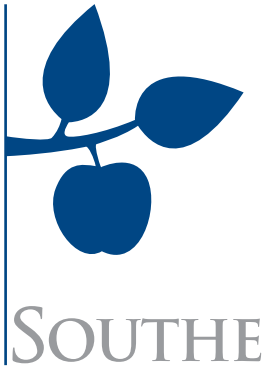
Bachelor



University of Southern Denmark

EasyInsert: Extended user interface and simplification of the interactions with RobWork

Anders Ellinge
aelli14@student.sdu.dk

Mathias Elbæk Gregersen
magre14@student.sdu.dk

This page is intentionally left blank

Informations

Institution:	University of Southern Denmark – The Technical Faculty
Course code:	RB-BAP6-U1
Project title:	EasyInsert: Extended user interface and simplification of the interactions with RobWork
Supervisor:	Lars-Peter Ellekilde, lpe@mmmi.sdu.dk
Project period:	1. of February – 22. of May
Number of pages:	50 pages (of this appendices: 3 pages)
Signatures:	

Anders Ellinge
aelli14@student.sdu.dk

Mathias Elbæk Gregersen
magre14@student.sdu.dk

Abstract

RobWork has a tedious process regarding loading and inserting various objects into the WorkCell, a simplified version of this process could be a great asset. The authors had no experience working with RobWork and thus this was acquired over the four months period from February to May that the project lasted. This includes learning how to use the RobWork library, learning to create plugins for RobWorkStudio, learning to create user interfaces using QT and gaining a better understanding of the process regarding loading and inserting various objects from seasoned users of RobWorkStudio. During this period as well, a solution to the above mentioned process was developed using methods and skills involving object oriented C++, software development and programming skills in general. A proof of concept plugin was created for RobWorkStudio, where a user can insert devices and geometric primitives with a few clicks of a mouse, furthermore the addition of removing said objects from the WorkCell was also achieved. There still exists errors and improvements to be made on the solution, however the requirements were met and a foundation for later work was laid so the project was considered a success.

Contents

1 Introduction

While looking for a project to work on for our bachelors degree, it was brought to our attention that there exists different tedious processes of editing the environment of the RobWork work space, involving reconfiguring files, unintuitive user interaction or reloading the software. RobWork is an open-source robotics software used for research and education as well as for practical robot applications. Since we (the authors), as students at the SDU Robotics section of The Maersk Mc-Kinney Moller Institute, would have to get acquainted with RobWork in one way or the other, it was then decided that our bachelor project would revolve around RobWork, and thus our work would hopefully be of great help to both new and old users of RobWork. With skills and experience within software development, object oriented C++ and various programming oriented skills, the authors of this project then tried to solve and implement a solution to above mentioned tedious processes.

To work on this project, it was required to become familiar with RobWork, RobWorkStudio and Qt, which was a large part of this project, therefore those topics are generally explained in section 2, 3 and 4 to give insight and a basis to understand the rest of the project. It is then recommended to read these sections before continuing to the Project Specification in section 5. After the project has been specified the report will then proceed to take the reader on a tour through the solution starting from section 6. The report will then explain various elements of the solution in a proper order the following sections and reveal how these are implemented and what impact they have on the solution.

2 General introduction to the RobWork project

This project has a deep connection to the RobWork project and its systems. This chapter will give a general introduction to the RobWork project and its systems. A more indepth explanation of the RobWork library will follow in section 3.

RobWork is developed by SDURobotics at the University of Southern Denmark and is a collection of C++ libraries [1] that provide a framework and toolbox for applications related to robotics. The framework of RobWork has the following benefits [2]:

1. It provides a flexible structure for modelling robot manipulators and dexterous hands
2. It can be customized and extended for a wide range of applications
3. It has been developed with industrial collaboration in mind

The RobWork project consists of a core and several packages which the user can choose to use. The core is usually just called RobWork and contains the framework for the library. The RobWork project contains 3 standard packages, RobWorkStudio, RobWorkHardware and RobWorkSim. RobWorkStudio is a graphical user interface used to visually represent the work done with RobWork. RobWorkHardware is a collection of drivers. RobWorkSim is a dynamics engine which can be used for e.g. grasping simulation. For this project only RobWorkStudio is used. The user is also capable of writing plugins for the RobWork core and and for the RobWorkStudio package, extending the use of these. An intuitive illustration of the RobWork project can be seen on figure 1.

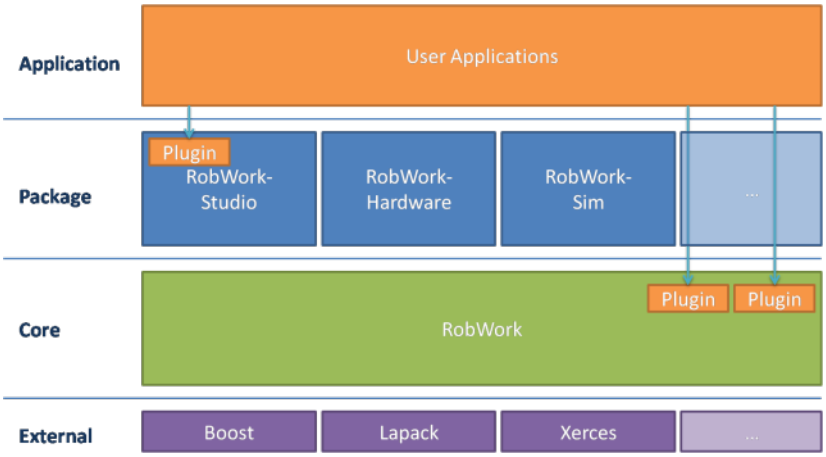


Figure 1: Overview of the RobWork project [3]

3 The RobWork library and its functionalities

This chapter is a general introduction to the RobWork library and the most commonly used data structures and functionalities within. There is a lot more to RobWork than what is in this chapter, the reason behind this chapter is to give a intuitive understanding of the RobWork library. More information about RobWork can be found on official homepage for the RobWork project [1].

3.1 WorkCells

A WorkCell is the basis structure in RobWork. The WorkCell can be thought of as a box containing all of the other building blocks and information needed to represent an environment (See figure 2). The WorkCell most commonly contains Frames, Objects, Devices which are used to represent the different items in the environment. The WorkCell also contains a State Structure used to describe how the items in the environment are related. The WorkCell also contains collision information for the environment.

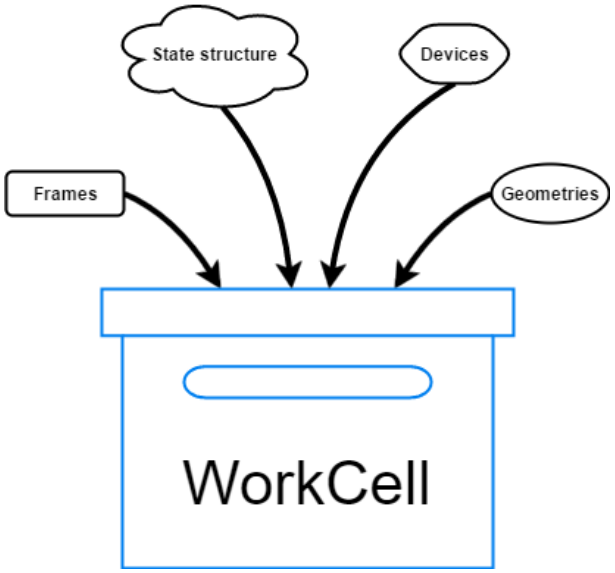


Figure 2: The WorkCell can be seen as a box containing the elements necessary to represent an environment

3.2 Frames

One of the most common data structures from the RobWork library is a frame. A frame is the basic building block in the RobWork library, representing (in the case of RobWork) a local 3D euclidean space. In RobWork frames are built in sequence to each other to form the environment. This means that all frames have a parent and potentially children.

In RobWork frames come in 3 different types: fixed frames, moveable frames and joints.
Fixed frames are frames that have a constant transform relative to the parent frame.
Movable frames are frames which transform can be freely changed.
Joints are frames that can be assigned values for position, velocity limits and acceleration limits. This type of frame is usually used for devices. Joints can be further divided into 3 subtypes: prismatic joint, revolute joint and dependent joint.

Prismatic joints are joints which motion is linear along a constant direction. Thinking of a pneumatic piston can be an intuitive way of thinking about prismatic joints.
Revolute joints are joints which motion is based on a rotation around a single axis. Thinking of hinges can be an intuitive way of thinking about revolute frames.
Dependent joins refer to joints which transform depends on one or multiple other joints. Dependant joins can also be divided into 2 subtypes, dependent prismatic joints and dependent revolute joints, adding the motion specification of the prismatic joint and revolute joint previously mentioned.

Frames in a WorkCell are required to have a parent and are given a unique name so that no frames can be confused for another. Only one frame in the WorkCell has no parent. This frame is called WORLD and is created when the WorkCell is constructed. The WORLD frame can be seen as the global 3D euclidean space for the WorkCell.
In RobWork there is also a special kind of frame called a DAF frame. DAF stands for dynamically attachable frame. A DAF frame is a frame which is implemented so that the parent of the frame is more dynamic to change.

3.3 Objects

Contrary to frames which represents the relationships in the environment, objects represents physical things in the scene.

In order for objects to get a relationship to the environment it is placed in, it has to be associated to a frame. This frame is called the base frame of the object. An object can be associated to multiple frames but only have one base frame.

An object consists of two important elements, a geometry and a model.
A geometry is used to represent the actual geometry of the object. The geometry can be scaled and transformed to allow for modifications. In order to perform a transform, the geometry need a reference. This is done with having the reference be a frame, a reference frame. RobWork is capable of creating simple geometries like spheres, boxes and cylinders, however it is also possible to import complex geometries. Geometries are also being used for the collision detection provided in RobWork.
A model is a graphical representation of the object. Models consists of geometries, materials, colors and texture information as well. It is also possible to apply a transform to a model and get the transform of a model.
Usually when an object is created, a geometry is created for collision detection and a model is created to visually represent the object in a viewer (e.g. RobWorkStudio).

There are two types of objects in RobWork, rigid objects and deformable objects. Rigid objects are objects which geometry does not change. Rigid objects can also posses information about inertia and mass. A deformable object however has the ability to alter its geometry via control nodes.

3.4 Devices

A device can be considered a description of a arbitrary device e.g. the FANUC LRM200 robotic arm (Seen on figure 3). The device also contains the configurations for the device it represents. These configurations are contained in a single configuration vector, making it easy to control the device. In case of a joint device (like the FANUC LRM200), it is

also possible to get and set the bounds, velocity limits and acceleration limits for the joints.

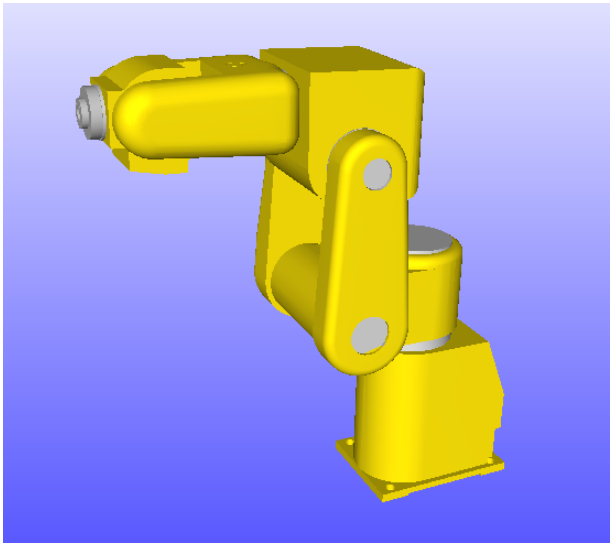


Figure 3: Model of a FANUC LRM200 in RobWorkStudio

Devices can be of 3 different types: joint device, mobile device and SE3 device. Mobile devices are devices that is differentially controlled e.g. a robot rover. Joint devices are devices that consists of moving joints much like the previously mentioned FANUC LRM200 robotic arm. SE3 devices are devices that can move in a 3D euclidean space and is not consisting of joints e.g. a drone.

Joint devices can be of 5 types: Serial devices, tree devices, parallel devices, composite devices and composite joint devices.
Serial devices are the simplest form of device since it consist of joints set in serial to each other. Many simple robotic arms like the before mentioned FANUC LRM200 are serial devices.
Tree devices are devices which joints follow a tree structure, meaning that a joint can have multiple children but only one parent joint. This also means that a tree device must have more than one end effector. This type of device is typically seen in dexterous hands.
Parallel devices are devices that, at some point in the structure of the device, creates a circle. E.g. a joint goes to two joints that then both go to the same joint. The two middle joints are said to be in parallel to each other and are called parallel legs in RobWork. Parallel legs can consist of multiple joints as well as just one.
Composite devices and composite joint devices are devices that are constructed from a series of other devices. The devices in the composite device may not share joints. Just like tree devices, composite devices can have multiple end effectors. However unlike the tree devices, composite devices does not require the path to the end effectors to have a common base. The difference between composite devices and composite joint device is that the devices used in a composite joint device needs to be of the joint device type, whereas in a composite device this is not a requirement.
Illustrations of the joint device types can be seen on figure 4.

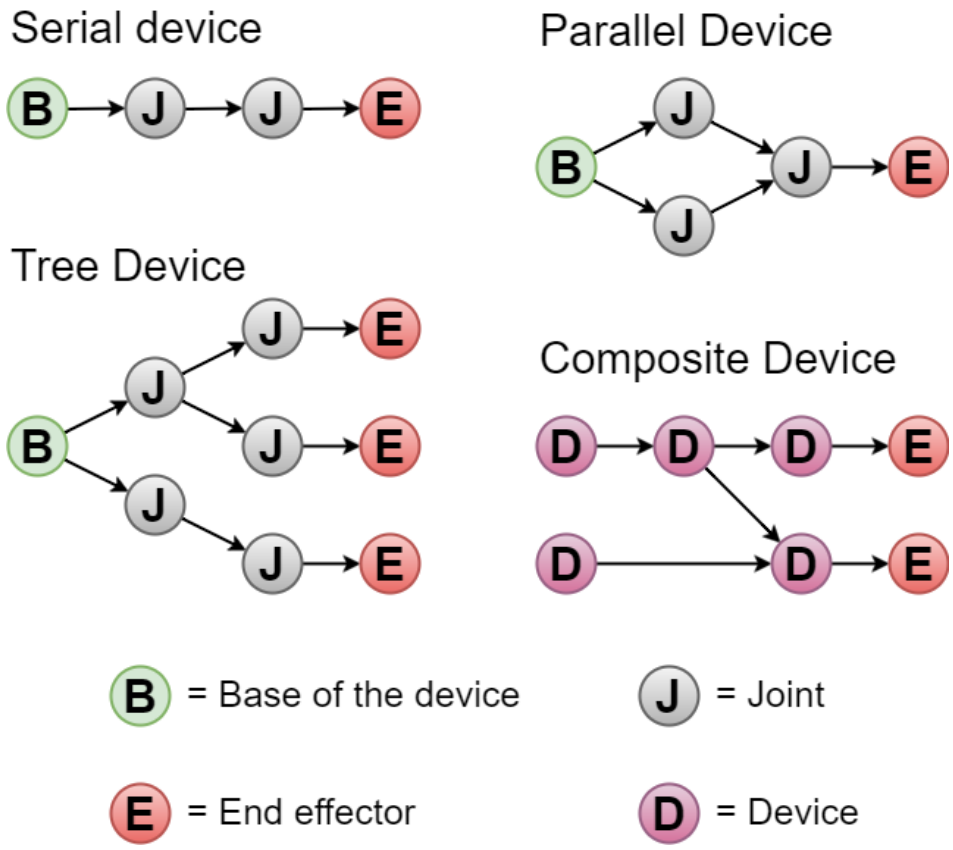


Figure 4: The WorkCell can be seen as a box containing the elements necessary to represent an environment

3.5 3D Transforms

In the RobWork library the general way of moving around frames and other items is through the concept of transforms. For applications using a 3D euclidean space, one would normally use 3D transforms. A 3D transform is a 4 by 4 matrix. This matrix is created from a 3 element vector containing the displacement and a 3 by 3 matrix containing the rotation. In RobWork, the rotation is commonly given in roll, pitch and yaw values, which is a way to describe the rotation with 3 values instead of the 3 by 3 matrix. Roll, pitch and yaw is normally referred to as the RPY or R, P, Y values. On figure 5 is an illustration of what the RPY values represent using a plane.

