The 2010 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 18-22, 2010, Taipei, Taiwan

Implementing a reactive semantics using OpenRTM-aist

Geoffrey Biggs
Intelligent Systems Research Institute
National Institute of Advanced Industrial Science and Technology
Tsukuba, Ibaraki, Japan
geoffrey.biggs@aist.go.jp

Bruce A. MacDonald
Department of Electrical and Computer Engineering
University of Auckland
New Zealand
b.macdonald@auckland.ac.nz

Abstract—**The expression of reactive behaviour is a significant and important requirement in robotic software engineering, since robots must cope with a wide range of unpredictable events and environments. However it is important that the semantics for reactive expression can be used across different architectures and languages. The RADAR robot programming language provides architecture– and language–independent semantics for managing the reactive parts of robot software together with the deliberative parts, allowing greater interaction between the two. We evaluate the architecture-independence of RADAR, as an example, by implementing its reactive semantics using the OpenRTM-aist component-based, distributed architecture. Our goal is to evaluate what limitations the choice of implementation environment may place on the capabilities of such an architecture-independent semantics. In our implementation, we aimed to produce a standard OpenRTM-aist system using the RADAR semantics. We have found that the architecture-independent semantics concept works well in the case of RADAR, although some specific improvements are needed for full interaction between deliberative and reactive sections of robotic software.**

I. Introduction

A key area when programming a robot is reactivity, the recognition of and response to events that occur in the real world. A recent application-specific robot programming language, RADAR, provides a semantics for specifying robotic reactivity events and responses from within deliberative robot code [1]. This prevents barriers to close interactions between the two parts of robotic software. The semantics allow for greater ease-of-coordination between the deliberative and reactive parts of a robot program, and in particular eases managing the behaviour of reactive parts when the program state changes.

RADAR is a set of language extensions designed for robot developers. RADAR was designed to be language- and architecture-independent. Rather than being designed for a specific architecture or tied to a particular language, the semantics are designed so as to be applicable to and implementable in a range of robot software architectures across a range of architecture styles and using any of the popular programming languages of the time. This facilitates robot developers replicating the same or similar concepts across languages and architectures. This is important for making robots more programmable.

In this paper, we present an evaluation of the architecture-independent part of the design. We have implemented the RADAR semantics for reactivity management using the OpenRTM-aist architecture. RADAR semantics have already been tested using a Python implementation based on Python threads. This is a monolithic programming method, while, by contrast, OpenRTM-aist is a distributed, component-based architecture. By implementing in OpenRTM-aist, we aim to test how effective the semantics are in a different style of architecture, and so evaluate how effective the concept of architecture-independent semantics is. We evaluate what limitations occur in the use of the implemented semantics, and so our implementation targets the creation of standard OpenRTM-aist systems using RADAR semantics. This allows us to discover where the implementation environment, that is, OpenRTM-aist, limits the capabilities of the semantics. We have noted, where relevant, how these limitations can be overcome. This analysis suggests design issues that reactive semantics should address in robotic software engineering.

Section II briefly describes the RADAR semantics for reactivity, including the original implementation. Section III discusses OpenRTM-aist in order to provide a background for the implementations discussed in this paper. The following sections describe the implementation design. Discussions are given in Section VII, followed by conclusions.

II. RADAR

RADAR's reactivity semantics are based on the signals and slots found in Boost [2] and Qt [3]. RADAR uses a structured version of this concept. A more detailed description can be found in [1] and [4]. A simple example is shown in Listing 1.

RADAR's reactivity semantics are focused on three key concepts of reactivity: ***events***, ***responses*** and ***connections***. It uses special objects to represent two of these, ***event objects*** and ***response objects***. The third concept is represented by special statements that can be used from the deliberative parts of the program.

Event objects encapsulate a condition check, representing conditions that may occur during program execution. When an event occurs, its ***triggered*** signal is emitted.

Response objects encapsulate behaviour that is executed when the response is activated, typically by an event occurring. They have three slot/signal pairs: ***start***, ***interrupt*** and ***exit***. The start slot represents the main behaviour of the response, and is where execution begins. When the response is interrupted before completing naturally, execution moves to the interrupt slot. Execution always finishes with the exit slot, no matter how the response reaches this point. The start and interrupt slots emit the ***started*** and ***interrupted*** signals, respectively, when they begin execution. The exit slot emits the ***exited*** signal *after* it completes executing.

