Coordinating software components in a component-based architecture for robotics

Geoffrey Biggs, Noriaki Ando, and Tetsuo Kotoku

Intelligent Systems Research Institute
National Institute of Advanced Industrial Science and Technology (AIST)
AIST Tsukuba Central 2, Tsukuba, Ibaraki 305-8568, Japan

Abstract. Component-based software is a major design trend in robot software. It brings many benefits to system design, implementation and maintenance. One step in using component-based methods in designing the structure of a robot program is managing the components and the connections between them over time, known as coordination. In this paper we present a framework for coordinating component networks using the OpenRTM-aist software architecture, implemented using the concurrent Erlang language. The framework provides a coordination system that mimics the internal state-change notification system of OpenRTM-aist. Rather than being a fixed- structure coordinator, it allows robot developers to implement a coordinator matching the style of coordination they need. This paper shows that Erlang has potential in robotics.

1 Introduction

Component-based software design and implementation is a current trend in software engineering. Software is divided into individual components, each with a well-defined interface that specifies what functionality that component provides. Multiple software components are combined together into a complete software system [13]. Using this design methodology, robot developers can create complete robot systems from off-the-shelf software components as easily as complete electric circuits can be created from hardware components.

Component-based practices bring many benefits to software design, implementation, maintenance and reuse, including known interfaces that act as "contracts" between components, "separation of concerns" (each component only deals with its individual problem), isolation testing, and rapid development of new systems using existing commoditised software resources.

These benefits also apply to the design, implementation, maintenance and reuse of robot software. As a result, component-based software is a major trend in robotics.

An issue that all robot developers faces is the coordination of behaviours, and so in turn the coordination of the software. Coordination is important to allow the robot's software to adapt to changes in the robot's state as it carries out its various tasks. Before the recent rise of flexible component-based architectures, architectures with fixed structure, often layered, were popular in robotics. In these, a higher layer manages the actions of a lower layer to provide coordination according to some generated plan. Often, the plan itself is generated by an even higher layer, although some architectures, such as CLARAty [7], intentionally do not use this approach.

In this paper, we present a coordination framework for the OpenRTM-aist component-based architecture. It is a framework rather than a complete coordinator because it provides the facilities for programmers to create their own coordination systems. Rather than a fixed coordination style, programmers are free to use whichever style suits their needs. We use the concurrent Erlang language to implement the framework in order to test its applicability in robotics.

The next section discusses coordination methods commonly used in robotics. Section 3 describes OpenRTM-aist, the architecture for which the coordination framework has been designed. The coordination framework itself is described in section 4. Discussion is given in section 5, and conclusions in section 6.

2 Coordinating robot software

Coordination has a long history in robot software. It has always been necessary to manage the actions of a robot in order to achieve complex goals. It is particularly common to see coordination play a major role in layered architectures. Often, a layered architecture will feature a low-level layer that consists of chunks of functionality, and at a higher level some kind of coordination system controlling the execution of these chunks to meet some goal.

Despite their strong coordination support, no layered architectures have managed to become widely used.

On the other hand, recent years have seen the popularisation of more flexible component-based software architectures for robotics. These architectures allow designers to create *component networks*. Rather than being layered, the network of components is effectively a single layer of individual programs communicating by one or more methods of transporting data. Such an architecture style is very heterogeneous and adaptable to the needs of the programmer.

Examples of these architectures include OpenRTM-aist [1], ORCA2 [3], ROS [9], ERIC'S [4] and OPRoS [12]. In each case, systems built using the architecture rely on networks of connected components with data flowing amongst them. The behaviour of the robot is represented by what data ultimately arrives at the components responsible for controlling actuators. The shaping of this data, and so determining the robot's current behaviour, is performed by the components that make up the component network between sensor components and actuator components. The reader may notice that this is similar to coordinating the actions of a lower layer in layered architectures.

Coordination in a component-based system therefore requires changing either the internal behaviour of the individual components, or changing part of or the whole component network.

None of the architectures mentioned above currently provide an automatic coordination method to alter the component network at run-time. Only two of them provide facilities for manually altering a running component network. OpenRTM-aist and OPRoS, both at least partially based on the same software specification, allow for “ports” (the points on components at which communication occurs) to be connected and disconnected by external tools at run time.