3.6 States and State Structures

In RobWork the structure of the frames are represented through a class called State Structure. The State Structure also holds the state of all the frames in the form of a class called State. The State is a collection of the states of all the frames contained in the State Structure. This is done as a kinematic tree.

3.7 Namespacing in RobWork

The RobWork library is neatly divided into sections. Some of the more used sections are kinematics, models, geometry, loaders and math. When writing code using the RobWork library, the namespace of the class is the same as the segment of the library. As

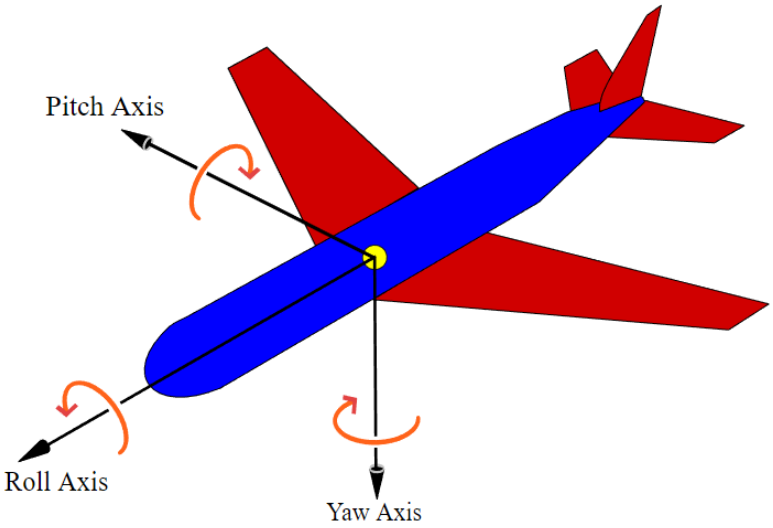


Figure 5: Illustration of the RPY values using a plane [4]

an example, accessing the frame class from the kinematics section would be done like rw::kinematics::Frame. This makes it easier to locate the implementation of a class or functionality e.g. the implementation of the frame class is in the kinematics folder.

Most of the functionalities in RobWork are implemented as static functions. This makes coding with the RobWork library more intuitive since for some of the functionalities in RobWork it does not make sense to require an object to access the functionalities. As an example, accessing the load function of the XMLRWLoader (which loads in a XML file describing the WorkCell) can be done simple by writing rw::loaders::XMLRWLoader::load(input for function), instead of creating a object of the XMLRWLoader and then calling load on the object.

3.8 Typical use of RobWorkStudio and WorkCell-files

Typically when using RobWorkStudio, the user create a WorkCell file written in XML. This file contains the necessary information for creating a WorkCell. The WorkCell file can then be loaded into RobWorkStudio by using the open button. The WorkCell file can also be loaded manually with the before mentioned XMLRWLoader returning a WorkCell object. When creating a WorkCell file it is necessary to know the tags used by RobWork. The root element in a WorkCell file is the WorkCell tag. The WorkCell tag should be written with the attribute "name", giving the WorkCell a name. Inside the WorkCell tag different tags can be used to add the elements of the WorkCell. Some of the more common tags used are: "frame", "RPY", "pos", "joint", geometry tags, device tags and the include tag.
The frame tag is used to define a frame in the structure. The frame tag needs to be supplied with a name attribute and a type attribute defining the frame type.
The RPY tag is used within the frame tag to define the rotation of the frame and the pos tag is used to define the displacement. A transform can be used instead of these if the transform is available.
The joint tag is representing a frame of the type joint. This tag also needs a "name" attribute and a "type" attribute to define the type of joint.

There are also some simple geometry tags that are used to define geometric primitives like a box. More complex geometries can be included via the polytype tag taking in the path for the model file.
Devices can be defined using different tags that define the different kind of devices in RobWork. As an example the serial device FANUC LRM200 would have a serial device tag with joint tags inside.
The include tag is used to include another XML file's content. This is usually used to include complex devices into an environment. This allows users to create only one description of a device and then include it whenever it is used. An example of a WorkCell described in XML can be seen in figure 6.

```
<!-- WorkCell with the name attribute set to Scene -->
<WorkCell name="Scene">

<!-- New Frame called Table added to the World frame -->
  <Frame name="Table" refframe="WORLD" >
    <!-- Transform of the Table frame -->
    <RPY>0 0 0</RPY><Pos>0.2 0.2 -0.408</Pos>
  </Frame>

<!-- New model named Table added to the Table Frame -->
  <Drawable name="Table" refframe="Table" >
    <!-- Transform of the model Table -->
    <RPY> 0 0 0</RPY> <Pos> 0 0 0 </Pos>
    <!-- Model information (box) of the model -->
    <Box x="0.8" y="0.8" z="0.816" />
  </Drawable>

<!-- New Frame called URMount is added to the frame Table -->
  <Frame name="URMount">
    <!-- Transform of the URMount frame -->
    <RPY>90 0 0</RPY><Pos>0 0 0</Pos>
  </Frame>

<!-- Include the UR robotic arm from another xml file -->
  <Include file="../../XMLDevices/UR-6-85-5-A/UR.wc.xml" />

</WorkCell>
```

Figure 6: Example of a WorkCell written in XML containing a UR robotic arm on a table. This example is from the examples following the the RobWork library (ModelData/XMLScenes/RobotOnTable)

4 Qt

Qt, pronounced "cute", is an open source cross-platform framework, mostly used for GUI(graphical user interface) programming. Qt has an easy to (re)use API(application programming interface), which in return gives high developer productivity. Qt is C++ class library, hence new developers using Qt should have some understanding of C++. This chapter introduces terminologies used in Qt, and tries to give some general insight to how Qt operates and works regarding GUI development. For more information please refer to [5].

4.1 Qt Class Hierarchy and Object Model

Qt broadly uses inheritance to create subclasses of instances in a natural way. QObject is the most basic class in Qt, see figure 7. A lot of classes inherit from QObject, like QWidget, which is the base of all user interface objects.

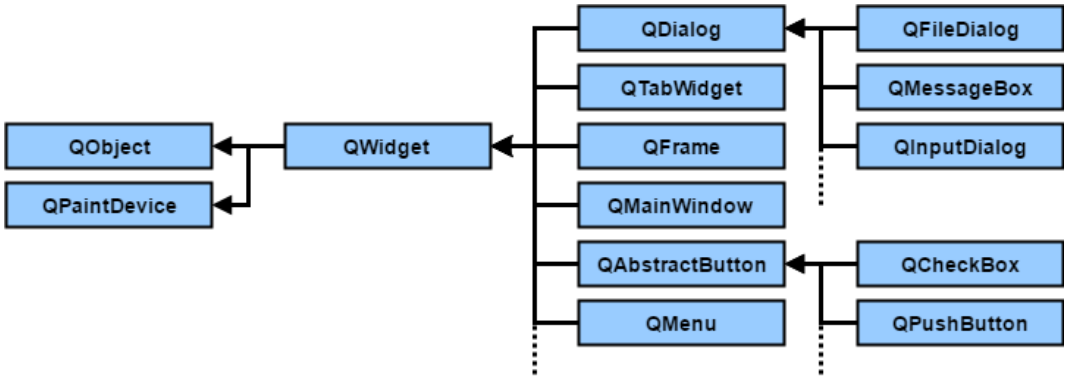


Figure 7: Illustration of some of the hierarchy that Qt is structured as. A complete illustration would be massive an span hundreds of classes.

C++ offers efficient runtime for an object oriented scheme, but lacks in regard to flexibility due to the static nature of the C++ Object Model. Qt has implemented the QObject as the hearth of the Qt Object Model, which preserve the efficient runtime while also offering more flexibility for the GUI domain. The Qt Object Model is implemented with standard C++ techniques. Some of the features that the Qt Object Model adds are e.g.

- Inter-Object Communication called Signal and Slots in the Qt Object Model. This topic is expanded upon in section 4.3.
- Object Trees which structures ownership of objects in a natural fashion. This topic is expanded upon in section 4.2.

4.1.1 The Meta-Object System

Due to the Qt Object Model the Meta-Object System was in turn created, which on the bottom line provides the Signal and Slots for inter-object communication and other features from the the QT Object Model. The Meta-Object System is based on three things: **a)** the QObject class **b)** the Q_OBJECT macro and **c)** the Meta-Object compiler(moc). Each QObject or subclass of QObject has an instance of QMetaObject created to hold the meta-data information, e.g. the name of the class or the class's meta-methods(signal, slots and other member functions). The Q_OBJECT macro helps and defines the meta data for

the moc at compile time. Please refer to figure 8 to see the influence of the Meta-Object System in compile time.

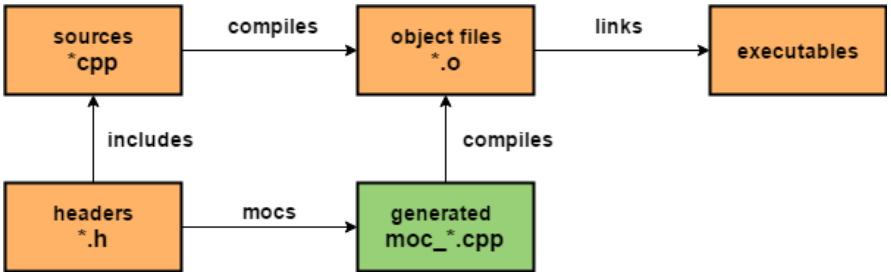


Figure 8: This figure shows how the the Meta-Object System is integrated at compile time. The yellow boxes indicate the normal C++ compiling procedure, whereas the green box is the added moc, which is compiled into the object files

4.2 QWidgets

QWidget is the base of all user interface objects(buttons, menus etc.). QWidget handles all events from the system the application is running on. In Qt, events are QEvent objects which is created upon outside activity (like a click on a mouse). Subclasses of QEvent involve more parameters to characterize a certain event, e.g. mousePressEvent(QMouseEvent* event). The event object is then sent to a specific QWidget object (maybe a button) and the QWidget handles the event with the according event handler.

As mentioned in 4.1 ownership of objects is structured in a tree, this means that a QWidget can have QWidget's within it self, see figure 9. It is the parent's responsibility to delete all it's children, if the parent instance is deleted. A QWidget with no parent is called a top-level widget, which means the QWidget is an independent window. An instance like QWidget subclass QDialog(a pop up dialog window) is a top-level widget. QDialog can be instantiated with a parent, but the QDialog is still a top-level widget in this case, though the position of the dialog window is now centred relative to the parent.

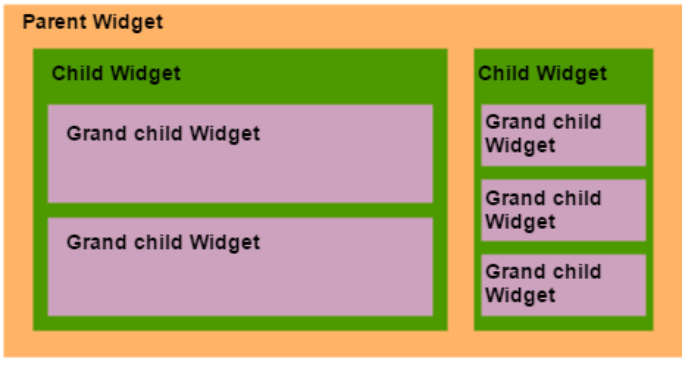


Figure 9: Qt structures ownership of objects in a parent child relationship. The diagram shows a parent widget with various child widgets in a layout, more on layouts in section 4.2.2.

When a QWidget is used as a container to hold and group children, the QWidget is called a composite widget. A parent widget is clipped to the size that it children requires, though this can be changed in the widget's size policy.

4.2.1 QMainWindow

QMainWindow is a subclass of QWidget, and is very essential to a Qt GUI application, since the QMainWindow is a framework for the application user interface. As seen on figure 10 a QMainWindow can have a menu bar widget, toolbar bar widgets, docked widgets and a status bar widget, though a QMainWindow must have a central widget, even if that widget is only a empty placeholder.

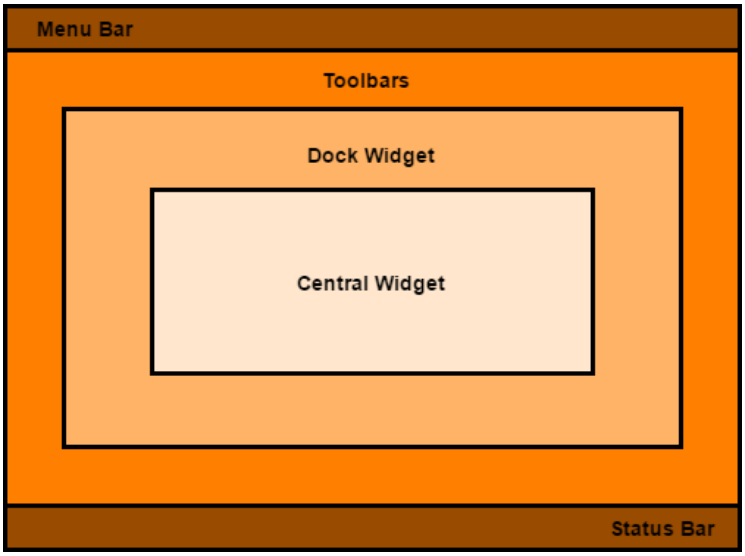


Figure 10: This figure shows how a QMainWindow object looks like. RWS uses QMainWindow as the main application widget.