The connection statements are used to manage connections between events and responses, both from deliberative code and from reactive code. ***Event expressions*** are used to reference events, while ***response expressions*** are used to reference responses. These expressions can define the flow of data along signals. They are used in conjunction with the ***once*** and ***whenever*** statements, which are the core statements for forming connections between events and responses. They are used to specify that when an event occurs, the response slot that it is connected to will begin execution. The once statement specifies a once-only connection, while the whenever statement specifies a persistent connection. Which slot of a response to connect to is specified by a keyword, ***start***, ***interrupt*** (or ***until***) or ***exit***.

A. Original implementation

The original implementation of the RADAR semantics used Python threads. A short example of its use shown in Listing 1. Figure 1 illustrates the objects of this RADAR implementation.

This implementation instantiated responses as individual Python threads, using accessor functions with locks to implement the slots. Each response executed in its own thread of control, relying on Python's thread scheduling. A response's thread slept on a coordination object, used to indicate that it had received the `start` signal and so should begin executing. Active events were all executed in a single Python thread, in sequence, within an object known as the "Event Manager". Signals between the events, responses and deliberative parts of the program were implemented in a "Connection Manager" object. This tracked who was listening for what signals, and when a signal was emitted, would inform the listeners.

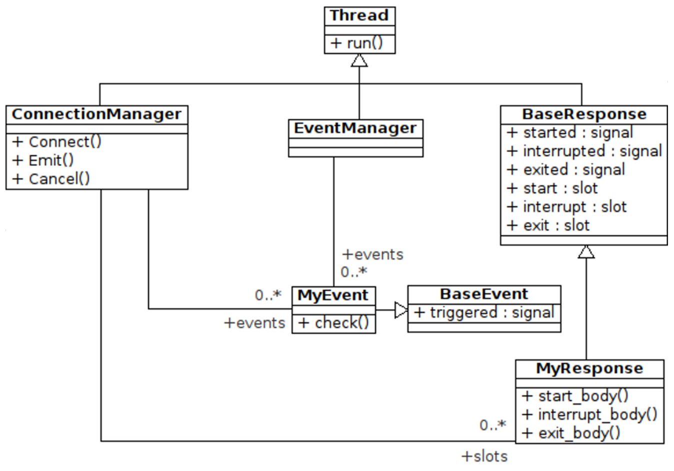


Fig. 1: The objects used by the Python implementation of RADAR.

Listing 1: A simple RADAR/Python example.

```
response CountUp:
  i = 0
  while i < 10:
    sleep(0.5~s)
    i += 1
    interrupt
  on exit:
    restart

event StartEvent:
  sleep (2~s)
  trigger

event EndEvent:
  sleep (4~s)
  trigger

def main:
  countUp = CountUp()
  start_event = StartEvent()
  end_event = EndEvent()
  once start_event start countUp until
      end_event
```

III. OpenRTM-aist

OpenRTM-aist is a framework for distributed intelligent systems [5]. Its main feature is software components distributed over a network. Diverse components, including components by different software vendors, can interact and be combined to produce a larger, more complex robotic system.

The core concept of RT-Middleware is the RT-Component. Each component contains a state machine with states "inactive," "active," and "error." This state machine is universal, allowing all components to be controlled at a high level. Components feature input and output data ports. The choice of what data ports a component provides is up to the component designer. Components are connected into networks to form complete systems. These networks are called RT Systems.

OpenRTM-aist was selected for this work because of its highly-dynamic capabilities not found in many other architectures, even other modern, component-based architectures. The ability to externally alter component connections at run time was key to implementing the RADAR features.

IV. Implementing RADAR

Our OpenRTM-aist implementation of RADAR uses distributed software components directly, as these are the key elements of OpenRTM-aist. Throughout the design, we have aimed to have the RADAR semantics create standard RT Systems using only standard OpenRTM-aist facilities. We do this in order to evaluate what restrictions the implementation environment may place on architecture-independent semantics.

