2010 IEEE International Conference on Robotics and Automation
Anchorage Convention District
May 3-8, 2010, Anchorage, Alaska, USA

Flexible, adaptable utility components for component-based robot software

Geoffrey Biggs
Intelligent Systems Research Institute
National Institute of Advanced Industrial Science and Technology (AIST)
AIST Tsukuba Central 2, Tsukuba, Ibaraki 305-8568, Japan
geoffrey.biggs@aist.go.jp

Index Terms—**Robot programming systems, component-based architectures**

Abstract—**Component-based software design is a current trend, both in general software practice and in robot software practice. It brings benefits to the field of robot programming. Component interfaces are fixed at design time and form a contract with other components, guaranteeing functionality. Known interfaces are typically important to reusability. However, in certain cases fixed interfaces can limit the reusability of components. Utility components provide general functionality that is reused a large number of times both within a single software system and between systems. They need to be adapted to the interfaces for each specific use case. This paper presents a set of utility components that can adapt their interfaces to the user's needs without any code changes. Dynamic programming language techniques are used to provide the adaptability. The components are a great benefit to the reusability of common utility components, removing a common cause of reinvention.**

I. 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 in much the same way as hardware components of electrical circuits are combined to create a complete hardware system [1].

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), and isolation testing, where each component can be tested isolated from the others.

These benefits also apply strongly to the design, implementation, maintenance and reuse of robot software. For example, the componentisation of hardware drivers and algorithms allows robot systems to be built from pre-existing, ideally off-the-shelf, software components. As a result, component-based software is a major trend in robotics, particularly service robotics. Recent examples include OpenRTM-aist [2], ORCA [3], ROS [4] and OPRoS [5].

A robotic software system, as with any software system, will accumulate some common, fundamental functionality. For example, data structures representing common data types such as 3D vectors. In a component-based system, some of the common functionality may be entire components that perform well-known, often-used functions on data streams between the other components of the system. We term these *utility components*. One of the difficulties in using utility components is that they are more than just code. They also feature a component interface that dictates how they interact with other components.

A component's interface is a contract it makes with other components. The interface tells the world outside the component what services it can provide and what it requires to perform those services. In a very real sense, the interface dictates the possible functionality of the component, significantly affecting its reusability [6].

Because components are designed for a specific purpose, their interfaces are typically fixed. While a well-designed, fixed interface can promote reusability, by their very nature, fixed interfaces tend to be inflexible interfaces. They guarantee a certain level of functionality, but cannot easily adapt to varying usage scenarios unforeseen by the original design. Interfaces must therefore be designed carefully to provide the necessary flexibility for their potential use cases. Careful design choices are required at interface design time to maximise reusability.

This is where the problem lies with regards to utility components. Rather than being the promoter of reusability, fixed interfaces are instead its antithesis. No matter how carefully the interface of a general-purpose utility component is designed, it will not be flexible enough, or it may become so large as to be unusable. As a result, utility components tend to be reimplemented over and over again, each time with a different interface matching the new use case. Utility components are likely to be implemented for a specific system, designed to meet the interfaces used in that system, or possibly even just the interfaces in the part of the system they are to be used in. Should a developer wish to reuse a utility component elsewhere, its interface must be altered to match the new use case. Even if internal functionality is copied, coding is still required to match it to the new interface.

This paper presents a set of utility components developed over a period of time and used in real robot systems. They target common needs in a robotic system built around a component-based architecture. The architecture used in this case is OpenRTM-aist [2], but the components should be applicable to any component-based architecture. The key point of these components is that they all provide flexible interfaces that can be configured at run-time to meet the user's specific needs without any code changes. This saves considerable reimplementation effort and vastly increases the reusability of these components.

The paper begins by describing the components and the common functionality each one provides in Section II. How the flexibility is implemented is described in Section III. A discussion of these components and their flexible nature is given in Section IV. General flexibility in robotics is discussed in Section V. Conclusions and recommendations are given in Section VI.

II. Flexible utility components

In 1994, Gamma *et al* published their now-famous book, "Design Patterns" [7]. This book laid out in detail 23 different common structures of software design that can be found in varying software projects and introduced the concept of software design patterns. In a similar vein, software projects will often use a collection of utility functions and objects. However, rather than being the basis of a software object's design, these are complete pieces of code.