QMainWindow is usually a good class to use as the framework for an GUI application, though it is optional whether to use it or not. In the case of RWS, QMainWindow is used, and figure 10 nicely reflect the structure of RWS' GUI, where the central widget is a custom subclassed QWidget using Qt GUI modules providing classes for OpenGL integration for graphic rendering. The central widget of RWS will also be referred to as the 3D View. Various plugins to RWS are available to be docked in the docking area, or to be top-level windows (more on plugins in section 4.4) and tool bars and a menu bar are present as well for use.

4.2.2 QLayout

QLayout is a subclass of QObject and QLayoutItem and is the base class of geometry managers. QLayout and its subclasses are managers for the layout of a group of widgets laid out in an application. All QWidget subclasses can use layouts to manage it's children e.g. figure 9 uses the parent widget as composite widget with two composite children. The parent on figure 9 use a QHBoxLayout, which lines the child widgets horizontally and makes each child fill one box. The children then has their own QVBoxLayout, which lines their children (grand children) vertically and assign each of those their own box as well. Figure 11 shows how an implementation of five arbitrary widgets laid in the layout as in figure 9.

```
QWidget* parent = new QWidget();              //Parent widget
QHBoxLayout* pL = new QHBoxLayout();          //Layout of the parent widget
QWidget* cL = new QWidget(parent);            //Left child widget
QVBoxLayout* cLL = new QHBoxLayout();         //Layout of left child widget
QWidget* cR = new QWidget(parent);            //Right child widget
QVBoxLayout* cRL = new QHBoxLayout();         //Layout of right child widget
pL->addWidget(cR); pL->addWidget(cR);         //add children to layout
parent->setLayout(pL);                        //set layout
QWidget* b1 = new QWidget(cL), b2 = new QWidget(cL), b3 = new QWidget(cR),
         b4 = new QWidget(cR), b5 = new QWidget(cR); //create grandchildren
cLL->addWidget(b1), cLL->addWidget(b2); //grand children added to layout
cL->setLayout(cLL);                   //set layout
cRL->addWidget(b3), cRL->addWidget(b3), cRL->addWidget(b5);
cR->setLayout(cRL);
```

Figure 11: This code shows how to make a composite widget with a layout like figure 9.

4.3 Signal and Slots

Signals and slots are one of the more unique features provided by Qt, compared to other GUI frameworks. Signals and slots are used for inter-object communication, and it is made possible by the meta-object system (see section 4.1.1).The Signal and Slot mechanism is similar to callbacks, which basically are function pointers passed as arguments to other code, that is expected to call back the argument at some time. Qt made signals and slots instead of using callbacks to provide simplicity and ensure type-correctness of callback arguments.

The Signal and Slot mechanism, see figure 12, can be thought of as an implementation of the Observer Pattern. A widget or object (the subject) can emit signals when particular events occurs, these signals can then be connected to slots of other objects (the observers). This means, when an object emits a signal, the connected object(s) slot will be called and executed. A signal can be connected to as many slots as needed, and slots can have as many signals connected to it as needed. A slot does not know if it has a signal connected to it, and an object does not know if anything receives its signals.

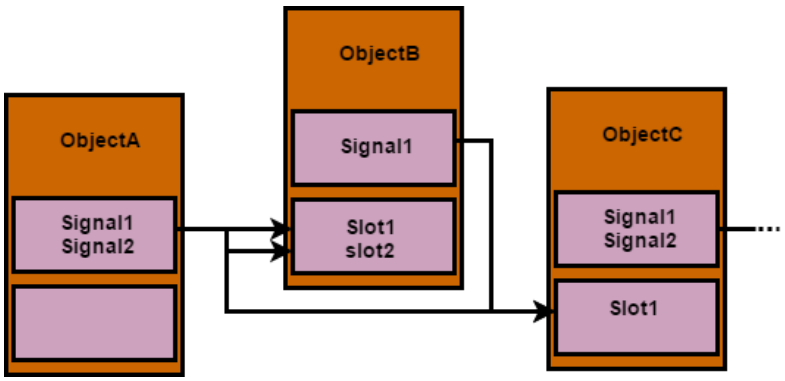


Figure 12: This figure shows how three objects can signal events to a specific slot of another object.

Qt widgets comes with many signals and slots for easy use, but these widgets can of course be subclassed to add our own signals and slots. Signals are public access functions

and can be emitted from anywhere. Slots are normal member functions, though the they have to be defined in the .h file as seen on figure 13, so the moc can find them. Likewise signals are defined in the .h file.

```
class object : public QWidget
{
    Q_OBJECT
public:
    object(QWidget *parent = 0);
    ~object();
private slots:
    void slot1();
signals:
    void signal1();
}
```

Figure 13: This is the .h file for an arbitrary object subclass of QWidget. It shows how the object can be constructed with a parent. The object also has the macro Q_OBJECT to help the moc define the slot and signal.

The Signal and Slot mechanism is independent of any GUI event loop. That means that any QThread, inherited by QObject, runs its own event loop and has no influence on the signal/slot mechanism, though it should be noted that using direct connections, where sender and receiver live in different threads, can be unsafe.

4.4 Plugin

Qt has a low-level API for extending Qt applications. This means that it is possible to make a custom widget(plugin), which can dynamically be loaded in and out of the application. E.g. if the application lacks a user functionality, then it can be written without changing the main application. This is nice, because then third-party developers can easily extend the application while keeping it separate from the source code of the application.

Qt plugins are stored as shared libraries (.so for linux), and are loaded on runtime with QPluginLoader. Though to extend an application with plugins, the application need to define a set of interfaces (these need to pure virtual functions), so that the plugins can communicate with the application. Further more, the moc should be made aware of the interface with Q_DECLARE_INTERFACE() in the header file of the defined set of interfaces. Lastly qobject_cast() should be used to test whether a plugin interfaces correctly to the application. See figure 14 for an illustration of how RWS uses plugins.

Creating a plugin involves declaring the plugin class inherited by QObject and the interfaces the plugin wants to provide. The plugin then need to use the Q_INTERFACES() macro to tell the moc about the interfaces. With the Q_PLUGIN_METADATA() macro the plugin is exported, the macro instantiates the meta-data. The build process of the plugin should then be appropriate using any preferred method. (E.g. CMake.)

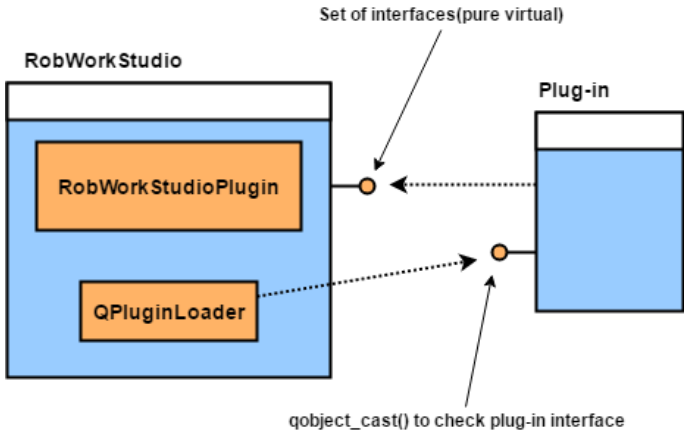


Figure 14: RWS has a class RobWorkStudioPlugin to define the interfaces for plugins to RWS. The plugin is then loaded into RWS, with the QPluginLoader class, and tested for interface compatibility with qobject_cast().

5 Project specification and analysis

Now that the a general understanding of the RobWork project and library is attained, the project can be specified. This chapter also goes through the process of understanding the problem that is the basis of the project and help specify a generalization of how the solution should look. In the end a specified list of requirements is acquired. These requirements are used to guide the solution and in the end secure a successful project.

5.1 Specifying the project

As briefly mentioned in the introduction, this project is about easing the process of using RobWork and RobWorkStudio. It was agreed upon that this should be done through a plugin to RobWorkStudio, since this would have several benefits. One of the benefits of making a plugin versus coding the solution directly into RobWorkStudio was that compiling and testing would be faster. Another good reason was that it would be easier to distribute.

The way of easing the process of using RobWork, that this project works with, is related to the creation of an environment in RobWork. This means easing the process of adding elements to a WorkCell. Therefore the plugin should, via a user interface, be able to insert RobWork related elements. In this project the elements that were deemed important to work with were, frames, devices and some geometric primitives.

As explained in chapter 3.8 this is usually done through an XML file. Doing it this way is seen by some as a tedious process. As new users ourselves, we also saw the potentially steep learning curve for using XML files. This is unwanted if a new user just wants to, as an example, create a simple model of an environment.

5.2 Analysis of the use of RobWorkStudio

Since we (the authors) had never worked with RobWorkStudio before. We had no personal experience regarding the problem and had to gather information about how RobWorkStudio is used by current users before implementing a solution. The primary method used

to gather this information has been through interviews, to learn about the difficulties and limitations of the current solution/system.

The interviews conducted were largely semi-structured interviews, i.e. an open interview with a template for the interviewer to direct the interview in a proper direction. See appendix A for the template used. Be aware that this template does not truly reflect what was learned from the interview. The reason for this type of interview, was to keep an open mind and try to get as many creative suggestions and inputs on how the interviewees uses RobWorkStudio and how they alternatively would like to use it. The interviews lasted from 15 minutes to 30 minutes depending on the interviewee.

The following people were interviewed:

Lars-Peter Ellekilde	Lektor at The Mærsk Mc-Kinney Møller Institute, SDU Robotics
Thomas Nicky Thuelsen	Engineer, Research Assistant at The Mærsk Mc-Kinney Møller Institute, SDU Robotics
Thomas Fridolin Iversen	Ph.d student at The Mærsk Mc-Kinney Møller Institute, SDU Robotics
Michael Kjær Schmidt	Student at The Mærsk Mc-Kinney Møller Institute, SDU Robotics
Kristian Møller Hansen	Student at The Mærsk Mc-Kinney Møller Institute, SDU Robotics

There were a general interest for the problem at hand. The tediousness of writing XML files to adjust a WorkCell was confirmed by some of the interviewees. Some of the interviewees also confirmed the expected problem concerning the learning curve of using XML files, saying it felt quite daunting and slow to get started with. The potential of also being able to show something fast in RWS without having to do much work on setting anything up, was seen as a great asset.
Some of the more concrete functionalities and ideas, that was discussed in the interviews, are summarized in the following bullet points.

- There should be some overview of which elements the user can insert. Maybe in categories or in some other intuitive way.
- When inserting anything, a pop-up window with adjustable parameters regarding the element, should appear, so the user can specify how the element should be inserted.
- Insertion of a device, read from an XML file, onto another device as the end-effector should be possible.
- Insertion of a device should snap, in a graphical drag and drop fashion, onto another device.
- When browsing the available devices, a description box with pictures and specification about the device should be shown.
- There should be some way to define a library of devices, which are then available to be loaded into the WorkCell.
- Insertion of an element into the WorkCell should happen in a drag and drop fashion.
- Static primitives and frames should be available as something to be inserted.
- Deletion of elements in the WorkCell should be possible.
- A undo button for the last insertion/deletion should be available for use.

Adding a frame/geometry to a WorkCell, current

Main Success Scenario: The user opens the XML file describing the WorkCell in an editor. The user then writes the appropriate tag for adding the frame/geometry. The user then writes the appropriate tags for adding the additional information for the frame/geometry. The user then saves the WorkCell XML file. The user then swaps to/ starts up RobWorkStudio. The user then loads the WorkCell from the XML file via open in RobWorkStudio.

Alternate Scenarios: The user makes an error, the loader or parser catches this error and stops the loading process. The user is informed of the error.

Figure 15: Use case describing how currently to add a frame/geometry to an environment

Adding a frame/geometry to a WorkCell, potential solution

Main Success Scenario: The user selects the wanted type of frame/geometry. The user then inputs the required information for the selected type of frame/geometry. The frame/geometry is then created and inserted into the WorkCell.

Alternate Scenarios: If the user supplied invalid information, the user is informed of the invalid information.

Figure 16: Use case describing how a solution would potentially add a frame/geometry to an environment

5.3 Defining use cases

From the interviews a clearer understanding of the use of RobWorkStudio and some general pointers towards the design of the solution was established. From this some general use cases were made, describing the current use of RobWorkStudio (with XML files). The use cases created were for adding a frame/geometry, adding a device and deleting an element. The use case for adding a frame/geometry can be seen on figure 15.

Based on these use cases an additional set of use cases were made describing the potential use of the solution. This set of use cases, combined with the first set of use cases, was used to verify that the creation of the solution would actually be beneficial for the user. The use case for adding a frame/geometry for the potential use of the solution can be seen on figure 16. The rest the use cases for both current use and the potential solution can be seen in appendix B.

5.4 Requirement specification

The requirements are used to specify the solution. The requirements for the project is based on the interviews and the use cases. The requirements have been divided into 2 sections, “need to have” and “nice to have”. The “need to have” requirements are requirements that needs to be fulfilled by the solution and should be verifiable, whereas the “nice

to have" requirements should not necessarily be fulfilled and can be more subjective. It was decided, that devices should be described via a XML file, as it is the standard already in use. It also makes it possible to use the parsers from RobWork to get the information from the device. It was also decided, that the ideas revolving use of the 3D view was ignored, since it would require a lot of research contra the time available to come up with a solution. The requirement that the solution should be done as a plugin is not part of these requirements since this was agreed upon before starting the project, hence it is seen as an ingrained part of the project instead of a requirement.

Need to have requirements:

1. The user should be able to create fixed frames and movable frames through the plugin.
   1.1. The user should be able to specify the parent frame for the new frame.
   1.2. It should be possible for the user to give the device an initial placement and rotation.
2. The user should be able to create geometric primitives through the plugin.
   2.1. The geometries are: Boxes, Planes, Spheres, Cones, Cylinders and Tubes.
   2.2. It should be possible for the user to give the geometries an initial placement and rotation.
3. The user should be able to insert a device described by a XML file.
   3.1. It should be possible for the user to supply a path to a XML file containing the description of a device.
   3.2. It should be possible for the user to define the parent frame of the device.
   3.3. It should be possible for the user to give the device an initial placement and rotation.
4. It should be possible to remove frames, geometries and devices through the plugin.

