classified based on how they affect the world if an error occurs while they were being performed. For example, a navigation action would likely not affect the world other than changing the robot's position, whereas a manipulation task will change the layout of objects in the world and thus the state of the world.

Miller and Tripathi [2002] discuss the concept of a "guardian" for distributed exception handling. The guardian is a form of global exception handling. It uses a distributed global entity to control each process involved, usually by raising an exception in each one. The exception raised may not be the same as the original exception. The guardian model has "exception contexts" (an execution phase or region of a program), a global "guardian" entity (which has a guardian member for each process, as a co-process), and a set of guardian primitives used by the participant processes. Recovery rules control how the guardian manages processes when responding to exceptions.

Example Exception handling is important to the vacuuming robot so that it can handle errors such as hardware failures. For example, if the vacuuming pipe were to be blocked by a large object, the robot would not be able to complete its task. This should raise an exception, causing the robot to call for help. Another example would be the failure of a sensor. In this case an exception should be handled that would allow the robot to continue its task using the remaining sensors. The system should reconfigure itself to handle the exception.

3 Trends in specifying reactive robotic behaviour

3.1 The present

It can be seen from the work summarised in section 2 that the current status of programming reactive robots is less than ideal. The first problem is that existing systems focus on either expected or unexpected events. This complicates robot programming as two separate methods must be used for a complete program.

Hybrid architecture developments are improving. They use the right tool for each job (long term planners for long term goals, behavioural components for quick reactions). There are, however, still issues to be solved with the differing methods of programming the different components.

The lack of a good generic language for programming robot reactivity is an important issue at this time. Current languages are either too complex, as in the case of FRP languages, or architecture specific, as in the case of behaviour specification languages.

Current exception handling methods are not entirely suited to robotics. For example, it is not possible to safely use the resumption model because the real world cannot easily be returned to a previous state. There are also issues with the use of the termination model because it can leave the system in an inconsistent state, which is particularly dangerous in robotics.

3.2 The near future

In the next few years, hybrid architectures are likely to become increasingly popular because they address issues with both long term control and quick responses.

Along with this, we expect to see better development systems for hybrid architectures, including the languages used. However, the gap between development of the deliberative and reactive components seems likely to remain.

Despite these advances, it does not seem straightforward for reactive systems to consider unexpected events.

3.3 The distant future

Even though it is nearly impossible to say what shape robotics will take in even just a decade's time, it is none the less interesting to consider how important reactivity will be and how it will be specified. It seems likely that biologically inspired systems will become the popular method of programming robots. These may use

low-level instinctive components, essentially similar to behavioural systems of today, and high-level intelligence. In other words, an extrapolation of today's hybrid architectures. Specifying reactivity is clearly going to remain an important issue.

3.4 The next step

What should be the next step to move towards the programming methods needed in the future? To answer this, we need to know what is missing.

The most important gap is the ability to specify both expected and unexpected events with a single programming construct. The two areas of handling exceptions and reacting to events are very closely related. They are, ultimately, all events. In robots, an exception can be seen simply as another event that must be reacted to, in other words they can be classed as "bad" events. Given the degree of uncertainty and asychronicity in mobile robot environments and the vagaries of the real world that the robot is closely tied to, the difference between expected and unexpected events is more blurred. While a robot's response to a "bad" event may be significantly different from its response to any normal event, the method of programming the response may be no different in either case. It would further our goals of making programming easier if a single programming construct could be used for both exception handling and reactivity.

Given that hybrid architectures are likely to become increasingly important in the future, the other issue is the ability to specify all parts of the architecture with a single language. We feel that a general language with reactive components is a good start and that this is a worthwhile direction for further research and development.

So the next step should be to develop a general-purpose language with simple, clear semantics for specifying reactions to events. It should handle both expected and unexpected events within the same mechanism. It should be suitable for programming various levels of robot software, from low-level, simple components to the high-level components. It should be usable by a typical robot developer, not just artificial intelligence experts.

4 Conclusions

This paper reviews the literature, issues and trends for specifying expected and unexpected event handling in a robotics programming context. It is important to provide convenient programming language constructs for specifying event handling in robots, which must handle a wide variety of expected and unexpected events, with uncertain timing and values as a result of interactions with the real world. Hybrid deliberative/reactive methods are likely to develop further, but suitable languages are needed for giving a coherent description of hybrid methods, that also includes exception processing for unexpected events.

References

[Arkin, 1998] Ronald C. Arkin. *Behavior-based robotics.* Massachusetts Institute of Technology, 1998.

[Boussinot, 1991] Frederic Boussinot. Reactive C. An extension of C to program reactive systems. *Software - Practice and Experience*, 21(4):401 – 428, 1991.