Component-based software is not excluded from this. It is likely to regularly reuse entire utility components. Such components may perform tasks such as selecting between the outputs of two similar components, logging the data streams between components, and filtering component outputs. These general purpose components are frequently required throughout a component-based system, and so need to be adaptable to a wide variety of use cases. The required adaptability cannot be achieved using fixed interfaces. This leads to significant duplication of effort and wasted time reimplementing utilities for specific needs.

Instead, interfaces that are configurable by the user to the specific use case are necessary. Ideally, the user should not need to recompile the component to create the interface they require.

This section describes a collection of utility components built over time based on experience creating robot systems using the OpenRTM-aist software architecture [2]. These components are created using the Python version of OpenRTM-aist, and written purely in Python. The key point of these components is that they all feature flexible, run-time configurable interfaces. The user specifies the interface they require when starting a component. No further work nor time is necessary. This is a major benefit to the reusability of these components; they are written once and used indefinitely. The following sections describe the functionality of each of these components, with particular focus on their flexibility.

A. FlexiDump

This is a simple console-dumping component. It was written in response to frustration at the limitations of the console-dumping component provided with OpenRTM-aist. This component, called `SequenceIn`, is designed to function with any numeric data type provided by OpenRTM-aist. To

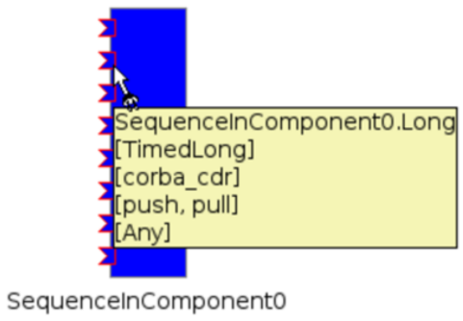
SequenceInComponent0.Long
[TimedLong]
[corba_cdr]
[push, pull]
[Any]

Fig. 1: The SequenceIn comp, as seen in RTSystemEditor. Each port represents a different data type.

provide this functionality, it has one port for each numeric type, including sequence data types (see Figure 1). It is hard-coded to have this interface. Should the user wish to dump more than one stream of the same data type, two instances of the component are required. The component doesn't even support all data types available in OpenRTM-aist.

Rather than this limited interface, the author decided that a component that could provide any interface required by the user would give much greater usability and reusability. It is more ideal that the ports simply don't exist when they are not required. The FlexiDump component was the result, and this idea was extended to other component types. Any combination of input ports using any combination of data types can be dumped to standard out. The output is suitably prefixed to increase readability.

B. FlexiFilter

The FlexiFilter component acts as a filter or an adaptor between two or more component ports, altering data types and data shapes as necessary to allow those components to work together. The user specifies one or more input and output ports, and a mapping between them. This mapping is important, as it tells the component which output ports to send data received at input ports to. For example, a `TimedFloat` port could be directed to a `TimedLong` port, causing its data to be cast from a float to an integer and so allow the data output by one component as floating-point values to be used by another that expects integers.

The component is more powerful than just casting data types. It can also map the individual values of sequence ports to other ports, and even to individual values of output sequence ports. Conceptually, this is like mapping individual pins of one port to individual pins of another. This is illustrated in Figure 2. When only some pins of an output port are altered, the remainder can either be left at their previous values or reset to a specified value, for example, zero.

For example, a FlexiFilter instance can be configured with three input ports, each a `TimedFloat` representing x, y and z values. These can be mapped to a single output port of type `TimedFloatSeq`, with each going to a different index of that port. The result is a filter component that can convert the outputs of three individual controllers, one for each axis