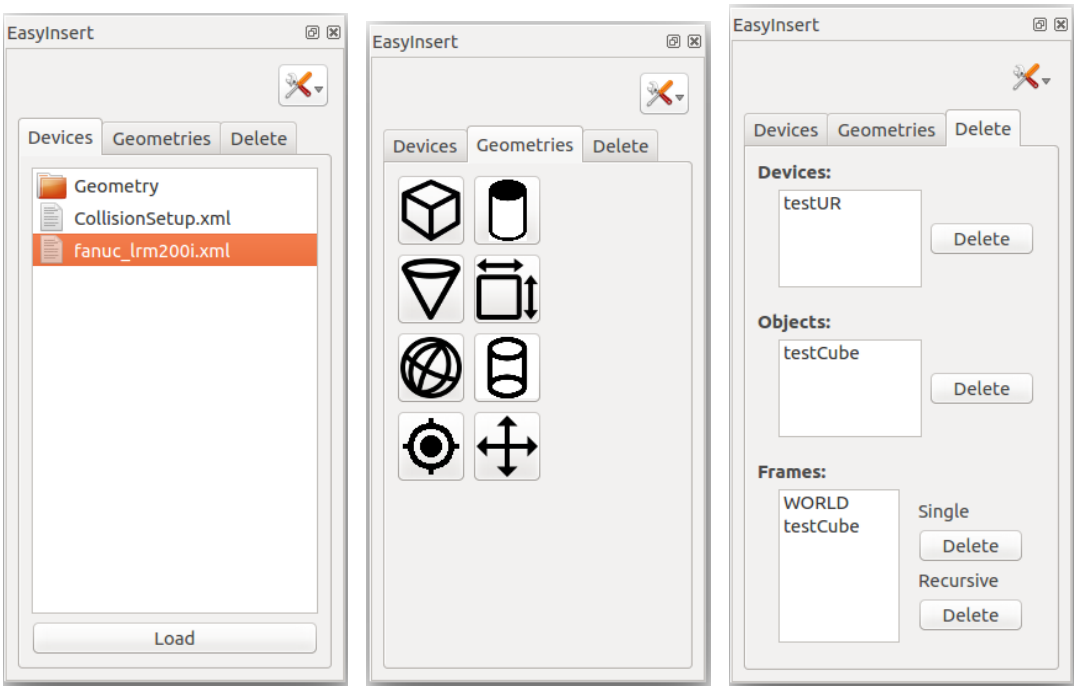
Nice to have requirements:

1. The user interface of the plugin should be intuitive to use.
2. The user should be able to define a library of devices for easier insertion.
3. The user should be able to undo an action via a button or a pressing a key sequence (like ctrl + z).

The following chapters will describe the solution made in regards to these requirements.

6 The User Interface

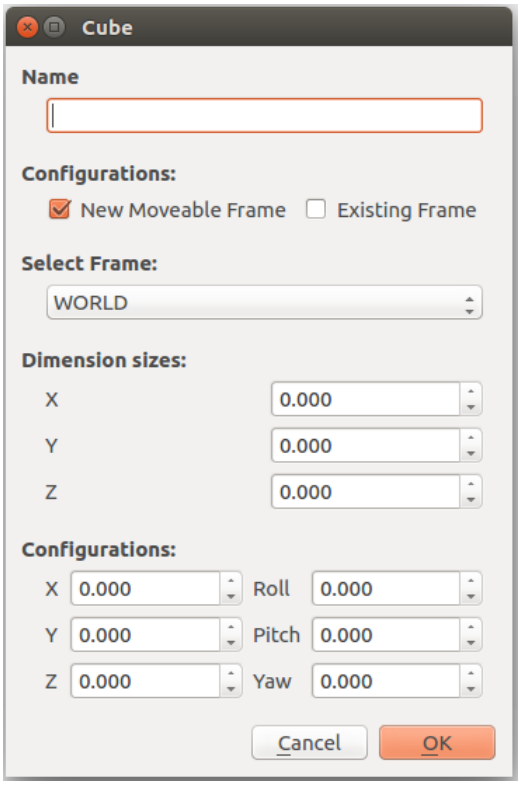
As an introduction to the solution, this section explores the user interface of the plugin. When the plugin is loaded and opened, the user is met with figure 17a. This is the Device tab of EasyInsert and from here it is possible to select a device listed and load it into the WorkCell by pressing the Load button. Upon pressing Load, the user is prompted with a dialog window where the user can specify options about the loaded device. Figure 18b is the exact dialog window the user would see after selecting a device and pressing the Load button. The user can now give the device a unique name, select a frame that the device should be on and specify the configurations such as displacement and rotation. A user can now select e.g. a FANUC LRM200 robotic arm, as the one on figure 3, and insert it. Afterwards the user can then select another device, let's say some kind of hand device, and now insert that device on the end frame of the fanuc. The hand device now acts as the end effector of the fanuc, which only took a few clicks to set up.



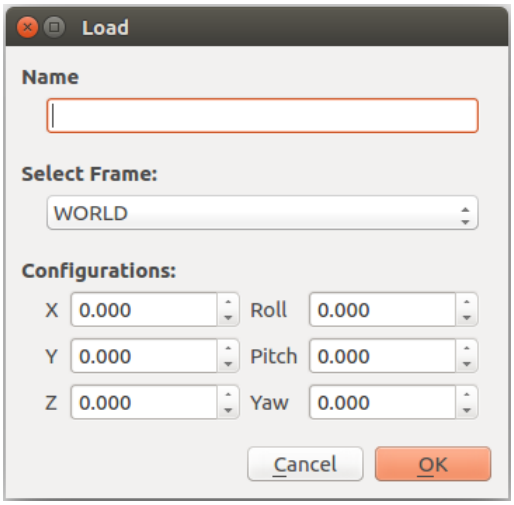
(a) Devices **(b)** Geometries **(c)** Delete

Figure 17: The three tabs a user can interact with in the plugin. In Devices it is possible to select and load a device. In Geometries the user can click an icon (representing a geometry) which then prompts a dialog window with options regarding the insertion of said geometry. In Delete a user can select either a Device, Object or Frame to delete it.

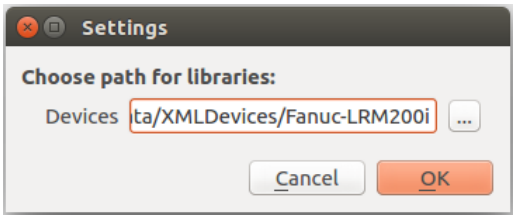
It should be noted, that the first time EasyInsert is loaded and opened, the device tab shows the root content of the operating system. Thus the user needs to change the path of the list shown in the Device tab to a path containing any desired devices. In the right top corner of the plugin is a tool bar button (the only button in the tool bar so far) called settings. Clicking this button and choosing Libraries will prompt the user with the dialog window on Figure 18c. The user can here see what path is used and change it with the [...] button. The [...] button shows a new dialog window where the user can navigate his/her operating system and specify the path that should be shown in the Device tab.



(a) Cube insertion dialog window



(b) Device insertion dialog window



(c) Settings of device path dialog window

Figure 18: Three, out of many, dialog windows the user can experience and interact with when using EasyInsert.

The Geometries tab shows a grid of icons, see figure 17b. These icons represent either geometric primitives or frames that can be inserted into the WorkCell. Currently there are the following objects in the Geometries tab, from top row to bottom: **1.** Cube. **2.** Cylinder. **3.** Cone. **4.** Plane. **5.** Sphere. **6.** Tube. **7.** Fixed frame. **8.** Movable frame. These icons are all buttons the user can click, and after a click has occurred a dialog window appears with options regarding the insertion of said geometry. When the user hovers over a button in the Geometries tab, a tool tip identifying the icon appears, telling the user what sort of object it is. Figure 18a is the dialog window prompted when clicking the cube icon, and as seen, the user can specify a name for the geometry, select a reference frame and set the displacement and rotation configurations, just like when inserting a device. Though for a geometric primitive, like the cube, the user can also specify the dimensions of the geometric figure. The user can also specify whether the primitive should create a movable frame to associate the object with, or if the user want to use an existing frame.

The Delete tab, as seen on figure 17c, is a tab where the user can select a device, object or a frame and press either the Single Delete or Recursive Delete button to remove said item from the WorkCell. The Devices list, shown in the Delete tab, shows a device name. This means that when the user deletes a device, the frames associated with that device are now "free", i.e. the device property is deleted, but the frames and objects are still in the WorkCell. These object and frames will now show up in the Object list and the Frames list of the delete tab, and the user can now remove the frames and objects from the

WorkCell. The Object list of the delete tab only deletes objects, but if the user selects a frame from the Frame list and deletes it with either the Single Delete or Recursive Delete, all objects associated with that frame will also be deleted. Single Delete will only delete the selected frame, and if that frame has children, then the user will be noticed and the deletion is cancelled. Recursive Delete deletes the frame and all children of said frame. It should be noted that there are various problems with the frame delete functionalities, certain scenarios will cause a segmentation fault (properly a rogue pointer somewhere) in the program because of different reasons. These issues will be further discussed in section 10.8.

7 Inserting frames and geometries

This section contains the description of the implementation of inserting frames and geometries. This is done through a class called creator made specifically for creating RobWork specific objects.

7.1 General explanation of the solution

The solution to the requirements related to insertion of frames and geometries was condensed into two parts, a user interface for the information needed (GUI) and the actions needed to insert the frames with the informations gotten from the user interface. In order to simplify the process of inserting frames and geometries, with the information given, this was separated into an extension to the RobWork library. The name chosen for this extension was creator since its focus was on creating and adding elements from the RobWork library.

Separating the problem into the GUI and creator have several benefits. First of all, it has the benefit of modularity. The creator is not dependant on the GUI and is written in a way that allows use in other applications than this one.
Another reason is that it makes it easier to divide the task as long as the the information needed is agreed upon.

7.2 Using the creator

The creator follows the same namespacing technique as RobWork employ in order to make it more intuitive to use next to RobWork. However instead of using rw as the first namespace, ei (for EasyInsert) is used. This could be changed to rw in order make the blend perfect, however it was chosen not to in order to make the user aware that the creator is not part of the official RobWork library.

In the case of frames, the creator is capable of creating fixed frames and movable frames. The creator is also able to add them to a supplied WorkCell and in the case of the movable frame, give it an initial transform in the WorkCell. The fixed frame should always be supplied with a transform, since this is a criteria for the fixed frame type. Both functionalities returns a handle to the newly created frame which the user can use instead of being forced to search the WorkCell for a handle after creating the frame.

In the case of geometric primitives, the creator is capable of adding 6 different geometric primitives to a specified WorkCell: boxes, planes, spheres, cones, cylinders and tubes. No matter what geometric primitive the user wishes to create, the user needs to provide the WorkCell. The user also needs to provide a pointer to the frame they wish to include the geometric primitive to. Instead of this the user can supply the name of a parent frame for a new movable frame on which the geometric primitive is put. The user also needs to provide the necessary information in order to create the different geometric primitives. This varies between the different geometric primitives, a list of the different inputs need for the different geometric primitives can be seen on figure 19. It is also possible to supply the function with a transform which then represents the local transformation of the geometric primitive in relation to the frame on which it has been included.

Geometry	Input type: input name
Box	float: x, float: y, float: z
Plane	None
Sphere	float: radius
Cone	float: radius, float: float: height
Cylinder	float: radius, float: height
Tube	float: radius, float: thickness, float: height

Figure 19: Table of geometric specific inputs for the individual geometric primitives

Since a lot of transformations are used in the creator it was decided also to create a function to easily create transform objects from R,P,Y and x,y,z values. The function, called getTransform3D, takes in these values and returns a Transform3D object.

An example of adding a sphere to a new WorkCell can be seen on figure 20.

```
1   // Create new wc
2   WorkCell::Ptr wc = ownedPtr(new WorkCell("wc"));
3
4   // Radius of 10 cm
5   float radius = 0.1;
6
7   // Displacement of 10 cm in x, y and z. No rotation
8   Transform3D<double> transform = getTransform3D(0.1, 0.1, 0.1, 0, 0, 0);
9
10  // Adding sphere to WorkCell
11  ei::creator::addSphere( "testSphere", // Name of Sphere
12                          "WORLD", // Name of parent frame
13                          wc, // Pointer to WorkCell
14                          radius, // Radius of Sphere
15                          transform); // Transform of Sphere
```

Figure 20: Code example of adding a sphere to a new WorkCell. The sphere has a radius of 10 cm and a displacement of 10 cm in x, y and z in relation to the frame on which it is set. The sphere is added to a new movable frame with the parent set to the WORLD frame.

7.3 Implementation of the creator

The creator was implemented with the purpose of using the most upper layers of the RobWork libraries functions to solve as much of the problem as possible. This was done since in RobWork it is possible to access lower levels of the library which makes the RobWork library way more flexible. An example of this could be when working with the frames of a WorkCell the upper layer way of doing it would be accessing the frames through the WorkCell's own functionality for getting frames, whereas the lower layer of doing it would be to access the state structure in the WorkCell and through it access the frames. The reason for using upper layers is that it is usually simpler code, meaning there is less mistakes to

make and the mistakes made are easier to find.

The implementation of the getTransform3D(...) function is rather simple since it utilises some conversions in the RobWork library. The inputs for the function are 6 doubles representing the x, y, z values, which is the displacement, and the R, P, Y values representing the rotation. In order to create a Transform3D object, a vector representing the displacement and a matrix representing the rotation is needed. The displacement vector is easy to create since it is just a vector containing the x, y, z values directly from the input. The rotation matrix however is more difficult to get since the input for the rotation is represented in R, P, Y values. The R, P, Y values needs to be converted to a rotation matrix. Instead of doing the calculations manually, RobWork, albeit a little hidden, can do this for us through the RPY class from the math namespace. First an RPY object is created with the values from the input. Then the member function toRotation3D(...) from the RPY class is called returning the rotation matrix of the given R, P, Y values. The displacement vector and rotation matrix is then used to create a Transform3D object that is returned to the user. It would be possible to just directly implement the functionalities from the toRotation3D(...) function from the RPY class, eliminating the creation of an RPY object. This would increase the computation speed, but not significantly unless the function is used a lot.

7.3.1 Implementation of creating and adding frames

There are a total of four functions in the creator that are related to creating and adding frames to a WorkCell: createFixedFrame(...), createMovableFrame(...), addFixedFrame(...) and addMovableFrame(...). The first two are functions related to creating frames. The implementation of creating frames in the creator is rather simple, since the process of creating frames in RobWork is simple in itself. The createFixedFrame(...) function take the name of the frame and a transform as input, whereas the createMovableFrame(...) only takes the name as input. The functions simply call the constructor for the given frame with the provided parameters. The reasoning behind having these rather simple functionalities is in the context that the creator should be consistent in the functionality it embodies. Another good reason was that this was some of the first functionalities implemented for the creator. At the time the idea was that there would also be a function, atleast for the fixed frame, that took in the x, y, z and R, P, Y values instead of a transform. The function would then calculate the transform by itself and use this transform when creating the frame. This was however deemed unnecessary to implement when the getTransform3D(...) function was implemented.