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

[Bruccoleri *et al.*, 2003] Manfredi Bruccoleri, Michele Amico, and Giovanni Perrone. Distributed intelligent control of exceptions in reconfigurable manufacturing systems. *International Journal of Production Research*, 41(7):1393 – 1412, 2003.

[Connell, 1989] J.H. Connell. A behavior-based arm controller. *IEEE Transactions on Robotics and Automation*, 5(6):784–791, 1989.

[Cox and Gehani, 1989] I.J. Cox and N.H. Gehani. Exception handling in robotics. *Computer*, 22(3):43–49, 1989.

[Dai *et al.*, 2002] Xiangtian Dai, G. Hager, and J. Peterson. Specifying behavior in C++. In *Proceedings of the IEEE Intl. Conf. on Robotics and Automation (ICRA '02)*, volume 1, pages 153–160, May 2002.

[Esterel, 2005] The Esterel Language. http://www-sop.inria.fr/meije/esterel/esterel-eng.html, 2005.

[Fetzer *et al.*, 2004] C. Fetzer, P. Felber, and K. Hogstedt. Automatic detection and masking of nonatomic exception handling. *IEEE Transactions on Software Engineering*, 30(8):547–560, 2004.

[Gat, 1991] Erann Gat. *Reliable Goal-directed Reactive Control for Real-world Autonomous Mobile Robots.* PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1991.

[Georgeff and Lansky, 1987] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *AAAI*, pages 677–682, 1987.

[Gluer and Schmidt, 2000] D. Gluer and G. Schmidt. A new approach for context based exception handling in autonomous mobile service robots. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 4, pages 3272–3277, April 2000.

[Govindarajan, 1992] R. Govindarajan. Software fault-tolerance in functional programming. In *Computer*

Software and Applications Conference, 1992. COMPSAC '92. Proceedings., Sixteenth Annual International, pages 194–199, Chicago, IL, 1992.

[Govindarajan, 1993] R. Govindarajan. Exception handlers in functional programming languages. *IEEE Transactions on Software Engineering*, 19(8):826–834, 1993.

[Hager *et al.*, 2001] G. Hager, J. Peterson, and A. Serjentov. Composable robot controllers. In *Computational Intelligence in Robotics and Automation, 2001. Proceedings 2001 IEEE International Symposium on*, pages 149–154, 2001.

[Hager *et al.*, 2002] Gregory Hager, Henrik Nilsson, and Izzet Pembeci. Functional reactive robotics: an exercise in principled integration of domain-specific languages. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 168–179, New York, NY, USA, 2002. ACM Press.

[Horswill, 2000] Ian Douglas Horswill. Functional programming of behavior-based systems. *Autonomous Robots*, 9(1):83 – 93, 2000.

[Hudak *et al.*, 2003] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[iRobot, 2005] irobot - Robots for the Real World. http://www.roombavac.com/, 2005.

[Lindstrom *et al.*, 2000] M. Lindstrom, A. Oreback, and H.I. Christensen. BERRA: A research architecture for service robots. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 4, pages 3278–3283, San Francisco, CA, 2000.

[Lyons and Hendriks, 1992] D.M. Lyons and A.J. Hendriks. Planning for reactive robot behavior. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pages 2675–2680, Nice, 1992.

[Lyons and Hendriks, 1995] D.M. Lyons and A.J. Hendriks. Planning as incremental adaptation of a reactive system. *Robotics and Autonomous Systems*, 14(4):255 – 288, 1995.

[Maes, 1990] P Maes. Situated agents can have goals. *Robotics & Autonomous Systems*, 6(1–2):49–70, June 1990.

[Miller and Tripathi, 2002] R. Miller and A. Tripathi. The guardian model for exception handling in distributed systems. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pages 304–313, 2002.

[Romanovsky *et al.*, 1998] A. Romanovsky, J. Xu, and B. Randell. Exception handling in object-oriented real-time distributed systems. In *Object-Oriented Real-Time Distributed Computing, 1998. (ISORC 98) Proceedings. 1998 First International Symposium on*, pages 32–42, Kyoto, 1998.

[Rosenblatt, 1997] J.K. Rosenblatt. DAMN: A Distributed Architecture for Mobile Navigation. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2–3):339–360, April-September 1997.

[Simmons and Apfelbaum, 1998] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 3, pages 1931–1937, Victoria, BC, 1998.

[Tucker, 2004] Allen B. Tucker, editor. *Computer Science Handbook*, chapter 92. Chapman & Hall/CRC, 2nd edition, 2004.

[Xu *et al.*, 1998] J. Xu, A. Romanovsky, and B. Randell. Coordinated exception handling in distributed object systems: from model to system implementation. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, pages 12–21, Amsterdam, 1998.