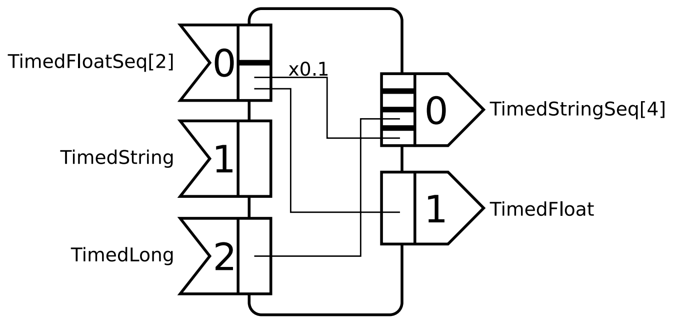

Fig. 2: A possible filter layout capable with the FlexiFilter component, mapping desired input values to their new positions in the output ports.

of a robot, to a single vector containing all three that can be used by a robot interface component expecting its data in that format. This is achieved without any reimplementation of the filtering component.

The component is also capable of modifying the data as it passes through. Data from a channel can be offset by a given value, or it can be scaled. The earlier example of a `TimedFloat` being cast to a `TimedLong` can be extended to include multiplying the data by 1000. This gives a FlexiFilter component that converts data from metres in floating point to millimetres in fixed point.

The user passes in the port configurations and port mapping on the command line. For example, the component shown in Figure 2 would be configured as follows:

```
./flexifilter.py -i TimedFloatSeq:2 -i
    TimedString -i TimedLong -o
    TimedStringSeq:4 -o TimedFloat -m
    "2>0:2,0:1>0.1>0:3,0:1>1"
```

Once the component is running, it waits for data to be received. As data is received on a port, that data is altered based on the port map and sent to the indicated output ports.

C. FlexiLogger

The FlexiLogger component performs a function commonly used, not just in component-based robotic systems, but in any robotic system: logging data. It is also capable of replaying log files, optionally using recorded timing to simulate the speed of the original system.

Logging is accomplished using either a flat text file or Python's `pickle` module. Pickled is the preferred mode as it is a binary dump of the object data. This preserves accuracy in floating-point data.

The component is switched between record and replay mode using a command line option. The default is to record.

The replay time replication is limited in its accuracy by the lack of real-time support in the Python interpreter. Replayed data types that feature a time stamp, such as `TimedLong`, can optionally have their time stamp adjusted for the current time, making it appear as if the data originates in the present rather than when it was recorded.

The flexible interface makes this component very versatile. It can log any combination of ports and OpenRTM-aist data types. Rather than creating one component for each data type and logging data streams separately, a user can create a single component representing all the data they wish to log in a synchronised fashion by specifying the appropriate port types. The data will be recorded, stored and played back together, preserving synchronisation between streams.

D. FlexiSelect

This component performs a very useful function in any component-based system: selecting between two or more inputs based on another input. For example, it could be used to select between two controller components based on the output of a third component that determines the robot state.

The user specifies the normal input ports, as with the other flexible components, but with the limitation that these ports must all be of the same data type. A single output port is created automatically that matches this data type. A separate integer input port is created for the selection port. The value received is used as the index of the input port to switch to. In the simplest case, with two input ports, sending a one or zero simulates binary high and low.

E. FlexiAdd

The FlexiAdd component can be thought of as the motor schemas [8] component. It takes the input ports defined by the user, all of the same data type, and adds them together to produce a single output.

The user can specify both scalar and vector data types. They will be added together correctly. In the case of vector types, addition is member-by-member, producing an output of the same length as the inputs. This means that all input ports must receive data of the same length. For example, this component can be used to add a controller moving forward and a controller moving away from obstacles together, producing a single input for the robot component.

F. FlexiConst

FlexiConst generates one or more constant values matching a data type specified by the user. It can generate the values at specified intervals, or produce a continuous stream. It is capable of outputting scalars and sequences.

III. Implementing flexible interfaces

All the flexible components described in the previous section share a common feature: their input and/or output ports are not predefined at compile-time. Instead, they are configured at run-time based on the user's needs. When starting the component, the user provides command line options that describe the input and output ports they require. The component does the rest, creating a component that meets the other components of software system the user is implementing.

This flexibility in the interface, and therefore in the components, is possible because of Python's dynamic programming language capabilities.