The last two functions are functions that are capable of creating and adding a frame to a specific WorkCell. Both of the functions take in a WorkCell, a name, a name of a parent frame and a transform. These functions eliminates the need for the user to understand how to create a frame and how to add the frame to the WorkCell. Even though one could say that this is rather simple process to learn, using the creator simplifies the process significantly. This comes at the cost of flexibility since the user is now locked to the implementation of the functions. However it was estimated that in many cases the functionality implemented in the creator could be used. Figure 21 showcases the standard way of creating a frame and then adding it to a WorkCell, versus using the creator.

```
1
2     string name = "test"; // Name of frame
3     string parent = "WORLD"; // Name of parent frame
4     Transform3D<double> transform =  // Transform used
5     ei::creator::getTransform3D(1, 1, 1, 1, 1, 1);
6     WorkCell::Ptr wc; // WorkCell
7
8     /// The standard way of adding a frame ///
9
10    // Create new movable frame object and cast to frame
11    MovableFrame* mframe = new rw::kinematics::MovableFrame(name);
12    Frame* frame = dynamic_cast<rw::kinematics::Frame*>(mframe);
13
14    // Find the parent frame in wc
15    Frame* parentFrame = wc->findFrame(parent);
16
17    // Test for eligible parent
18    if(parentFrame != NULL) {
19
20        // Add the frame to the wc
21        wc->addFrame(frame, parentFrame);
22
23        // Get state of wc
24        State state = wc->getDefaultState();
25
26        // Set transform of the movable frame in the wc
27        mframe->setTransform(transform, state);
28
29        // Upgrade state and update state
30        state = wc->getStateStructure()->upgradeState(state);
31        wc->getStateStructure()->setDefaultState(state);
32    }
33
34
35    /// Using the creator ///
36    ei::creator::addMovableFrame(wc, name, parent, transform);
```

Figure 21: Code example of the standard way of adding a movable frame with a transform and the same process using the creator

7.3.2 Implementation of creating geometries

Since in RobWork a geometric primitive, like a box, is called an object, a way to create an object that contains the information to represent the geometric primitive is needed. Getting inspiration from the implementation of the RWXMLLoader (See section 8.3) in RobWork, it was chosen to use the same factory approach as the loader. This means that when creating the geometry part of the object, the GeometryFactory from the loaders part of RobWork is used. When the model part of the object is created the Model3DFactory is used, again from the loaders part of RobWork. All of this is neatly put into 3 functions, addObject(...), createGeom(...) and createModel(...).

In order to understand the process of adding e.g. a box it is easiest to start from the bottom with what the factories need in order to produce the geometry and the model. From the GeometryFactory a function called load(...) is used. This function takes in a string containing the information about the geometry it needs to produce and a bool that

signifies whether or not the user wishes to use cached geometries. As an example, the syntax for the string when creating a box is "#Box dx dy dz", where dx, dy and dz are the dimensions of the box. The syntaxes for the rest of the geometries can be seen on figure 22. The function create createGeom(...) uses the GeometryFactory to create a geometry and return it.

Geometry	Syntax	Explanation of parametres
Box	"#Box dx dy dz"	The dx, dy and dz are the dimensions of the box
Plane	"#Plane"	Planes need no parameters
Sphere	"#Sphere radius"	The radius is the radius of the sphere
Cone	"#Cone radius height"	The radius is the radius of the cone and the height is the height of the cone
Cylinder	"#Cylinder radius height level"	The radius is the radius of the cylinder, the height is the height of the cylinder and the level is the discretization level of the cylinder
Tube	"#Tube radius height level"	The radius is the radius of the tube, the height is the height of the tube and the level is the discretization level of the tube

Figure 22: Table showing the syntaxes for the different geometric primitives

From the Model3DFactory a function called getModel(...) is used. This function takes in a string that either describes a geometry in the same syntax as the GeometryFactory or a file name. It also takes in a string representing the name of the model. It is rather beneficial that the input for describing the geometry is the same for both the GeometryFactory and the Model3Dfactory since it makes it simpler to implement. The function createModel(...) uses the model3Dfactory to create a model and return it. The function addObject(...) uses both the createGoem(...) and the createModel(...) functions when constructing the object.

The addObject(...) function now needs to be interfaced to the individual geometric primitives. It is also known that the addObject(...) needs to be supplied a string describing the geometry. To do this a series of easy to use functions related to the geometric primitives were created. An example of one of these is the addSphere(...) which takes in the necessary information to describe a sphere (the radius) and then creates the appropriate string for the factories. It then calls the addObject(...) function with this string, and other relevant information, which creates and adds the object to the WorkCell. An example of the function flow, using addSphere(...), can be seen on figure 23.

7.4 The future of the creator

Most of the functionalities that were implemented with the purpose to fulfil the requirements of the solution. The possibilities for the creator is however almost limitless. This is mostly due to the fact that the creator is written like a library extension to the RobWork library. One of the more immediate expansions to the creator that could be made, would be to extend the geometric primitive related functions with the possibility to only create

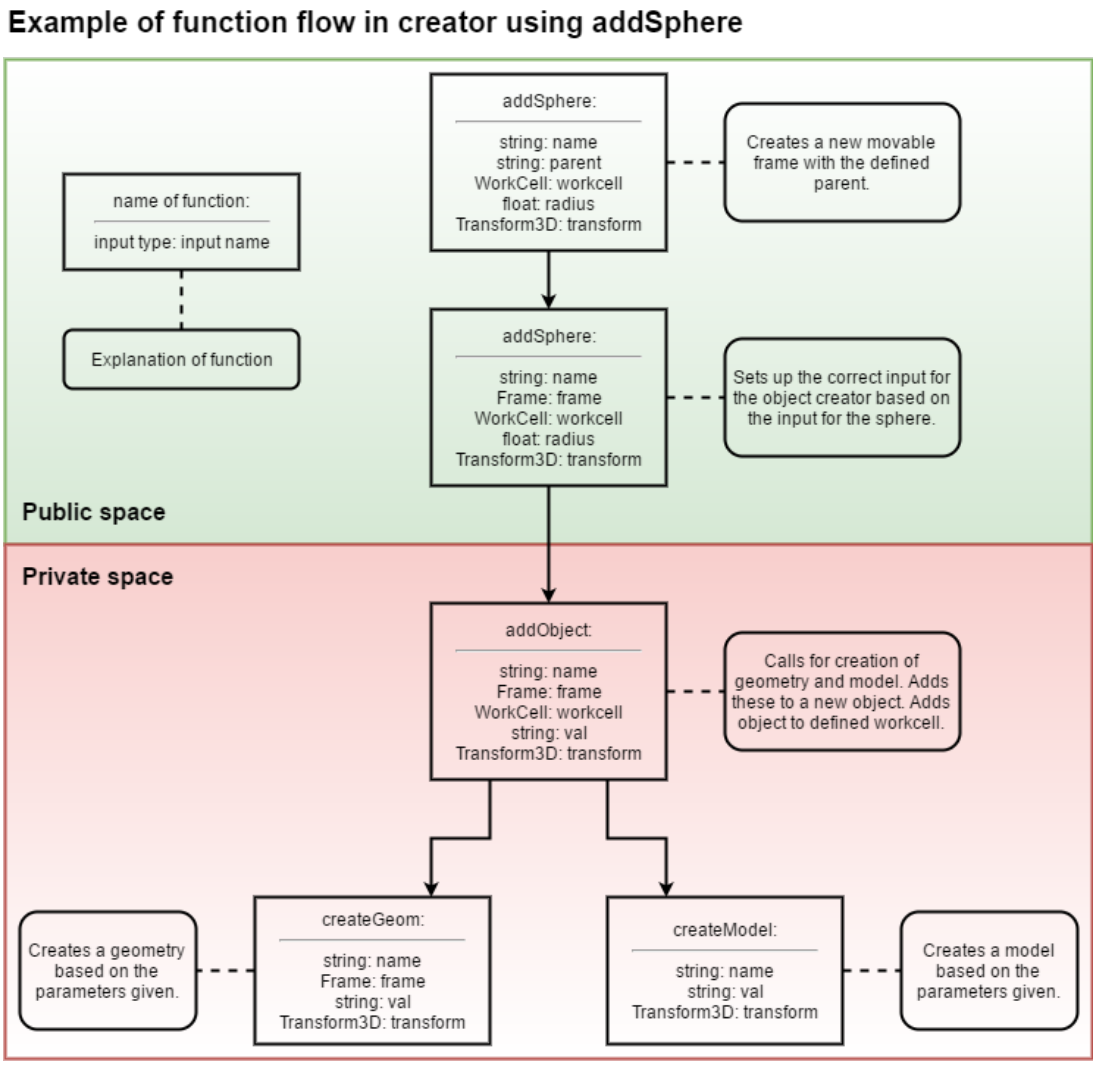


Figure 23: Function flow of the creator using addSphere(...) as example

the geometry or model when creating the object. This extension would make it possible to create visible objects with no collision or invisible objects with collision. Another good addition would be to create a functionality capable of loading geometries from a geometry file. This is already possible in RobWork, so it would make sense to also simplify this process in the creator.

8 Inserting devices

This section contains the description of the implementation regarding inserting devices. The implementation was heavily inspired by the XMLRWLoader from the loaders section of the RobWork library. This was implemented in a class called loader.

8.1 General explanation of the solution

The solution for the requirements related to insertion of devices (See need to have requirement 3) was, just like with inserting frames and geometries, divided into two parts. The first part is the user interface, which is required for the user to be able to supply the necessary information needed. The second part is the process of creating and inserting the device defined by the user. The reason for this separation is the same as the one given in inserting frames and geometries section (section 7.1).

8.2 Using the loader

The loader contains two functionalities the user should be aware of. The first functionality is a function called add(...). This functions adds the device described in a XML file to a given WorkCell. The function add(...) takes in a string representing the path to the WorkCell which the information should be added to, a string representing a custom name of the device and a string containing the name of the frame on which the device should be placed. The user can also supply a transform which is applied to the base frame of the device, giving it an initial placement in the environment. The transform can be supplied in two different ways, the user can supply the function with a transform object or the x, y, z and R, P, Y values.
The loader also contains the functionality to load a WorkCell from an XML file, just like the XMLRWLoader. This functionality was made solely to give the loader more consistency in its functionalities.
An example of using the loader to add a device can be seen on figure 24.

```
1    WorkCell::Ptr wc; // WorkCell needing the device
2    String path = "Some path"; // Path to the device
3    String name = "test"; // Custom name given to device
4    String parent = "WORLD"; // Name of parent frame
5    Transform3D transform; // Initial placement of the device
6
7    // Add device with the parameters given
8    ei::loader::add(path, wc, name, parent, transform);
```

Figure 24: Code example of using the loader to add a device

8.3 Understanding the XMLRWLoader

In order to understand what it takes to load information from a XML file into the format used in RobWork, it is worth the time to study the XMLRWLoader. Another good reason to study the XMLRWLoader is that it solves a problem very similar to the problem of interest in this chapter. There are two main differences, the XMLRWLoader can load items other than devices and it creates a new WorkCell every time it is used.

The function of interest from the XMLRWLoader is the loadWorkCell(...) function. This function takes in a single input which is a string containing the path to the XML file describing the WorkCell.
A good place to start is to analyse the flow that the loadWorkCell(...) function have. The function can be seen as a sequence of actions:

1. Parse the XML file using parseWorkCell(...) from XMLRWParser contained in the loaders section. This takes in the provided path.
   1.1. The information is parsed into a struct called DummyWorkCell that arranges the information neatly for when it should be used.
2. Do sanity check on all the frames
   2.1. Each frame needs to have a valid parent frame
3. Create a new WorkCell object
4. Create and add all the frames, defined in the DummyWorkCell, to the WorkCell
   4.1. Does not include frames from devices
5. Add all frame properties to the WorkCell
6. Create all devices defined in the DummyWorkCell
   6.1. Create and add all device related frames
   6.2. Create all device related models
7. Create and add models, belonging to the added frames, to the WorkCell
8. Create and add all DAF (dynamically attachable) frames to the WorkCell
9. Create and add collision models to the WorkCell
10. Initialize state with initial actions
11. Add devices to the WorkCell
12. Add objects to the scene
13. Create and add collision and proximity setup from corresponding files
    13.1. This only applies if the files are defined in the XML file
14. Add name of the WorkCell XML file and the path to the property map of the WorkCell
15. Return the WorkCell

It should be advised that this sequence is a intuitive understanding of the loadWorkCell(...) function. The function does much more than this, however this is the bread and butter of the function. A more in depth illustration of how the XMLRWLoader functions can be seen on the flowchart contained in figure 25.

The XMLRWLoader also implements a helping struct that is called DummySetup, which task is store information related to the setup. This include several maps related to frames, the world frame of the WorkCell, the state structure of the WorkCell, a map related to objects, a vector contain initial actions, the DummyWorkCell, a map containing the devices, a scene descriptor, collision and proximity setup.

8.4 Implementation of the loader

The implementation of the loader can be explained in the different capabilities that were added as time went on. The first capability of the loader was to add a device to the WorkCell. The second capability was to be able to give the device a name. The third capability was to be able to apply a transform to the device. The last capability added

Figure 25: Illustration of how XMLRWLoader loads the contents of an XML file containing the description of a WorkCell

was the ability to define the parent frame of the device. After all of these capabilities were implemented, the loader complied with requirements for inserting a device (See need to have requirement 3).

8.4.1 Adding a device to a WorkCell

Starting with the loadWorkCell(...) function as a template, the place were the code handles what WorkCell is worked in, needs to be rewritten. One of the first things that happen in the loadWorkCell(...) function is that the DummySetup gets a new scene descriptor. This is changed so that the scene descriptor is set to the one from the defined WorkCell. The next thing related to the WorkCell is the state structure and world frame in the DummySetup. The loadWorkCell(...) function normally creates a new state structure and then uses the root of this state structure as the world frame. This is changed so that the DummySetup is supplied the state structure and the world frame of the defined WorkCell. Lastly the loadWorkCell(...) function creates a new WorkCell, this is simply just removed since it is no longer relevant. Figure 26 shows the changes made to the loadWorkCell(...) template in order for it to support defining a WorkCell.

```
    DummySetup setup;

    /// XMLRWLoader
    // New scene descriptor is created
    setup.scene = ownedPtr(new SceneDescriptor());

    // New State structure is created
    setup.tree = new StateStructure();
    // World frame is saved in setup
    setup.world = setup.tree->getRoot();

    // New WorkCell is created using the parsed information
    WorkCell::Ptr wc = ownedPtr(new WorkCell(
        ownedPtr(setup.tree), setup.dwc->_name, fname));

    // Scene descriptor of WorkCell is set to that in the setup
    wc->setSceneDescriptor(setup.scene);


    /// Changes (wc is the supplied WorkCell)
    // Scene descriptor in setup is set to that of the WorkCell
    setup.scene = wc->getSceneDescriptor();

    // Get state structure from the WorkCell
    setup.tree = wc->getStateStructure().get();
    // Get World frame from the WorkCell
    setup.world = wc->getWorldFrame();
```

Figure 26: Code example showing changes made to the template from the loadWorkCell(...) function regarding the WorkCell

8.4.2 Custom naming the device

In order to choose the name of the device we need to access the information regarding the device and its frames before the device and its frames are created. Luckily, it is already known that this information is situated in the DummyWorkCell. The DummyWorkCell contains a vector with the description of all of the devices. However, taken into account how this functionality is going to be used we assume there is precisely one device description. This description of the device contains the name which is simply changed into the wanted name.