The lack of a coordinator in general component-based software is understandable. The component-based design paradigm was originally aimed at static software systems, such as business software. A component-based system, once in place, was not expected to change. This is generally the case for component-based systems today. The need to dynamically alter a component network at run-time arose as the component-based paradigm began to be applied to more dynamic systems, such as factory automation and robotics.

Previous work in robotics has produced many coordinators aimed at task management. Examples include Colbert[6] and TDL[11], both designed for specifying and controlling reactive components of a hybrid architecture. Another is the Reactive Model-based Programming Language[14], which is a synchronous system for coordination of embedded systems using a deductive reasoning engine.

3 OpenRTM-aist

OpenRTM-aist is a component-based architecture for intelligent systems, known in Japan as “Robot Technology,” or “RT.” OpenRTM-aist implements the Robot Technology Component (RTC) specification [10], which defines a common introspection interface. Introspection is an important part of OpenRTM-aist, providing the basis upon which tools build to interact with and manage systems using the architecture.

The central concept in OpenRTM-aist is the RT Component, or RTC. Each RTC has an internal state machine with known states. The component’s health can be both monitored through the introspection interface. Components begin executing when they are activated, entering the *active* state. Execution is terminated by deactivating the component, moving it to the *inactive* state. The state machine can also enter the *error* state.

OpenRTM-aist supports data-flow- and request-based communications, with interaction between components occurring at “ports.” The component network is formed by making connections between the ports of the components.

OpenRTM-aist can use manager daemons to load and manage components dynamically on remote nodes. This point is important for the dynamic management of component networks in which components may come and go as needed.

OpenRTM-aist uses CORBA [5] to implement the introspection interface. Components must register on a known name server (there can be more than one known name server in use). Typically, components are organised on the name server using naming contexts below the root context in order to provide hierarchical categorisation.

3.1 Managing RT Systems

As mentioned above, introspection interfaces are an important feature of RT Systems. It is through these interfaces that components are monitored and managed, connections are created and removed, and the RT System as a whole is managed.

This introspection interface provides completely dynamic access to an RT System at run-time. Changes, both into the interface and out of it, are reflected immediately by the receiving end. This dynamic control over the component network is what allows for external run-time coordination of the network.

OpenRTM-aist also features in its implementation a large number of internally-accessible callback functions. These functions are used by the components themselves to react to changes in the network around them and in the component itself.

For example, when a connection is made between components, a callback is triggered in each component involved. The components can use this callback for any purpose the component designer deems suitable. One example is having the component respond to a new connection on an output port by activating a part of its behaviour that produces data. Prior to a connection existing, that part of the component can remain dormant, reducing resource consumption. Another example is a callback triggered when data is received on a port, which a component can use to perform out-of-band processing on data rather than processing it within its behaviour loop.

These callbacks provide a considerably more reactive interface than the externally-accessible introspection interface. We can use them to give a component some intelligence about its behaviour based on its abstract view of the state of the network around it. Unfortunately, they cannot be accessed externally to a component. We are unable to use them for external coordination of the component network. We have therefore created an external coordination framework that mimics these callbacks.

4 Coordinating RT Components

A coordination framework has been implemented for OpenRTM-aist. It gives developers the necessary functionality to support external, fine-grained reactive and pro-active coordination of an RT System.

The coordination framework has been integrated into the "rtctree-erl" library, implemented in Erlang [2]. It consists of a set of user-definable callbacks. The following section describes rtctree-erl, while the callbacks that allow coordination are described in Section 4.2. An example of using the framework is given in Section 4.3.

4.1 rtctree-erl

This library provides the low-level framework into which the coordination callbacks are integrated. Its purpose is to provide an API for the OpenRTM-aist introspection interfaces in Erlang.

rtctree-erl uses a tree structure to represent the component network and related OpenRTM-aist objects, such as managers. An example of this structure is shown in Figure 1. Below the root node are all the known name servers. Name servers may have multiple naming contexts to which objects can register for organisational purposes. Developers can use naming contexts to impose an organisation on the running components.

Within the rtctree-erl library, each node in the tree is represented by a separate Erlang process. We take advantage of Erlang's light-weight processes to allow concurrent access to any part of the tree (a single Erlang VM can support thousands of processes on an average desktop computer). Each node acts independently of the others, communicating using Erlang's message-passing communication mechanism (Erlang processes cannot share memory, in order to provide robustness).