The start-up process of a flexible component best describes how the flexibility is accomplished. Briefly, the steps are as follows:

1) User specifies via command line options what input and output ports they require, in terms of data type.
2) Using Python's reflection facilities, determine what data types are available for use on data ports and ensure those required by the user are available.
3) Create a component without any flexible ports (other ports are created as normal) and pass it the list of ports it should have.
4) Component object's `onStartup()` method adds the ports using Python's ability to dynamically add member functions and variables to a class at run-time.

As these steps show, the flexibility depends on two key features of Python: reflection allowing introspection of the available name spaces and names, and dynamic programming language features allowing run-time type modification of classes. Implementation in a language without these features, such as C++, would be less straight-forward and may depend on some hard-coding of type names.

A. Finding available port types

Python's reflection facilities, in particular its `inspect` module [9], provide an array of functionality for inspecting a live Python program. Of particular interest here is the module's functions to list the available members of a class or name space. OpenRTM-aist provides a set of standard data types, and user-specified IDL can also be used for the data type of a data port. The flexible components do a search using `inspect` to find the available types:

```
def FindPortType (typeName):
  types = [member for member in
      inspect.getmembers (RTC,
      inspect.isclass) if member[0] ==
      typeName]
  if len (types) == 0:
    return None  # Type not found
  elif len (types) != 1:
    return None  # Ambiguous type name
  return types[0][1]
```

The key line here is line 2. It builds a list of classes in the RTC name space that match the given type name.

This function returns the name of the data type and its value, which in Python is essentially a constructor that can create objects of that data type. Each data type factory is added to a list of input or output ports to be created during component construction.

We note that the components, as currently implemented, cannot change their interfaces after construction. The reason for this is not related to the component implementation itself; the facilities described here are quite capable of doing this. However, the system editor tool provided with OpenRTM-aist, RTSystemEditor, only reads a component's ports when the component registers with the name server. After that point any changes in the ports are not reflected in the system editor's graphical system diagram and so are not usable. Moreover, removing ports would lead to serious errors in the use of the system editor. Should a user be programmatically connecting ports, this would not be an issue.

B. Creating ports

Once the creator objects for each data type have been located, the empty component can be instructed to add its ports. Each flexible component has a step in its `onStartup` method that adds input and/or output ports according to the list created earlier. For example, the code shown below is from the FlexiSelect component's `onStartup` method.

```
self.__inPorts = []
self.__inPortBuffers = []

newPort = ports[0]
for ii in range(newPort[2]):
  newInPortData = newPort[1](RTC.Time (0, 0),
      [])
  newInPort = OpenRTM.InPort('input%d' % ii,
     newInPortData, OpenRTM.RingBuffer(8))
  self.registerInPort('input%d' % ii,
     newInPort)
  self.__inPorts.append([newInPortData,
     newInPort])

self.__outPortData = newPort[1](RTC.Time (0,
    0), [])
self.__outPort = OpenRTM.OutPort('output',
    self.__outPortData, OpenRTM.RingBuffer(8))
self.registerOutPort('output', self.__outPort)
```

Adding a new member variable to a class instance in Python is simple. This is because Python classes are implemented as dynamic name spaces referenced by the `self` variable. Anything added to the name space becomes a member of the class instance and can be accessed both within that class and by external objects.

Once all ports have been created, the `onStartup` method exits, and the component is available for use with its customised interface. The process is very quick and requires no user interaction beyond specifying the interface layout on the command line.

IV. Flexible utility component discussion

Flexible interfaces are, in some cases, very important for increasing both usability and reusability. The components presented here are adaptable to the user's needs. Rather than reimplementing components, even if just wrappers around a provided core, to suit a specific need and specific interface, the user only has to provide options when starting the components.

The collection of flexible components, built up over a period of time through observation of repeating implementations, illustrate some of the most common functionality required in a component based system. A component-based architecture should endeavour to include components representing this set of functionality. This would save considerable reimplementation by architecture users.

While these interfaces are flexible, they are not a hindrance to automatic system construction. They can be treated as any other component with a known interface, as their interfaces remain static after component initialisation and can be known from the start-up configuration.

Unfortunately, because Python does not support real-time, the components also do not support real-time. However, we have not had any difficulty using them in several control schemes.