The name of the frames however is the same which is a problem in the case that one would like to include more then one of the same device. In order to solve this a naming convention for frame names was decided. The standard naming convention includes the device name and a description of the frame as the name of the individual frame. As an example, the base frame of a FANUC LRM200 would be named "LRM200.Base". The new naming convention follows this one with the modification that the custom name does not overwrite the device name. It is instead added to the existing name of the frame, preserving the information of the device for the user. Again using the FANUC LRM200 as an example, the new name for the base frame, where the inserted FANUC LRM200 device was renamed test, would be "test.LRM200.Base".

In order to implement the new naming convention for the frames, each frame description

contained in the description of the device needs to be accessed. The frame descriptions are contained in a vector in the device descriptions which is iterated through using a simple for loop. Changing the name of the individual frame is a little different then with the device since the name of a frame consists of two parts, a scope and the actual name of the frame. In the case of the example earlier with "LRM200.Base", the "LRM200" part would be the scoped part and the "Base" part would be the actual name. Luckily for us the scope is implemented as a vector of strings and the problem is solved by simply adding the custom name to this scope vector.

Some new problems now arise. First of all the parent frames for frames which name we changed does no longer point to the right frames any more. In the frame description this is called the reference frame and is a string reflecting the name of the parent frame. This needs to be updated to the new name of the frame. This does not apply to the base frame of the device. The base frame has a parent frame which name has not been changed, since the parent is not part of the device. The name of the reference frame is contained in a single string so the scoping is applied manually.

Another problem is that the models are saved in a map that uses a string, containing the name of the frame the model is associated with, as the key. This key also needs to be changed since it contains the old name of the frame. First all of the models is saved along with the name of the associated frame. The models are then removed from the model map and inserted again with the the correct name of the associated frame. This problem is the same for a map containing information about limits and a map containing information about properties. The problem is solved the same way as with the models.

The last problem is with the collision setup. The problem here is that the name of the frames used in the description of the collision setup, from the parser, are not corrected. The intuitive way of understanding how this is a problem is to imagine the device as a collection of geometries. When drawing the device it is inevitable that these geometries collide. In RobWork this causes a collision between the geometries that make up the device. In order for this not to happen, the geometries that makes up the device are excluded in the collision detection. This is done with the frame names which, after the renaming, are wrong. In order to fix this problem the exclude list (A list containing a pair of excluded frames) is extracted from the collision setup and the frames are renamed in a similar fashion to the way done with the models. In order to add the changes to the collision setup in the WorkCell, the two collision setups are merged using the member function merge(...) from the collision setup.

8.4.3 Adding a transform to the device

Adding a initial transform to the device is relatively simple as long as a handle to the device is available. This handle is fetched by using the name of the device to extract the newly inserted device from the WorkCell. The transform supplied by the user is then applied to the base of the device. The transform can be given in two ways, as a Transform3D object or as the x, y, z and R, P, Y values. In the case that the x, y, z and R, P, Y values are supplied, the loader calculates the transform from these values using the creator (see chapter 7.2).

8.4.4 Defining the parent frame for the device

Changing the parent for the device has a really simple solution. Remember when renaming the device, it was figured out that the reference frames (parent frames) were stored in a string in the frame description. The reference frame for the base frame of the device is the same thing as the parent frame for the device. Hence all that needs to be done to solve this problem is to find the description for the base frame and change the reference frame to the defined parent frame.

8.5 The future of the loader

As the reader might already have suspected, the loader can be used to add much more than a device. In the case that a XML file with the contains more than a single device, all the other items are also added. This can in some cases be useful, it however makes it less intuitive to use the loader. Because of this, one of the more immediate additions to the loader would be to flesh out the functionalities and add catches so the user cannot use the function in an unintended way. The functions, as an example, could be addDevice(...) which adds a single device, addObject(...) which adds a single object described in a XML file and addWC(...) which would add the entire content of another WorkCell described in a XML file to another WorkCell.
The reader might also have noticed that when inserting a single device, a lot of the actions are never used. A good example is the first step where the frames not associated with a device are created. This step is unnecessary since there should be none of these frames in the description of a single device. Since the plan would be to create specialized functions, these functions could be optimized by removing the unnecessary parts of the functions.

9 Dialog Windows

The dialog window is a great tool to communicate with a user with. Dialog windows are top-level windows, and they provide the user with either information, or they allow the users to select options to perform a command or task. This section will show how dialog windows has been created and made available in this project with the dialog class.

9.1 Using the dialog class

The dialog class is fairly simple to use and extend. A dialog is made of blocks in a vertical layout of QWidget's, i.e. a dialog instance is a composite widget containing composite widgets. Figure 27 shows a dialog window with four blocks initialised.

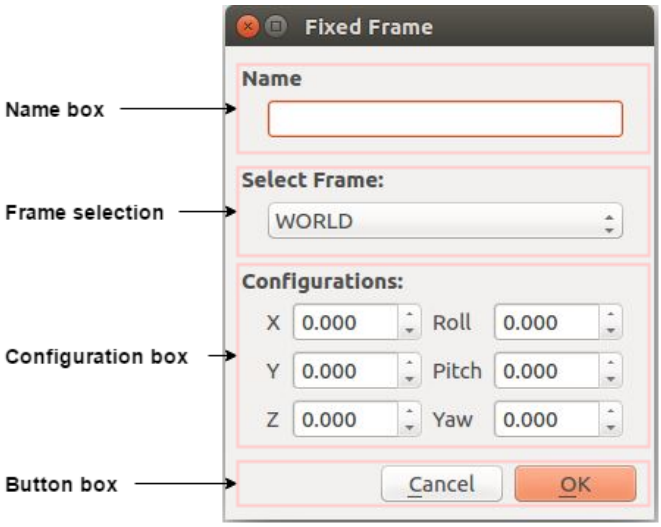


Figure 27: This figure shows a dialog window of adding a fixed frame to a WorkCell containing various widgets relevant to the action.

All the blocks available for dialog windows in the dialog class are shown on figure 32. These blocks are just member functions of the dialog class which returns a composite widget. So to create a dialog window as shown on figure 27, a dialog instance should be constructed (in this case with a WorkCell pointer) and then simply adding the blocks to the dialog with the utility member function addToDialog(). See figure 28 which shows a code example on how to use the dialog class to construct the dialog window as shown on figure 27. Since the dialog class is a subclass of QDialog, the position of the dialog window (opposed to a normal QDialog object) is always centred to the main application (RobWorkStudio), as long as the plugin is docked in RobWorkStudio.

exec(), seen on figure 28 line 8, is an inherited public slot that executes/shows the dialog window. The dialog shown is in modal mode, meaning the window denies all user actions on any other widgets in the application. The function returns a DialogCode containing the information about whether the dialog window was accepted or rejected. The accepted or rejected information is determined by the user input to the createButtonBox() function block, since the OK button emits a accepted() signal to the accept() slot of the dialog object when pressed. Likewise the Cancel emits a rejected() signal to the reject() slot of the dialog object, though a rejected() signal is also emitted when the user presses ESC on the keyboard or the x(close) button of the dialog window.

```
QString st = "Fixed Frame"; //Name of the dialog window
rw::models::WorkCell::Ptr wc = getRobWorkStudio()->getWorkCell(); //
    WorkCell
dialog* geoDialog = new dialog(wc,st,this); //Create the dialog window
geoDialog->addToDialog(geoDialog->createNameBox()); //Adding something
geoDialog->addToDialog(geoDialog->createFrameSelection());
geoDialog->addToDialog(geoDialog->createConfigurationBox());
geoDialog->addToDialog(geoDialog->createButtonBox());
geoDialog->exec(); //execute/open dialog window
```

Figure 28: A sample code to illustrate how to construct the dialog window from figure 27.

So when a dialog window has been created and used by a user, then some functionality should be executed based on the inputs given to the dialog. That means when the function exec() returns, we need to know whether the user accepted or rejected the dialog window (pressed OK or Cancel). This is done with the inherited public function result(). The function result() returns QDialog::Accepted when the dialog was accepted and QDialog::Rejected when it was rejected (QDialog::Accepted and QDialog::Rejected are both just normal enumerators, i.e. 1 and 0).

```
1  geoDialog->exec();
2  if(geoDialog->result() == QDialog::Accepted) //Check the result
3  {
4      std::string name = geoDialog->getNameBox(); //Get the name input
5      std::string frame = geoDialog->getFrameSelection(); //Get frame selc
6
7          /*  Do something with the information    */
8
9  }
10 delete geoDialog; //Delete the dialog object when done using it
```

Figure 29: A sample code illustrating what to do after the exec()function returns.

As seen on figure 29, after the exec() function returns, we simply check whether the result was QDialog::Accepted or not. If the result is accepted, then something should be done with the inputs given to the dialog window. We access these input via self explanatory public get-functions, e.g. the NameBox contains the name input (as seen on figure 29 line 4). After we have collected all the information we wanted out of the dialog window, then we can proceed to do something with the information, like e.g. the insertion of a frame. When we are done with the dialog window it should be deleted, because the dialog object is still alive until the parent is dead. If we do not delete the dialog object the dialog objects, created in the lifetime of the parent, will just waste space in memory.

9.2 Implementaion of the dialog class

The dialogs of EasyInsert has been implemented in it's own class dialog.hpp, i.e. a subclass of QDialog, see figure 13 for an example of a Qt subclass. This was done to keep it independent and self-contained from the rest of the plug-in. The constructor for the dialog class is very simple, see figure 30. A constructor with no WorkCell parameter is also

available and passing a parent is optional, though a name must always be given to the constructor.

```
1 dialog::dialog(rw::models::WorkCell::Ptr wc, QString dialog, QWidget *
      parent)
2     : QDialog(parent)
3 {
4     _workCell = wc;
5     mainLayout = new QVBoxLayout();
6     setWindowTitle(dialog);
7 }
```

Figure 30: The dialog class constructor. A WorkCell pointer to a RW instance is passed along as a parameter. The QString parameter is the name of the dialog box and the QWidget is a pointer to the parent widget

It would take to much time to inspect all of the function blocks on figure 32. Therefore only one function block, namely createNameBox() figure 31 will be inspected. All the function blocks has a similar structure.

```
1 QWidget* dialog::createNameBox() {
2     QGroupBox* nameBox = new QGroupBox(tr("Name")); //The nameBox widget
3     QHBoxLayout *layout = new QHBoxLayout; //layout for the nameBox
4     nameLine = new QLineEdit(); //The line edit the user can type the name
5     layout->addWidget(nameLine); //Add the line edit to the layout
6     nameBox->setLayout(layout); //Set the layout to the namebox
7     return nameBox; //Return the nameBox
8 }
```

Figure 31: This figure shows the pulic function createNameBox() from the dialog class. The function is used to add a name line edit to dialog windows.

The general procedure in the function blocks, as seen on figure 31, is to first create the composite widget. This widget, in the case of createNameBox, is a QGroupBox widget. QGroupBox is a widget that is mainly used as a composite widget. The QGroupBox provides a box to group widgets in, a frame if wanted and a title. As seen on figure 31 line 2, the title is set upon construction with the string "Name". The tr() function is for translation purposes [6]. After the composite widget has been created, the layout for the parent widget has to be chosen and the child widget(s), in this case a QLineEdit(), has to be created. QLineEdit() is a simple line where text can be typed and edited in. When all the child widgets has been created, they need to be added to the layout of the parent. Lastly the layout is set to the parent and the parent is then returned. It should be noted that when the QLineEdit() was created, figure 31 line 4, no parent was defined, i.e. it is actually not a child widget but a top-level widget. This is fine though, because when a top-level widget is added to a layout of another widget, that widget automatically takes responsibility as a parent and the top-level widget is now a child of that widget.

9.3 The future of the dialog class

The dialog class can be extended formidably as a dialog library to create dialog windows for a quite variety of other future plugins or applications. The dialog class could greatly

Function blocks	Description
createButtonBox()	The Cancel and OK buttons. This block is used in all the dialog windows.
createNameBox()	A line edit to specify a name.
createCheckFramesBox()	Two exclusive check boxes, i.e. only one of the boxes can be checked at a time.
createConfigurationBox()	The different configurations available to adjust regarding displacement and rotation of the inserted element.
createConfigurationBoxCube()	A cube specific configuration option, i.e. the dimensions of the cube.
createConfigurationBoxSphere()	A sphere specific configuration option, i.e. the dimensions of the sphere.
createConfigurationBoxCone()	A cone specific configuration option, i.e. the dimensions of the cone.
createConfigurationBoxTube()	A tube specific configuration option, i.e. the dimensions of the tube.
createLibSettingsBox(PropertyMap *map)	This block is used to edit the settings of the device library.
createFrameSelection()	A way to select a parent frame for the inserted element. Default is the WORLD frame.

Figure 32: This table summarizes all the different dialog function blocks used to construct the dialog windows in the plug-in.

benefit from having more layouts upon construction. As of now, blocks of a dialog window has to fit into a vertical layout, but if the class could be extended to support defined layout in the constructor, then a user would be able to customize their dialog windows even further.

10 Implementation of the User Interface

This chapter will go through how EasyInsert achieved the functionalities and interfaces mentioned in section 6. The plugin uses the RobWork libraries, the stand alone classes creator, loader and dialog to achieve many of it's functionalities, while relying on the Qt library to set up the GUI elements.