The processes communicate with OpenRTM-aist distributed objects using the OpenRTM-aist CORBA introspection interface. The library uses Orber [8], the Erlang CORBA implementation, to make the remote procedure calls.

Because Erlang is derived from functional programming languages, the only way to maintain data is by passing it from function to function. Each node in the RTC tree follows this pattern, maintaining a structure specific to its node type. In keeping with Erlang conventions, clients of the library do not directly use this data structure. Rather, they make remote calls to the process containing the data structure, sending a message corresponding to the operation to be performed. The calling process typically waits for the result, but it is possible to continue execution and check for a result at a later time. This is a key feature of Erlang's method passing that increases the flexibility of its RPC. The rtctree-erl library provides an API for each node type that hides the RPC code and provides what appears to be a traditional function-based API to clients of the library.

RT Components only exist as leaves of the tree. The node representing a component contains a cache of the component's information, retrieved through the introspection interfaces of OpenRTM-aist. This information can be updated at any time by requesting an update of all or a part of the cache, which will cause the node process to make a new remote procedure call and get the latest value. It is not updated automatically by default (although see the next section for an exception) as it does not change regularly; rtctree-erl assumes that the developer knows best how often to update the information.

A component's node may also contain child processes representing its ports, execution contexts and configuration sets. These are represented using their own processes to allow for concurrent access to ports as well as components. Like the component nodes, port and execution context nodes cache information retrieved through the introspection interfaces.

Using a tree structure allows additional tree-structure operations to be performed, such as iterating over every node in the tree to perform an operation on every node matching certain criteria.

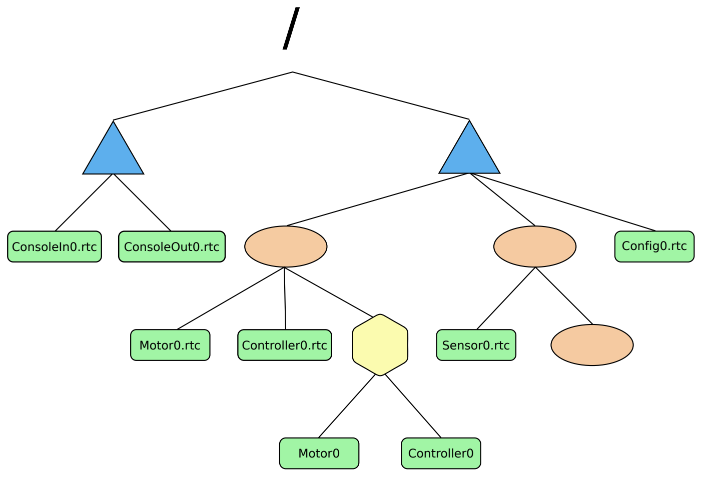

Fig. 1. An example of the tree structure used by the *rtctree-erl* library, dubbed the *RTC Tree*. The root node is at the top of the tree. The blue triangles are naming service nodes. Red ovals are naming contexts below a root context. Green boxes are RT Components. The yellow hexagon is a manager. Below it are references to the two RT Components it is managing.

4.2 External callbacks

The rtctree-erl library provides easy access to the introspection interfaces of OpenRTM-aist, as though accessing an API on the objects directly. However, writing coordination code with it is not so simple because the introspection interfaces do not provide a reactive interface. A coordinator would need to manually check important conditions.

On the other hand, OpenRTM-aist provides a large set of internal callbacks for components to used, a small sample of which is shown in Table 1. Having this fine-grained information available externally would make implementing powerful coordination systems easier.

We have implemented a set of external callbacks that use the introspection interfaces to mimic the internal OpenRTM-aist callbacks. A sample of these external callbacks is given in Table 1. Not all of the internal callbacks are implemented externally; in some cases the information is not available externally in any way. For example, information relating to data being received cannot be detected externally.

The framework acts as an automated intermediary between coordinators and the rtctree-erl library's introspection APIs. The callbacks are triggered automatically when appropriate. Developers can write their coordinators to be reactive to the changing state of the robot software system.