We have also attempted to maintain the distribution capability of OpenRTM-aist, as this is a key feature of the architecture. Although the RADAR semantics do not explicitly support distribution of events and responses across a network, they also do not explicitly prevent it. We investigate how usable they are in a distributed environment, and what limitations the distributed environment may place on their use.

A. RADAR objects in OpenRTM-aist

Our design uses RT Components to implement the RADAR event and response objects. Because RADAR does not explicitly define how event and response objects exist in a running system, including on which computing resources and in which memory spaces, we are able to directly apply the semantics for them to RT Components. This applies equally well when we distribute the event and response components across a network as when they are executing on a single, local computing resource. We consider this to be an important benefit of the RADAR semantics.

In the original RADAR implementation, event and response objects are always created as local objects within the scope of the deliberative program. There is no support for spreading events and responses across a network. By contrast, our implementation, by using RT Components to implement these objects, gains the ability to distribute them in not just different programs, but across separate computing resources on a network. We can instantiate the components as part of the local program, instantiated by the deliberative code, or rely on the developer to instantiate them manually on the appropriate computing resources. This is a benefit of implementing the semantics in a distributed architecture.

There is no direct support for specifying where components should execute in the RADAR semantics, so we have implemented this as an option to the preprocessor. A switch chooses between instantiating all components locally, or generating them as separate programs that must be executed manually by the developer on the appropriate computing resources.

The instantiation of event and response objects as local objects in the original implementation, albeit objects that utilise separate threads for execution, allows the sharing of resources. For example, it is possible to copy (either directly or via a function argument in the signals and slot implementation) any object from the deliberative code into a response or a component, or from an event or response into the deliberative code. As the instantiated objects are part of the same program and on the same system, this capability extends to system resources, as well. The deliberative code can open a network socket, then pass that socket to a response (via one of its slots) for use during the response's execution.

We partially lose this capability in our implementation. Components instantiated as part of the local program, while they also run in a separate thread, do share memory and so retain the same capability to pass objects between the deliberative code and the events and response objects. However, components instantiated on separate computing resources from the deliberative program cannot so easily exchange data with the deliberative code. The exchange is limited to objects that can be passed over an OpenRTM-aist connection between two components (which is itself a CORBA connection). This prevents, for example, system resources being exchanged. In addition, the exchange is only one way; it is not possible to receive data back from an event or response object.

This restriction on data sharing is a direct consequence of using a distributed environment for implementation, causing components to exist in separate memory spaces. Distributed systems that are capable of automatically migrating between locations based on the resources they require, and sharing memory across distributed computing resources, do exist [6]. However, this capability is not currently present in OpenRTM-aist.

1) Implementation:

The difference between any two components that are used as event components by our implementation is typically just the code executed to perform the condition check. Likewise, responses typically differ just by the code executed in its three slots. To simplify the code generation for the preprocessor, we followed the lead of the original implementation and implemented base classes for events and responses.

These classes define component "patterns" defining the input and output ports of events and responses. Events feature just a single output port, corresponding to the "triggered" signal. Responses have three input ports, one per slot, and three output ports, one per signal. The preprocessor places the developer's code into callback functions that are called appropriately for event checks and response slots.

Both the event and response objects are self-activating. Events activate themselves when a connection is made to them, deactivating again once all connections are removed ("once" connections being removed automatically by the component after the condition check succeeds). This implements the RADAR concept of events only performing their check when a connection is present. Responses activate themselves on construction and remain active until destruction.

There is no support in RADAR semantics for defining extra signals or slots on events or responses. As a consequence of this, if the developer wishes for an event or response component to have additional input or output ports, they must be added manually. An alternative approach to this would be for the developer to specify that an event or response component should inherit from another component designed by the developer. Unfortunately, the RADAR semantics do not support inheriting events or responses from other objects. There is no way for the developer to specify this.

Support for transmitting data with signals is inherent in the use of OpenRTM-aist. Both signals and OpenRTM-aist's data connections are one-way transmissions carrying data. The RADAR signal concept maps naturally into OpenRTM-aist. This is a major benefit for the portability of the semantics.

Likewise, the events and responses map quite naturally onto OpenRTM-aist's component model, with output ports for possible signal emissions and input ports for slots.