On figure 33 the constructor for the plugin is seen. Before creating the widgets making up the plugin, all settings are set up with the setupSettings() function on line 4 (discussed in section 10.1). The parent widget of the plugin is then created which is a QScrollArea. A QScrollArea widget provides a scrolling view onto another widget (the child). The QScrollArea widget is created on line 5 and the widget is then set to be resizeable. When a QScrollArea is set to be resizeable, the child widget of the QScrollArea will automatically be resized so to either avoid scroll bars (decrease size of child) or utilize extra space (increase size of child). It should be noted that when QScrollArea has a composite child with a layout, the size policy of that layout will determine the size (or resize) of the widget. The composite child is created on line 7 and then a vertical layout is created. QToolbar is then created on line 9 using the createToolBar() function (discussed in section 10.2), the tool bar is then added to the layout and it is right aligned. On line 12 the QTabWidget is created, a QTabWidget is a widget with stacked tabs of widgets, where the user can select a tab and see the widget associated with that tab. To create a tab, a QWidget needs to be passed to the addTab function and a name can be given as well. On line 13 to 15 all the tabs are created with the help of createDevTab() (section 10.3), createGeoTab() (section 10.4) and createDeleteTab() (section 10.5) which all returns a QWidget. The QTabWidget is then added to the layout and set to the composite child which in turn is set to the QScrollArea widget and finally the QScrollArea widget is set to the QDockWidget of RobWorkStudio(this happens when the plugin is loaded).

```
1  EasyInsert::EasyInsert():
2      RobWorkStudioPlugin("EasyInsert", QIcon(":/pa_icon.png"))
3  {
4      setupSettings();
5      QScrollArea *widg = new QScrollArea(this);
6      widg->setWidgetResizable(true);
7      QWidget *dockWidgetContent = new QWidget(this);
8      QVBoxLayout *verticalLayout = new QVBoxLayout(dockWidgetContent);
9      _toolBar = createToolBar();
10     verticalLayout->addWidget(_toolBar);
11     verticalLayout->setAlignment(_toolBar,Qt::AlignRight);
12     QTabWidget *tabWindow = new QTabWidget(dockWidgetContent);
13     tabWindow->addTab(createDevTab(), "Devices");
14     tabWindow->addTab(createGeoTab(), "Geometries");
15     tabWindow->addTab(createDeleteTab(), "Delete");
16     verticalLayout->addWidget(tabWindow);
17     dockWidgetContent->setLayout(verticalLayout);
18     widg->setWidget(dockWidgetContent);
19     this->setWidget(widg);
20 }
```

Figure 33: This figure show the constructor of the plugin.

10.1 Settings

Defining settings for an application is a common thing. It was therefore decided that Easy-Insert should have some way to set settings. Settings for the plugin is stored in a XML file. The only current setting a user can set is, as mention in 6, the path for the library in the device tab, which is also the 2. nice to have requirement and in turn indirectly also fulfils the need to have requirement 3.1. Though it should be easy to extend the plugin to have more settings, and even other utilities through the tool bar if needed.

The first thing the plugin does under it's construction, is to load the settings. This is done through the setupSettings() function, and the code of the function can be seen on figure 34. It is determined if there exists a settings file for the plugin (eisettings.xml), this is done on line 1 and 2 on figure 34 with the boost library [7]. If a settings file exists, the code will proceed to load the file and warn the user if something went wrong. The loading of the file is done with the XMLPropertyLoader class [8], which loads the settings file into a Property container called a PropertyMap. The PropertyMap class is part of the common namespace and can be used to store various user information, in this case for settings purposes. As seen on figure 34 line 4, the PropertyMap, _propMap, is loaded with the settings for the plugin. On line 13 we store a pointer, _settingsMap, to the PropertyMap of the conveniently named Property EasyInsertSettings from the settings file. The following code checks if any settings were actually loaded, since if the user has not used the plugin yet, no settings files exists yet (or maybe it was deleted). The EasyInsertSettings Property would then have to be added, as seen on line 15, so a proper settings file can be created later. Settings are then stored under this property with an appropriate tag, so it is easy identifiable.

```
1  boost::filesystem::path settingsPath("eisettings.xml");
2  if( exists(settingsPath) ){ //If the file exists
3      try {
4          _propMap = rw::loaders::XMLPropertyLoader::load("eisettings.xml");
5          } catch(rw::common::Exception &e){
6              RW_WARN("Could not load settings from 'eisettings.xml': "
7              << e.getMessage().getText() << "\n Using default settings!");
8          } catch(std::exception &e){
9              RW_WARN("Could not load settings from 'eisettings.xml': "
10             << e.what() << "\n Using default settings!");
11         }
12     }
13 _settingsMap = _propMap.getPtr<PropertyMap>("EasyInsertSettings");
14 if(_settingsMap==NULL){ // if there is no settings set yet
15     _propMap.add("EasyInsertSettings", "Settings for EasyInsert",
           PropertyMap());
16     _settingsMap = _propMap.getPtr<PropertyMap>("EasyInsertSettings");
17 }
```

Figure 34: Code example of how the settings of the plugin are loaded. If there were any trouble loading the settings, the user will be noticed and default settings will be used.

As mentioned in section 6, the user is prompted with a dialog window when pushing the [...] button on figure 18c. This is a QFileDialog [9] which allow users to select a directory. The createLibSettingsBox(PropertyMap *map) function from figure 32, the function block

that is part of figure 18c , signals the public slot setDirectoryDialog() on a click event of the [...] button. On figure 35 the code from the setDirectoryDialog() slot is shown. The QFileDialog instance saves the directory the user chose in the QString dir on line 1, and we then make sure on line 4 that the user didn't select nothing(if he cancels the dialog). The directory is then set in the _settingsMap on line 5 using the PropertyMap member function set(const std::string &identifier, const T &value). The identifier is here "Devices" and the value associated with it is the chosen directory dir. Finally the pathLine, the line edit of figure 18c, is updated as well.

```
1 QString dir = QFileDialog::getExistingDirectory(this,
2         tr("Open Directory"), pathLine->text(),
3         QFileDialog::ShowDirsOnly | QFileDialog::DontResolveSymlinks);
4     if (dir != "") {
5         _settingsMap->set("Devices", dir.toStdString());
6         pathLine->setText(dir);
7     }
```

Figure 35: The setDirectoryDialog() function. This function prompts the user with a QFileDialog dialog window, which allow users to select directory. The selected directory is then set in the settings and the pathLine of the LibSettingsBox is updated.

When the settings dialog from figure 18c returns, it is checked if the user accepted the settings, as seen on figure 36. If the user accepted we try to save the settings using the XMLPropertySaver [10] class and catch any errors. The root path is then updated (see section 10.3 for more information about the root path) with the PropertyMap member function get(const std::string &identifier, const T &defval). "Devices" is here the identifier, that means, if a Device tag exists in the _settingsMap, then the associated setting is returned. Otherwise the default value "/" is returned.

```
1  if (settingsDialog->result() == QDialog::Accepted) {
2      try {
3          rw::loaders::XMLPropertySaver::save(_propMap, "eisettings.xml");
4      } catch(const rw::common::Exception& e) {
5          RW_WARN("Error saving settings file: " << e);
6      } catch(...) {
7          RW_WARN("Error saving settings file due to unknown exception!");
8      }
9      view->setRootIndex(dirmodel->setRootPath(QString::fromStdString(
           _settingsMap->get("Devices", "/"))));
10 }
```

Figure 36: Example code of when a settings dialog window has returned and something should be done to the settings.

10.2 Tool Bar

The settings button, as noted in 6, is the only button available in the tool bar so far, and thus the extend of the tool bar will be shortly discussed in this chapter. The tool bar is constructed in the createToolBar() function (figure 37) and it is pretty straight forward.

The only hiccup can be managing the menu's for the buttons in the tool bar. We create a tool bar with the QToolBar class and the buttons with QToolButton. After the toolbar and button(s) has been created, the icons and tool tips are set. On line 5 a QMenu is created, QMenu is a selection menu. Afterwards a QAction is created as well, which is an abstract user interface action that can be inserted into widgets. We then add the action to the menu. On line 10 the settingsButton gets the menu set, followed by a pop up mode selection. Lastly we add the settingsButton to the toolBar, and we connect the settingsAction to the settings() slot. The settings() slot is the function that creates the settings dialog window on figure 18c and waits for the user to select the settings, before saving them, like on figure 36.

```
1  QToolBar* EasyInsert::createToolBar()
2  {
3      QToolBar *toolBar = new QToolBar(this);
4      QToolButton *settingsButton = new QToolButton();
5      settingsButton->setIcon(QIcon(":/settings.png"));
6      settingsButton->setToolTip("Settings");
7      QMenu *settingsMenu = new QMenu("Settings menu");
8      QAction *settingsAction = new QAction("Libraries",this);
9      settingsMenu->addAction(settingsAction);
10     settingsButton->setMenu(settingsMenu);
11     settingsButton->setPopupMode(QToolButton::InstantPopup);
12     toolBar->addWidget(settingsButton);
13     connect(settingsAction, SIGNAL(triggered()), this, SLOT(settings()));
14     return toolBar;
15 }
```

Figure 37: The createToolBar() function. This function sets up the tool bar in the plugin and connects the actions to the appropriate slots.

10.3 Devices Tab

This section discusses the implementation of the Devices tab from figure 17a, which also enables the need to have requirement 3. The Devices tab is made with the function createDevTab(), which returns the composite widget making up the Devices tab. createDevTab() can be seen on figure 38. It should be noted that the procedure of setting up the widget is very similar to that of the EasyInsert constructor on figure 33, i.e. using QScrollArea as the parent of a composite widget. Thus only the composite child will be discussed. The composite child is created on line 5 and then a vertical layout is created. On line 7 a QListView is created, this is a widget that provides a list view onto a model. Afterwards a QFileSystemModel is created and put onto the list view. The QFileSystemModel provides a data model for the local file system. Following, the _settingsMap with the Devices library path is read, and then used to set the root path on the model with the QFileSystemModel public function setRootPath(const QString &newpath). This actually installs a QFileSystemWatcher to monitor the path and update the QFileSystemModel accordingly. The view of the QListView's root index is then set to the model's root path, so as the content of the QFilSystemModel is shown in the list. The load button is then created and connected to the loadDevice() slot which creates the dialog window shown on figure 18b. Finally the widgets, the view and the button, are added to the layout and the composite widget gets the layout set. The composite widget has now been made, so the QScrollArea is then imposed onto the composite widget and returned.

```
QWidget* EasyInsert::createDevTab(){
    QScrollArea *widg = new QScrollArea(); //the parent
    widg->setWidgetResizable(true); //children will scale
    widg->setFrameShape(QFrame::NoFrame); //no frame
    QWidget *devTab = new QWidget(); //container widget
    QVBoxLayout *verticalLayout = new QVBoxLayout(devTab);
    view = new QListView(devTab); //list view widget
    dirmodel = new QFileSystemModel(view); //filesystem model
    view->setModel(dirmodel); //set the model
    view->setRootIndex(dirmodel->setRootPath(QString::fromStdString(
        _settingsMap->get("Devices", "/")))); // set the root
    QPushButton *loadBtn = new QPushButton("Load",devTab); //make button
    connect(loadBtn, SIGNAL(clicked()), this, SLOT(loadDevice()));
    verticalLayout->addWidget(view); // add to layout
    verticalLayout->addWidget(loadBtn);
    devTab->setLayout(verticalLayout); //set the layout
    widg->setWidget(devTab); // set the widget
    return widg;
}
```

Figure 38: The createDevTab() slot. The function creates the widget in the tab from figure 17a

10.4 Geometries Tab

This section discusses the implementation of the Geometries tab from figure 17b, which enables the need to have requirements 1, 2 and 2.1. The Geometries tab is made with the function createGeoTab() and it uses a similar procedure as the createDevTab(). The differences between the Devices tab and the Geometries tab, are the children of the composite widget, a QGridLayout, and that the QScrollArea is not set to be resizeable. The reason for different children of the composite widget is quite obvious (this tab should show something else), and a QGridLayout was chosen because it is a convenient layout to present the child widgets in. The resizeable option of the QScrollArea is not used because of aesthetic reasons. Since, if the resizeable option was on, the child widgets could either be scaled to ridiculous sizes or, if the childs widgets have fixed sizes, have a lot of spacing.

The children of the composite widget are eight QToolButtons. QToolButtons were used instead of QPushButtons, because even if QToolButtons are more sophisticated, they are generally preferred when the button is used as an icon. Each button has an intuitive icon and size set, and a tool tip to reflect the action. The buttons are then connected to their appropriate slot function and added to the layout. The slot functions are respectively **1.** cube() **2.** cylinder() **3.** cone() **4.** plane() **5.** sphere() **6.** tube() **7.** fixedFrame() **8.** movableFrame(). Likewise the composite widget then gets the layout set, and the QScrollArea is imposed onto the composite widget and returned.

10.5 Delete Tab

This section discusses the implementation of the Delete tab from figure 17c, which fulfils the need to have requirement 4. The Delete tab is made with the function createDeleteTab() and it uses a similar procedure as the createDevTab() function. The only difference between the Devices tab and the Delete tab are the children of the composite widget.

The composite widget of the Delete tab has three composite children in a vertical layout. Each composite child is a QGroupBox, one for Devices, Objects and Frames with each

their own QGridLayout. They all have a QListWidget their layout. The QListWidget is similar to the QListview used in the Device tab, but whereas QListWidget uses a more classic item-based interface for adding and removing items, it lacks in flexibility. However the QListWidget was still preferred because of it's more straight forward use. It should be noted that only the names of the devices, objects and frames will be saved in the list and not pointers to the actual objects. The Devices view and the Objects view each have a QPushButton added to their layout, while the Frames view has a composite widget with two QLabels and two QPushButtons in a vertical layout added to the QGridLayout of the Frames QGroupBox composite widget. The buttons are then connected to their proper delete slot. Respectively **1.** deleteDev() **2.** deleteObj() **3.** deleteSingleFrame() **4.** deleteFrame(). The three QGroupBoxes are then added to the vertical layout of the Delete tab composted widget and the layout is set. The QScrollArea is then imposed onto the composited widget and returned.

10.6 The Slots

In this section the slots of EasyInsert will be discussed, though the stateChangedListener() slot is left out to be discussed in section 10.7.

The Devices tab has only one button connected to a slot, loadDevice(). This slot is pretty straight forward and is very similar to the slots that the Geometries tab buttons are connected to. loadDevice() takes advantage of the dialog class to create an interactable dialog window (discussed in section 9.1) so the user can define parameters such as the need to have requirements 3.2 and 3.3, which are then passed to the loader class to load the desired device (discussed in section 8.2).

All the slots that the buttons from the Geometries tab are connected to are heavily boilerplated code. They all take advantage of the dialog class to construct an interactable dialog window (discussed in section 9.1) so the user can define parameters such as need to have requirements 1.1, 1.2 and 2.2 which are then passed to the creator class to load the desired geometry (discussed in section 7.2).