A. Alternative implementation methods

The components implemented here depend on the availability of dynamic language features that allow introspection of a run-time system and modification of class instances at run-time. Without such features, these components would be more difficult to implement, but not impossible.

For example, C++ templates and Java generics can be used to create components customisable by data type. However, these generic programming techniques require the component to be compiled before use. The technique used in this paper does not, making reconfiguration much faster and simpler.

A faster approach in a language that does not support reflection is the use of standard object-oriented techniques and dynamically allocated types. For example, in C++ with OpenRTM-aist, the port objects could be allocated on the heap at run time. This would provide much of the flexibility of the Python implementation, but at the expense of much longer, more repetitive code. It would also be limited to those types available at compile time, unlike reflection.

B. Comparison with fixed interfaces

Examples of fixed-interface utility components can probably be found in any research lab using a component-based architecture. There is little to describe; these components simply present an interface that represents the data stream they manipulate.

The flexible components offer an enormous advantage over what they would look like were they implemented with fixed interfaces. In the case of FlexiDump, we can directly compare with an existing component, the `SequenceIn` component. The interface style of `SequenceIn` is to present one port for each data type in the system. This is clearly not an ideal situation. Such an interface would need to have a port for every data type in the system. Should a new basic data type be added, any component with this style of interface would need to be updated. Even worse is the possibility of a user-defined data type, a feature provided by OpenRTM-aist, being desired.

The advantage offered by the flexible components is especially the case for the more complex utility components, such as FlexiFilter, FlexiLogger and FlexiSelect. While a fixed-interface console dumper can work, the ability to configure a logging component to handle any configuration of ports allows all data to be logged to a single log file, aiding both in reviewing that data and in keeping that data synchronised during playback. In a more extreme case, the FlexiFilter concept is simply not possible with a fixed interface. A new component would need to be implemented for every new filter configuration.

C. Application to other components

The technique presented here is not limited to utility components. It may find uses in other components. For example, a component implementing multiple interfaces, of which only a subset are active at any one time based on use case, could use this technique to prevent the other interfaces from existing when not needed. This would bring benefits in less system designer confusion (due to less ports cluttering the interface), ability to reuse port names, and reduced clutter in graphical system editor tools such as OpenRTM-aist's RTSystemEditor and ROS's node viewer.

The reduction in space used in a graphical system is, we concede, a minor benefit. However, it should not be overlooked. The usability of a component within a graphical system editor should be considered. A large component interface leads to a large graphical representation of components using that interface, which can lead to complicated system diagrams and node graphs.

The same principle could be applied to a component implementing a complex interface of which only a well-defined subset of ports are available for use at any one time. Complex interfaces, like complex APIs, can reduce usability and increase maintenance costs through programmer error.

In general cases, however, it is the author's opinion that standardised interfaces are more important than flexible interfaces that may be unknown. Fixed interfaces only reduce reusability in a limited subset of use cases.

V. Other routes to flexibility

The use of dynamic language facilities described in section III is not necessarily the only way to achieve flexibility in component design. There are other possibilities. These include property bags and subdividing interfaces.

In addition, the dynamic programming language features used for the flexible components are not limited to user-specified configurations. They could be used for components that adapt their interfaces automatically at run-time based on any number of criteria, from changes in internal robot state to the direction of an outside orchestration system. They would facilitate reconfigurability of entire robot systems.

Component wrappers, which wrap one component in another to alter its interface, are another option for flexibility. There is likely to be considerable work involved in using this option each time to create the new interface, though.

The CORBA specification provides a dynamic interface facility [10]. This is a special generic interface used to make requests on distributed objects where the object type is specified at run-time. This is an implementation method, and the facilities presented here would still need to be implemented if this approach were used.

A. Property bags

A property bag is a technique for providing additional "properties" of an object beyond what its main interface provides, which is typically based on some standard or inherited from a parent object in some way. Prudent use of property bags can allow the common aspects of a general interface to be provided by that interface, while the parts that vary considerably between more specific component types within the same category are provided by a property bag on each component.

Property bags are not a new concept, but they are relatively under-utilised in robotics. Previous work has been done implementing a property bag system for a robot programming architecture [11].