The callbacks are implemented in the component node, port, execution context and configuration set processes of the rtctree-erl library. The developer must

Table 1. A sample of the callbacks available internally in OpenRTM-aist, and those provided by rtctree-erl.

Internal callback	External callback (module:type)	Purpose
onActivated, onDeactivated, onError	component:state	Notify of changes in a component's state.
onConnect	port:connect	Notify of a new connection on a port.
onDisconnect	port:disconnect	Notify of the removal of a connection from a port.
OnUpdateParamCallback	configuration: update_param	Notify of a configuration parameter being updated.

implement callback functions and then register them in the process of interest, indicating the piece of state information the callback is for. For example, a developer can write a function to handle a component entering the *error* state, then register it with a component process as a callback for state changes.

The component and node processes monitor the state information relevant to the callbacks that are currently active. When this state information changes, the registered callback functions are called. (We term the process that triggered the callback the "source process.") Callbacks are passed:

– The relevant piece of new state information,
– What the value was before the change,
– The process identifier (PID) of the component or port process that triggered the callback, and
– Any extra data relevant to the specific callback.

Callbacks are executed in a separate process from the source process. One reason for this is to prevent callbacks from blocking the source process from triggering other callbacks by taking too long to execute. However, the primary reason is to prevent deadlocks: if the source process is executing the callback, the callback cannot use the RPC-based API to manipulate the OpenRTM-aist object that triggered the callback. Instead, the source spawns a worker process to handle each callback it triggers. We take advantage of Erlang's light-weight processes and rapid process spawning here.

The next section uses an example to illustrate the use of the framework to coordinate an event in a robot software system. Through the example, it gives further details on how the framework functions.

4.3 Example: re-configuring a localisation system

In this example, we are coordinating the simple RT System shown in Figure 2, a localisation system utilising a laser, gyro and odometry. Two localisation components are available, with the laser-based component in use when the robot starts. Only one is active at a time.

We monitor the system for faults in the laser. When a fault occurs, we need to switch from the laser-based localisation component to one based solely on the gyro and odometry. We must shut down the laser-based localisation component, create the odometry/gyro localisation component, form new connections, and activate the new component. The code to do this is shown in Listing 1.1.

First the coordination program starts the framework (Line 6). Libraries in Erlang are implemented as "applications," providing an API for accessing their functionality and operating concurrently to client programs. The coordination program then registers the callbacks it is interested in (Lines 8 to 9). The coordinator retrieves the node of interest from the RTC Tree using its path, then registers its callback function with this node. It provides the callback function object, the type of callback it is registering (a state-change callback, in this case), and the rate at which to update the state from the remote object.

The coordinator is now fully-activated. The starting process is free to exit at this point; the callback functions will be executed in their own processes.

The callback function is shown on lines 12 to 30. Note the rule-like pattern matching ensuring that this callback only executes when the new state is the error state, and the triggering component is the component of interest. It uses the rtctree-erl library APIs to handle the event, replacing the failed laser node and the now-unusable localisation component with a new localisation component. Note particularly line 20, where a new component is instantiated dynamically.

Figure 3 provides a sequence diagram showing the flow of control amongst the various processes involved in this example. Figure 2 shows the component network over time during the example.

5 Discussion

The framework presented is designed to be flexible. The goal is to allow robot developers to implement coordinators that meet their needs. The callback system allows for continuous monitoring of any aspect of an RT System. Its speed is limited by the response of the CORBA introspection interfaces.

This callback system provides flexibility to coordinator implementers. For example, it is easy to register a callback that is called whenever the state of a component changes. Sample code showing this is given in Listing 1.1, which shows waiting for a component to go into the error state. Callbacks can also be registered that are called when a new connection is created on a port, when a manager creates a new component, and so on. Combining this framework with direct usage of the rtctree-erl library provides control over the network being monitored, such as re-arranging the component network when an error occurs in a component.