The slots that the Delete tab buttons are connected to are all delete functionalities. They all make use of the remove function provided by the WorkCell. deleteDev() and deleteObj() are very similar in structure, they both check if anything has been selected in their list and act if so. They use the name of the device/object selected in the list to find the device/object with respectively findDevice(const std::string &name) and findObject(const std::string &name), from the WorkCell, and then deletes them with remove(rw::common::Ptr<Device> device) and remove(rw::common::Ptr<Object> object), also from the WorkCell. deleteSingleFrame() and deleteFrame() however are more tricky, since frames can have children and it was decided that when the user removes a frame, all associated objects also should be deleted as well.

deleteSingleFrame() is shown on figure 39, as seen, some swapping of the WorkCell of RobWorkStudio is happening on line 6, 8, 9, 15 and 23, this will be explained in section 10.8, because it is in fact a recurring thing that is done in all the slots. The deleteSingleFrame() function first checks if anything has been selected on the Frame view, and if not it returns. If a frame was selected, the function then finds a pointer to the actual frame and then checks whether that frame has any children with the getChildren(const State &state) function. If it has any children the WorkCell is swapped back, RobWork then throws a warning and the deletion is then ignored. Though if the frame had no children, all the

objects associated with the frame will be removed with the removeAllOjects(...) function on line 16. Finally the frame is checked if it is the WORLD frame and removed if it is not, the WorkCell is also swapped back.

```
1  void EasyInsert::deleteSingleFrame()
2  {
3      if (_frameWidget->currentItem() == NULL)
4          return;
5      QListWidgetItem* item = _frameWidget->currentItem();
6      rw::models::WorkCell::Ptr wc = getRobWorkStudio()->getWorkCell();
7      rw::kinematics::Frame* frame = wc->findFrame(item->text().toStdString()
           );
8      rw::models::WorkCell::Ptr dummy = rw::common::ownedPtr(new rw::models::
           WorkCell("dummy")); // Create dummy wc for swap
9      getRobWorkStudio()->setWorkCell(dummy); // Temporarily swap out wc from
            rws
10     rw::kinematics::Frame::iterator_pair iter = frame->getChildren(wc->
           getDefaultState());
11     if( iter.first!=iter.second ){
12         getRobWorkStudio()->setWorkCell(wc); // Swap back wc into rws
13         RW_THROW("Frame has staticly connected children and therefore
               cannot be removed from tree!");
14         return;
15         }
16     removeAllObjects(frame,wc);
17     if(frame->getName() != "WORLD")
18         wc->remove(frame);
19     getRobWorkStudio()->setWorkCell(wc); // Swap back wc into rws
20 }
```

Figure 39: This figure shows the deleteSingleFrame() slot which is called from the Delete tab when deleting frames. This slot will delete a selected frame and all associated objects.

deleteFrame() has a similar structure to deleteDev() and deleteObj() up to the point when it calls the recursive deleteChildren(...) function. deleteChildren(...), see figure 40, works as a depth-first search method, i.e. the code traverses along a branch in the parent/child hierarchy for as long as possible before backtracking. The base case of the recursion is seen on line 12 when a frame has no children. The children are found with the getChildren(...) function, which returns a iterator pair for the children of that frame. This iterator pair is then run though a loop and the children stored are in a vector. When a frame is childless, the objects are removed and the frame deleted as with the deleteSingleFrame(). Of course if the WORLD frame is selected the deletion is ignored.

10.7 The StateChangedListener

The stateChangedListener() is a special slot which is subscribed to the stateChangedEvent() in RobWorkStudio. That means, whenever a change of state in a WorkCell occurs, the stateChangedEvent() emits a signal that the stateChangedListener() listens after and can react to.

The stateChangedListener() is set up to listen for events from the stateChangedEvent() under the initialize() virtual function, see figure 41. The stateChangedListener() slot is then called when ever a change occurs. The slot calls the update() function, which in turn

```
void EasyInsert::deleteChildren(rw::kinematics::Frame* frame, rw::models::
    WorkCell::Ptr wc)
{
    if (frame->getName() == "WORLD")
        return;
    rw::kinematics::Frame::iterator_pair iter = frame->getChildren(wc->
        getDefaultState());
    std::vector<rw::kinematics::Frame*> children;
    if ( iter.first!=iter.second){
        for (; iter.first!=iter.second; iter.first++) {
            rw::kinematics::Frame* child = &(*(iter.first));
            children.push_back(child);
        }
        for (size_t i = 0; i < children.size(); i++)
            deleteChildren(children[i],wc);
    }
    removeAllObjects(frame,wc);
    wc->remove(frame);
    }
}
```

Figure 40: The recursive deleteChildren(...) function. This function will also remove all objects associated with the frames deleted.

```
void EasyInsert::initialize()
{
    getRobWorkStudio()->stateChangedEvent().add(boost::bind(&EasyInsert::
        stateChangedListener, this, _1), this);
}
```

Figure 41: During plugin initialization the stateChangedListener() slot will be connected to the stateChangedEvent().

makes sure that appropriate elements of the plug in is updated. Currently only the lists of the Delete tab is updated through the function showFrameStructure(). showFrameStructure() clears all the elements in the lists of the Delete tab and also the vectors containing the actual object pointers with the clearListContent() function. showFrameStructure() will the proceed to find all the devices and store the pointers to them in a vector. These devices are then checked whether they are a serial devices or a tree devices, since these are the only devices supported by the plug in. When the device has been identified, the device name is added to the device list of the Delete tab and all the frames of the devices are stored in a vector to be used later, since the Objects and Frames list are not interested in showing device frames. showFrameStructure() will then go on to find frames not part of any device by first getting all frames from the WorkCell and storing them in a vector. The vector of device frames is then compared to all the frames in the WorkCell, and if any frame is found to be part of a device it is simply removed from the frames vector until only non device frames are left in that vector. The names of the remaining frames are then simply added to the Frames view of the Delete tab. Finally a vector of all the objects of the WorkCell will be compared to the objects of the device frame vector. If any objects are found to be part of a device, these objects will be removed from the vector of objects. The names of the objects left in the vector will then be saved in the Objects view from

the Delete tab.

10.8 Problems

This section will go through problems regarding the implementation of the plugin. Some of the problems have been temporarily solved, while suggestions for solving others or the root of the problems are given as well.

As noted in section 10.6, when inserting and deleting both devices and geometries, the WorkCell of RobWorkStudio will temporally be swapped with a dummy WorkCell while working on the WorkCell. If this was not done, then RobWorkStudio would seem to duplicate seemingly random elements of the insertions and in general behave unwanted when inserting devices or geometries. This is most likely due to state update problems of RobWorkStudio, because if the WorkCell was inspected, disregarding RobWorkStudio, there seemed to be no problem at all. In fact, if no swapping was done upon inserting, a state change after the insertion would update the WorkCell of RobWorkStudio properly, indicating problems with the update mechanism of RobWorkStudio. Swapping the WorkCell out of RobWorkStudio when inserting was deemed to be a fine solution to the problem, since it is a pretty cheap action. Though the problem should be inspected more in depth to improve the update mechanism of RobWorkStudio, this of course is out of scope for this project.

If geometries are created with the same base frame, the objects view will show the name of that frame, because objects are named after the base frame. This will in turn cause problems when deleting objects with the same base frame, because they can't be identified easily. This means no matter which of the objects are chosen for deletion (they all have the same name), the first object created will always be deleted and so forth. A solution to this problem could be to use QMetaType [11] to declare a custom type compatible with QVariant [12] of the Object class from RobWorkStudio. Using QVariant to store a copy of the Object will then allow one to set the data of a QListWidgetItem to that QVariant value. A user would then be able to, though a tool tip, when selecting a item from the Object view see all the contents (e.g. models and geometries) of that object. The deleteObj() slot would then have to be extended to check whether the object it is deleting contains the same contents.

As mentioned in section 6, there are problems with the delete functionalities. Deleting a device is working fine, but when trying to delete the frames after the device property has been removed may causes errors. E.g. after removing the device property of a inserted FANUC LRM200 like the one on figure 3, removing the frames then causes a segmentation fault, whereas doing the same for a UR-6-85-5-A arm is no problem at all. It would then draw suspicion toward the removeFrame(...) function from the RobWork library having problems with the frame type. However, the segmentation fault was actually tracked to happen when the WorkCell was swapped back into RobWorkStudio with the setWorkCell(...) function, and weirdly enough, if e.g. a moveable frame is present in the WorkCell while trying to delete the FANUC LRM200 frames, then no segmentation fault happens. This clearly suggests that somehow the setWorkCell(...) causes the error.

10.9 Future of the Plugin

Further work on the user interface would revolve around solving the problems discussed in 10.8 and generally improving the GUI, while still keeping the GUI intuitive. The settings of the plugin however would also require a revisit, since RobWorkStudio actually has a

settings file. This means that settings of the plugin should be reworked to add its settings to the RobWorkStudio settings file instead of creating a new file. Also a possible extension to add to the plugin would be a feature such as an Undo button, from the nice to have requirements 3, for undoing an insertion/deletion.

11 Discussion and Conclusion of the project

There are still errors and improvements to the solution, as mentioned in the chapters 7.4, 8.5 and 10.9. However testing the solution ourself, we were able to prove that the solution uphold the need to have requirements.

The need to have requirement 1 and 2 was satisfied with through the user interface and the creator, which proved capable of creating frames and geometries with the necessary parameters given in the requirement. The need to have requirement 3 was satisfied with through the loader and the user interface, which was capable of loading devices from a XML file. The need to have requirement 4 was satisfied with through the part of the solution described in section 10.5. Look at the attached USB for a video showcasing/testing the solutions regarding the need to have requirements.

The solution made should be regarded as a proof of concept since it is not optimized in either speed or usability. In order to optimize for speed, a better understanding of coding C++ and the RobWork library is required. If better usability is to be achieved, having users test the solution and getting feedback regarding the usability would be optimal. The reason this testing phase was not done in this project is because of the time restraint of the project. In case of more time, this would be one of the things prioritised, since the core functionalities are somewhat in place.

In terms of the nice to have requirements, it is our understanding that the user interface is intuitive to use. The functionality for the user to define a library for the devices were also made. Albeit a simple solution, its simplicity gives the solution, for defining a library, modularity. The nice to have requirement 3, regarding an undo action, was not implemented due to the time constraint imposed on the project. It was also discovered that this functionality would be rather difficult to implement.

All in all, the project was considered a success.

12 References

[1] SDURobotics. *The RobWork project webpage*. URL: http://www.robwork.dk (visited on 17/05/2016).

[2] Lars-Peter Ellekilde and Jimmy A. Jorgensen. 'RobWork: A Flexible Toolbox for Robotics Research and Education'. In: *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)* (June 2010).

[3] SDURobotics. *Figure showing the overview of the RobWork project*. URL: http://www.robwork.dk/apidoc/nightly/rw/overview-640.png (visited on 17/05/2016).

[4] Wikipedia. *Figure showing the RPY values of a plane*. URL: https://en.wikipedia.org/wiki/Aircraft_principal_axes (visited on 19/05/2016).

[5] The Qt Company. *Qt Documentation*. URL: http://doc.qt.io/ (visited on 07/05/2016).

[6] The Qt Company. *Qt Documentation, Writing Source Code for Translation*. URL: http://doc.qt.io/qt-5/i18n-source-translation.html (visited on 13/05/2016).

[7] Boost C++ Libraries. *Class path*. URL: http://www.boost.org/doc/libs/1_53_0/libs/filesystem/doc/reference.html#class-path (visited on 14/05/2016).

[8] RobWork. *XMLPropertyLoader Class Reference*. URL: http://www.robwork.dk/apidoc/nightly/rw/classrw_1_1loaders_1_1XMLPropertyLoader.html (visited on 14/05/2016).

[9] The Qt Company. *QFileDialog Class*. URL: http://doc.qt.io/qt-5/qfiledialog.html (visited on 15/05/2016).

[10] RobWork. *XMLPropertySaver Class Reference*. URL: http://www.robwork.dk/apidoc/nightly/rw/classrw_1_1loaders_1_1XMLPropertySaver.html (visited on 15/05/2016).

[11] The Qt Company. *QMetaType Class*. URL: http://doc.qt.io/qt-5/qmetatype.html#Q_DECLARE_METATYPE (visited on 18/05/2016).

[12] The Qt Company. *QVariant Class*. URL: http://doc.qt.io/qt-5/qvariant.html#setValue (visited on 18/05/2016).

Appendix A Interview template

RobWork EasyInsert-plugin Survey

Name:

Title:

Occupation:

Questions:

1. **Relationship regarding RW:**
2. **Do you recognize the problem that EasyInsert seeks to resolve?**
3. **How do you expect to use the plugin?**
    a. **Access to the loadable objects? (browser, pop-up.. etc.) (draw)**
        i. **layout for the browser? (in detail)**
    b. **Drag and drop, load button.. etc?**
    c. **Selection description? (images, text.. etc.)**
    d. **Extending the library? (if one)**
        i. **Standard objects?**
4. **Remove objects from the workcell?**
    a. **Undo button?**

Changing the configurations of the simulated robot in an intuitive matter, involving cursor interaction pulling the robot into the desired position.

Adding tools or end effector to a simulated robot in a drag and drop-like feature. The tool should then be able to snap onto various parts of the robot.

Appendix B Use cases

B.1 Use cases for current use

Adding a frame/geometry to a WorkCell
Main Success Scenario: The user opens the XML-file describing the WorkCell in an editor. The user then writes the appropriate tag for adding the frame/geometry. The user then writes the appropriate tags for adding the additional information for the frame/geometry. The user then saves the WorkCell XML-file. The user then swaps to/ starts up RobWorkStudio. The user then loads the WorkCell from the XML-file via open in RobWorkStudio.

Alternate Scenarios:
The user makes an error, the loader or parser catches this error and stops the loading process. The user is informed of the error.

Adding a device to a WorkCell
Main Success Scenario: The user opens the XML-file describing the WorkCell in an editor. The user uses the include tag to refer to the description of the device from another XML-file. The user then saves the WorkCell XML-file. The user then swaps to/ starts up RobWorkStudio. The user then loads the WorkCell from the XML-file via open in RobWorkStudio.

Alternate Scenarios:
The user manually writes the description of the device.

The user makes an error, the loader or parser catches this error and stops the loading process. The user is informed of the error.

Deleting an element
Main Success Scenario: The user opens the XML-file describing the WorkCell in an editor. The user deletes the tag that resembles the part the user wishes to delete. The user then saves the WorkCell XML-file. The user then swaps to/ starts up RobWorkStudio. The user then loads the WorkCell from the XML-file via open in RobWorkStudio.

Alternate Scenarios:
The user makes an error, the loader or parser catches this error and stops the loading process. The user is informed of the error.

B.2 Use cases for solution

Adding a frame/geometry to a WorkCell
Main Success Scenario: The user selects the wanted type of frame/geometry. The user then inputs the required information for the selected type of frame/geometry. The frame/geometry is then created and inserted into the WorkCell.

Alternate Scenarios:
If the user supplied invalid information, the user is informed of the invalid information.

Adding a device to a WorkCell
Main Success Scenario: The user supplies the description of the device. The user then supplies the information about where the device should be placed and possibly a transform. The user also gives the device a name.

Alternate Scenarios:
If the user gave the device a name that is already in use, the adding process is stopped and the user is informed of this error.

If the user supplied invalid information, the user is informed of the invalid information.

Deleting an element *Main Success Scenario:* The user selects the element that the user wishes to delete. The user then initialises the deletion process.

Alternate Scenarios:
If the user supplied invalid information, the user is informed of the invalid information.