B. Reactive/deliberative interaction in OpenRTM-aist

Interaction between the reactive and deliberative code in RADAR comes in two forms. The first is managing the responses of the system to events that occur. The second is exchanging data between the reactive and deliberative parts of the system. We have previously described how our implementation manages this second form in Section IV-A. In this section we discuss the management of system reactivity.

RADAR's concept of a connection between a signal and a slot maps well into OpenRTM-aist. If we treat a signal as an output port of one component and a slot as an input port of another component, then a RADAR signal can be represented by the connection between the input port and the output port – a core part of OpenRTM-aist's design.

RADAR's connection statements create connections between the signals and slots of its objects. In particular, they connect the slots of a response to the "triggered" signal of an event. In the semantics, this connection is not explicitly expressed as an object, but it does implicitly behave like one due to the scoping behaviour. The original implementation uses a Connection Manager to manage connections, recording them in a list and iterating over that list to look for connections that have an action to be performed.

Our OpenRTM-aist design treats a connection as its own object (connections in the OpenRTM-aist implementation are, in fact, their own object, but this is a separate concept). The actual connection underlying this object, between the signal and the slot, is an OpenRTM-aist connection between an output port and an input port.

RADAR's semantics specify that connections must obey program scope; a connection created within a function must be removed when that function terminates. By creating an object that represents and manages a single connection between two components, we are able to implement scoped connections in a natural way. The connection object is created when the connection needs to be established and deleted, destroying the connection, when the connection should be cancelled.

The connection statements of RADAR are therefore very useful for managing the component network. We are able to create and destroy connections between event and response components as execution moves through the deliberative code.

A side benefit of the definition of RADAR's semantics for the "once" and "whenever" statements allows arbitrary code to be executed in the body of these statements. This was used in the original implementation to effectively create anonymous slots in the middle of deliberative code, using data from that code's name space. A consequence of our implementation of connections as connections between two data ports on components, we are unable to support this feature of the semantics and still maintain our goal of creating standard RT Systems using RADAR syntax.

The RADAR semantics allow for objects to be passed from the deliberative code into slots. Because we use connections between components for triggering slots, we are unable to do this. This is partly because a connection can only carry data from an output port to an input port, and partly because the slot, being an OpenRTM-aist input port, can only accept one data type. The original implementation, which used function calls for slots, could accept an arbitrary number of objects of any type. Combined with the inability to execute arbitrary code in response to an event, described previously, our implementation can only pass data from events to responses.

1) Implementation: An obstacle we encountered during our implementation is RADAR's lack of explicit support for a distributed name space combined with OpenRTM-aist's lack of advanced deployment support. This restricted our options for specifying components that are distributed across the network from within the deliberative code. Ideally, we would use a component discovery system. However, OpenRTM-aist does not currently provide deployment and component discovery. We considered three alternatives:

- Specify that all distributed components must register on a single, known CORBA naming service. We would then be able to look them up by name, allowing the same method of specification as in the original implementation.
- Require that the developer specify full CORBA paths to each component.
- Use the `rtfind` tool from the rtcshell toolkit, part of OpenRTM-aist. A form of component discovery could be implemented, provided the developer specifies to rtcshell at deployment time which naming services will be used.

We settled on the third option, as it is more flexible. `rtfind` is used to search the known naming contexts for a component matching the name used in the RADAR code. The resulting full path is used in making connections. This method was chosen over the first because it is considered to offer greater flexibility. It was chosen over the second because that method was considered restrictive as it forced the developer to know in advance where each component would register, and also exposed OpenRTM-aist through the RADAR semantics.

The rtcshell toolkit is also used to manage the connections. `rtcon` and `rtdis` are called to make and break connections. They are passed the full paths in the distributed name space of the ports to connect. The `rtinject` tool is used to directly send a signal to a response's slot from deliberative code.

V. Design summary

Some changes in syntax can be expected when implementing RADAR in a new system. A summary of the mapping from RADAR concepts to OpenRTM-aist concepts used in our design is given in Table I. The objects that are created are shown in Figure 2.

VI. Example

The example in this section uses two events and one response. The first event counts up to 5, resets its counter, and triggers. It sleeps for one second between each count. The second event is similar, but triggers after 3 seconds. The response counts for 10 seconds after activating, then exits normally. It may be interrupted between each count.