The Player architecture recently added a feature resembling property bags [12]. In this case, the bags are split open and each property is treated individually. The addition of properties is still not widely used by Player's drivers, but in cases where it is, it provides additional driver-specific run-time configurability beyond what the standard Player interfaces allow. For example, the scanning speed of a laser scanner can be altered via the standard laser interface, but the baud rate of the connection to the scanner cannot. Creating a property for this baud rate allows it to be changed.

A disadvantage of properties and property bags is that they may not tend towards standardised properties. Instead, the property names may diverge considerably between components even for identical properties. This has been seen to some extent in Player. However, they are a powerful means of providing additional flexibility to components beyond what a known interface may allow.

B. Subdividing interfaces

Perhaps the most obvious method for creating flexible interfaces is to split those interfaces up. Creating many small interfaces that can be combined together as needed gives great flexibility. Interfaces could be defined at as fine-grained a level as the individual ports. This would still ensure that each port conforms to a known contract, while at the same time allowing which ports are provided to be selected as needed by the component designer or the component user (given a suitable implementation of the component - see below). This is the approach taken by, for example, ROS, in which interfaces are not monolithic, and only individual message types, each published on its own channel, are defined [4].

The downside of the coming and going of ports is the same in any case of flexible interfaces: knowing at run-time which ports are available for an automated orchestration method to connect to them. This is considered an implementation issue that can be solved by, for example, a meta-interface query for which ports are provided.

It is worth noting that subdivided interfaces could be implemented using the same techniques used to create the flexible utility components described in section III. This would give a combination of flexibility and standardisation, which could be very powerful, especially with a suitable introspection system for finding which ports are available programmatically.

VI. Conclusions

This paper has presented a set of flexible utility components. These perform many functions commonly found in component-based software systems, and, in particular, in component-based robotic systems.

The components are implemented for the OpenRTM-aist architecture, using the Python edition. They make heavy use of Python's dynamic programming language and reflection facilities to allow for run-time configuration of their external interfaces. This ability to alter their external interfaces is the key to their flexibility. Without it, they would not be able to adapt to different use cases and meet the widely-varying needs placed on utility components by users.

The usefulness of this approach to flexible component interfaces has been discussed. How useful this flexibility may be in other types of components has also been considered. It is the author's view that, in general, known fixed interfaces are likely to be a greater benefit to reusability than flexible interfaces, but in certain cases flexible interfaces bring the greater benefit. Other ways in which components and their interfaces can be more flexible have been discussed, including sub-divided interfaces and property bags.

In suitable situations, flexible and configurable component interfaces greatly extend the power of component-based software design. Flexibility of an interface should always be a consideration during the interface design process.

References

[1] C. S. with Dominik Gruntz and S. Murer, *Component Software – Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley and ACM Press, 2002.

[2] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "RT-middleware: distributed component middleware for RT (robot technology)," in *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, August 2005, pp. 3933–3938.

[3] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Towards component-based robotics," in *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, Aug. 2005, pp. 163–168.

[4] (2009) ROS.org. [Online]. Available: http://www.ros.org

[5] B. Song, S. Jung, C. Jang, and S. Kim, "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*, Nov. 2008, pp. 592–603.

[6] G. Broten, D. Mackay, S. Monckton, and J. Collier, "The robotics experience," *Robotics & Automation Magazine, IEEE*, vol. 16, no. 1, pp. 46–54, March 2009.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison–Wesley, 1995.

[8] R. C. Arkin, *Behavior-based robotics*. MIT Press, 1998.

[9] (2009) 28.12. inspect – Inspect live objects – Python v2.6.2 documentation. [Online]. Available: http://docs.python.org/library/inspect.html

[10] M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*. Addison-Wesley Professional, 1999, ch. 2.

[11] Y. hsin (Oscar) Kuo and B. MacDonald, "A distributed real-time software framework for robotic applications," in *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'05)*, Barcelona, 18–22 April 2005, pp. 1976–81.

[12] (2007) SourceForge.net: The Player Project:. [Online]. Available: http://sourceforge.net/mailarchive/message.php?msg_id=46346FE0.5010801%40auckland.ac.nz