Because the rtctree-erl library, the coordination framework and coordinators are all implemented in the inherently-thread-safe Erlang programming language, there are no concerns with using such a large number of processes. However, this

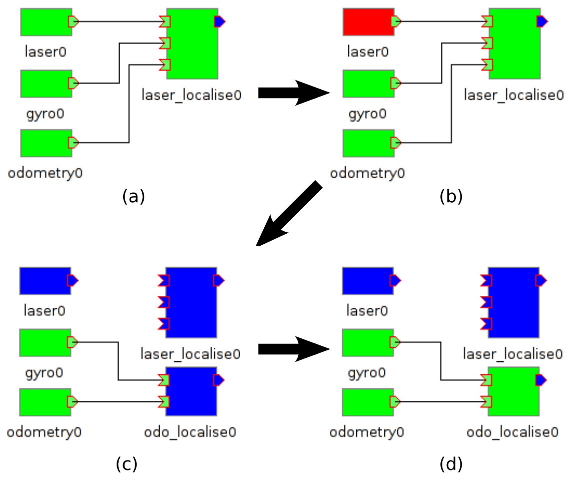

Fig. 2. The changes in the component network of the example. (a) The components operating as normal. (b) An error occurs in the laser component. (c) The connections to the failed component are removed, a new localisation component is instantiated and connected. (d) The new localisation component is activated.

is a framework, and so it is up to the programmer to ensure their coordinator design will not lead to, for example, conflicting callbacks. While Erlang supports soft real-time, it is up to the programmer to ensure their callbacks will not execute for too long. The framework does not check if a callback is currently running if the next tick causes it to be triggered again. Hard real-time is not possible using Erlang, which may limit the applications of this system.

This coordination framework is more dynamic than previous methods. We no longer need to have every component we may potentially use executing and a part of the component network all the time. Instead, we can start and stop components as necessary in response to changes in robot state. This preserves system resources and simplifies the component network.

This framework compares favourably to the languages mentioned in Section 2. Its primary benefit is that it uses an existing language rather than inventing a new one, simplifying the framework development and providing it with the power of a complete language. Despite this, it still has a rule-like syntax similar to custom-designed languages. In addition, both Colbert and TDL are designed for *specifying* tasks as well as coordinating them. This framework is purely for coordination of existing components. The difference with RMPL is that this framework is not synchronous. While this limits its use at lower levels for fine control, it is suitable for higher-level planning tasks.

The most important contribution of this work is showing that the Erlang syntax is a benefit when writing coordinators. Erlang's function declarations use a pattern-matching system. This makes them look like rules. We benefit from this in our coordinator framework. As the example shows, we can write

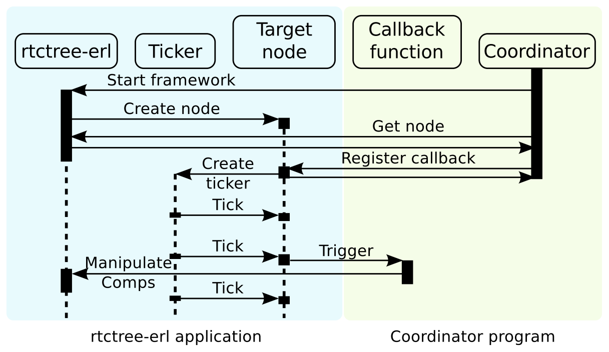

Fig. 3. The processes involved in using a single callback coordinator. Note how the processes appear and disappear as needed; when no events need management, the coordinator itself requires no resources and rtctree-erl only requires enough to regularly update the relevant status.

the function declaration of the callback functions as a set of rules. The callback in the example is a rule for a component moving to the error state. We gain this syntax benefit without needing to create or modify a language. Erlang has relatively wide support in industry, and has been in use in industrial applications, particularly telephony, for nearly 25 years [2].

6 Conclusions

Component-based software architectures are becoming popular in robotics. They lead to more flexible robot software structures. Coordination of the component networks to adapt to changing situations and changes in robot state is still a topic of research. Many popular frameworks do not provide an externally-accessible interface for the management of components and the connections between them, making it difficult to create an external coordinator tool.

The OpenRTM-aist architecture features fully-dynamic connections and components. It does not yet have a coordination tool, and its internal state-monitoring callbacks are also more capable than its external introspection interfaces.

We have taken advantage of the introspection interfaces to create a coordination framework that provides the same fine-grained monitoring and control as the internal monitoring callbacks. The framework is implemented on top of the rtctree-erl library. It is implemented in Erlang, a concurrent programming language. It uses features of Erlang to gain concurrency support and robustness. The framework shows that Erlang has much to offer robotics in concurrency support, robustness and error-handling, and in syntax.