Feature	Support
Event objects	Self-activating RT Components with a single output port.
Response objects	RT Components with three input ports and three output ports.
Signals	Connections between input and output ports, managed by objects in the deliberative program.
Creating connections	Tools from *rtcshell*.
waitfor statement	No support.
Arbitrary code in "once"/"whenever"	No support.

TABLE I: Implementations of RADAR concepts in OpenRTM-aist.

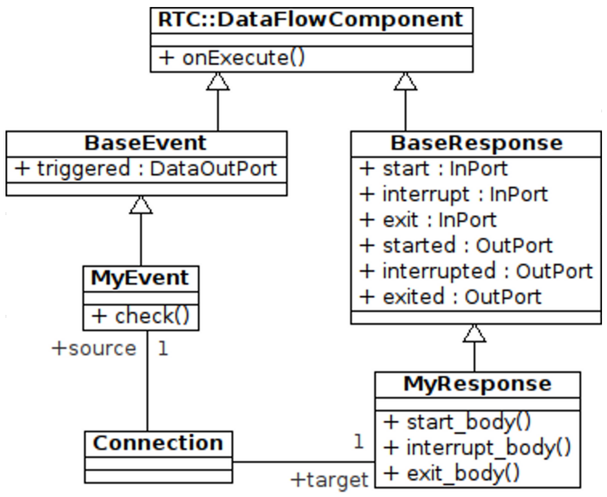


Fig. 2: The objects used by the OpenRTM-aist implementation of RADAR.

The deliberative code, which controls the connections in the system, is shown in Listing 2. The code output by the pre-processor is shown in Listing 3.

Figure 3 illustrates the changes in connections that occur as the deliberative RADAR code executes. Figure 3a shows the state immediately before the deliberative code begins executing. No connections are present. The response is active even now, as it is active throughout its lifetime.

Figure 3b shows the state after line 5 of Listing 2 (line 6 of Listing 3). The creation of the persistent connection causes the activation of the first event component. When execution moves into the called function, a once-only connection is created, leading to the system state shown in Figure 3c. This state will remain until the second event triggers or the function exits, at which point the system will again look like Figure 3b.

VII. Discussion

We first note that the core concepts of RADAR, event and response objects with connections between them, works well with OpenRTM-aist as the implementation environment. The mapping from event and response objects to RT Components, and the mapping of signals and slots to connections and ports on components, is very natural. This is despite RADAR not being designed specifically for OpenRTM-aist. We argue that this supports the idea of architecture-independent semantics and the idea that suitably-designed semantics can work across a range of architecture styles.

Listing 2: Example deliberative part using RADAR/OpenRTM-aist.

```
1  import sys
2  from time import sleep

4  def do_loop():
     once TestEvent2 interrupt TestResponse
6    for ii in range(15):
       sleep(1)
8
   def main():
10   whenever TestEvent start TestResponse
11   do_loop()
12   while True:
       pass
14
   if __name__ == '__main__':
16     main()
```

Listing 3: The deliberative part of a simple example, after processing to standard Python. The connection objects internally use `rtfind` to translate the target component name into a full path.

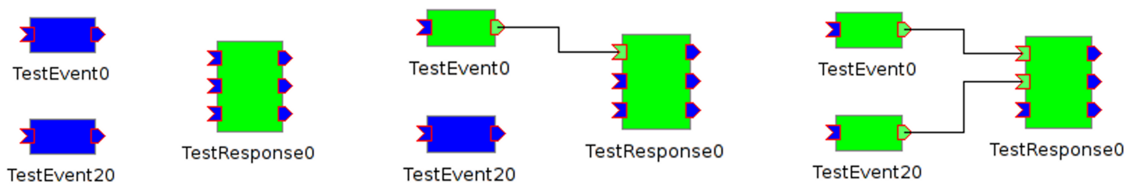
```
1  from radar.connection import Connection,
       OnceConnection
2  import sys
   from time import sleep
4
   def do_loop():
6      c2 = OnceConnection(
7          'TestEvent2:triggered',
8          'TestResponse:interrupt', 2)
       for ii in range(15):
10         sleep(1)

12 def main():
       c1 = Connection('TestEvent:triggered',
14         'TestResponse:start', 1)
       do_loop()
16     print 'Entering infinite loop'
       while True:
18         pass

20 if __name__ == '__main__':
       main()
```

Most limitations in the resulting system are on the use of the semantics, and originate in our choice of implementation environment, OpenRTM-aist.

- Anonymous slots in "once" and "whenever" statements are not possible because connections have been implemented purely as OpenRTM-aist connections between two data ports, and data ports exist on components.
- The ***waitfor*** statement is not supported for the same reason.
- There is limited interaction possible between the deliber-



(a) Start-up. (b) Stage 1. (c) Stage 2.

Fig. 3: The state of the event and response components in the example system at different times throughout execution of the deliberative code.

ative code and reactive code. This is a direct consequence of using RT Components, which cannot interact with deliberative code.

We do also note, however, that, just as OpenRTM-aist's feature set does not cover all of RADAR, the RADAR semantics do not cover all of OpenRTM-aist's features.

- If the developer chooses to distribute components across the network, they must start them manually, as RADAR is unable to specify distributed concepts (although it does not explicitly prevent them, either).
- Connection properties and component configuration values cannot be used via RADAR.
- Additional ports on the generated components cannot be specified using RADAR syntax, and must be added manually by the developer.

These problems are all solvable. We could create a hybrid of the original implementation and our OpenRTM-aist implementation to allow anonymous slots. We could extend the semantics to provide for inheriting events and responses from other objects to allow adding ports to the generated components.

These problems are generic ones that must be solved by any distributed model implementation of reactive semantics. The distributed model must provide rich enough components to enable constructs similar to "once," "whenever," and "wait-for," and rich interactions between code in different components. Component management methods should provide enough functionality to start components.

RADAR and OpenRTM-aist compare favourably with previous robot architectures. There is a long history of creating tiered architectures with an accompanying language. For example, the TDL language is used to program architectures that use a task tree [7]. By contrast, RADAR is designed to be architecture neutral, which this work reinforces. Mainstream architectures have tended to avoid including a language or specifying a structure. RADAR is shown in this work to be able to bring a structure of the programmer's choosing to a mainstream architecture (OpenRTM-aist) while still providing specialised semantics.

VIII. Conclusions

The RADAR robot programming language features semantics for reactivity. RADAR was designed to have semantics that are portable across programming languages and robot software architectures. This brings benefits to programming by increasing reuse of concepts and so reducing retraining time for developers moving between different robot software architectures. We evaluated this claim by implementing the semantics using the OpenRTM-aist architecture.

Concepts from RADAR map naturally onto OpenRTM-aist concepts. Event and response objects can be represented by RT components. Slots can be represented by input ports on these components. Signals can be represented by one-way connections between ports. We argue that this shows the concept of architecture-independent semantics to be sound.

The choice of implementation environment does place some limitations on the semantics. For example, our implementation is unable to support as much interaction between deliberative and reactive code. Conversely, the semantics may restrict the available features of the implementation environment. We are unable to, through RADAR syntax, specify the properties of a connection between an event and a response, even through OpenRTM-aist supports this. These short-comings are able to be overcome; we consider this to be future work. These issues will be faced by other distributed component implementations of reactive semantics.

This work brings a coordination language to a mainstream architecture, OpenRTM-aist, that did not previously have one, validating the architecture-neutral approach of RADAR.

References

[1] G. Biggs and B. MacDonald, "Specifying robot reactivity in procedural languages," in *Proc. IEEE/RSJ Int. Conference on Intelligent Robots and Systems*, Beijing, China, October 2006, pp. 3735–3740.
[2] "Boost.Signals," http://www.boost.org/doc/html/signals.html, 2006.
[3] "Trolltech - QT Product Overview," http://www.trolltech.com/products/qt/index.html, 2008.
[4] G. Biggs, "Designing an application-specific programming language for mobile robots," Ph.D. dissertation, The University of Auckland, 2007.
[5] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: Openrtm-aist," in *SIMPAR*, 2008, pp. 87–98.
[6] D. Stewart, R. Volpe, and P. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *Software Engineering, IEEE Transactions on*, vol. 23, no. 12, pp. 759 –776, Dec 1997.
[7] 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*, vol. 3, 1998, pp. 1931–1937.