Acknowledgements

This research was supported by a JSPS *kakenhi* grant 21700230.

References

1. Ando, N., Suehiro, T., Kotoku, T.: A software platform for component based rt-system development: Openrtm-aist. In: SIMPAR '08: Proceedings of the 1st International Conference on Simulation, Modeling, and Programming for Autonomous Robots. pp. 87–98. Springer-Verlag, Berlin, Heidelberg (2008)
2. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
3. Brooks, A., Kaupp, T., Makarenko, A., Williams, S., Oreback, A.: Towards component-based robotics. In: Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on. pp. 163–168 (Aug 2005)
4. Bruyninckx, H.: Open robot control software: the orocos project. In: Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on. vol. 3, pp. 2523 – 2528 vol.3 (2001)
5. Henning, M., Vinoski, S.: Advanced CORBA Programming with C++. Addison-Wesley Professional (1999)
6. Konolige, K.: COLBERT: A language for reactive control in sapphira. Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science) 1303, 31–52 (1997)
7. Nesnas, I.A.D., Wright, A., Bajracharya, M., Simmons, R., Estlin, T.: Claraty and challenges of developing interoperable robotic software. In: Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on. vol. 3, pp. 2428–2435. Las Vegas, Nevada (October 2003)
8. Erlang – Orber Reference Manual. `http://www.erlang.org/doc/apps/orber/index.html` (2010)
9. ROS Wiki. `http://www.ros.org` (2010)
10. The Robotic Technology Component Specification - Final Adopted Specification. `http://www.omg.org/technology/documents/spec_catalog.htm` (2010)
11. Simmons, R., Apfelbaum, D.: A task description language for robot control. In: Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on. vol. 3, pp. 1931–1937 (1998)
12. Song, B., Jung, S., Jang, C., Kim, S.: An Introduction to Robot Component Model for OPRoS (Open Platform for Robotic Services). In: Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots 2008, Workshop Proceedings of. pp. 592–603 (Nov 2008)
13. Szyperski, C., Gruntz, D., Murer, S.: Component Software – Beyond Object-Oriented Programming. Addison-Wesley and ACM Press, 2nd edn. (2002)
14. Williams, B.C., Ingham, M.D., Chung, S.H., Elliott, P.H.: Model-based programming of intelligent embedded systems and robotic space explorers. Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software 91(3), 212–237 (January 2003)

Listing 1.1. An example callback using the framework. The callback setup is just three lines of code; adding additional callbacks would require one to two lines per callback at most. The callback itself uses the rtctree-erl library's API to manipulate the component network in response to the laser node suffering a failure.

```
-module(example).
-export([run/0, cb/4]).
-include("rtctree-erl/include/nodes.hrl").

run() ->
    ok = rtctree:start(),
    ok = rtctree:add_servers(["localhost"]),
    {ok, C} = rtctree:get_node_by_path(["/", "localhost",
        "laser0.rtc"]),
    component:add_cb(C, fun state_change/4, state, 1000).


state_change('ERROR_STATE', _Old, C, _Extra) ->
    % Disconnect and shut down the old localiser and laser
    ok = component:reset(C, 1),
    {ok, Loc} = rtctree:get_node_by_path(["/", "localhost",
        "laser_localise0.rtc"]),
    ok = component:disconnect_all(Loc),
    ok = component:deactivate(Loc, 1),
    % Instantiate the new localiser
    {ok, M} = rtctree:get_node_by_path(["/", "localhost",
        "manager.mgr"]),
    ok = manager:create_comp(M, "odo_localise"),
    % Connect the new localiser
    rtctree:connect_by_path(["/", "localhost",
        "odometry0.rtc", "odo"],
         ["/", "localhost", "odo_localise0.rtc", "odo"]),
    rtctree:connect_by_path(["/", "localhost", "gyro0.rtc",
        "gyro"],
         ["/", "localhost", "odo_localise0.rtc", "gyro"]),
    % Activate the new localiser
    {ok, OdoLoc} = rtctree:get_node_by_path(["/",
        "localhost", "odo_localise0.rtc"]),
    component:activate(OdoLoc, 1);
state_change(_, _, _, _) ->
    ok.
```