Internal Report 2012-2013–14 August 2013

Universiteit Leiden
Opleiding Informatica

Testing of
channel based
service connectors

Joost Leuven

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

1. Introduction

Nowadays most applications (software) are build as individual components that are combined later on in the project. This keeps the development easy and clear. However this presents a whole new set of problems. Since these components need to work together to create the complete application, it is paramount that the communication between elements is well structured and well organized. It is for this purpose that REO is incredibly useful. REO offers a standardized method to model the interconnection of software components. Before the linking of individual components was done using what's called "glue code". This glue code was very messy and inconvenient. Another problem with "glue code" is that it's not always clear if that piece of coding does what it's supposed to do. REO doesn't have these problems, because REO can be tested for input/output conformance. The current REO framework within Eclipse doesn't support automated input/output conformance checking for the REO-connectors. Every input/output conformance check needs to be done by hand. In this paper I will show how I've implemented an automated process for the checking of input/output conformance of 2 REO-connectors.

2. A practical overview of REO

In this chapter an overview of REO connectors is given as discribed in [1]

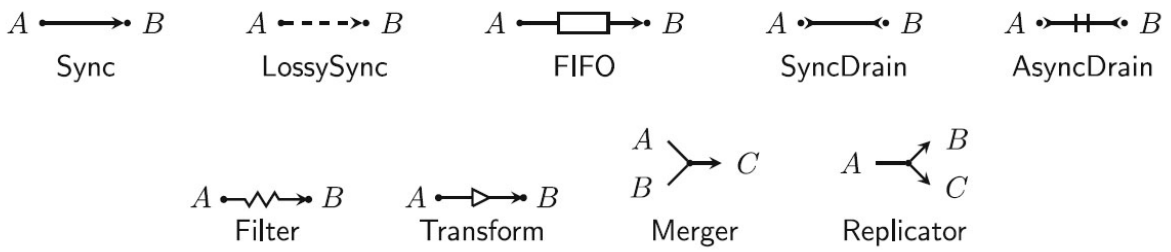

Figure 2.1: The basic REO-connectors

2.1 Channels

The nodes in a Reo connector are connected via channels. In this section we deal with the meaning of the channels as displayed above. Every channel has a source and a sink end. Following is a discription of what heretofore mentioned channels do with their source and sink ends. The Sync channel accepts a data item only if it can instantly output it on the sink end. LossySync always accepts a data item and tries to unload this on it's sink end. If this is not possible the data item is lost. The FIFO channel accepts a data item on its source end and can act as a buffer with capacity 1. SyncDrain has 2 source ends. It accepts data items from both it's inputs at the same time and loses these. AsyncDrain only accepts data item from 1 of its source ends and lose it. The following 2 channels are for data manipulation. First the filter is used to accept a data item from its source end and only puts the data item on the sink end of the channel if the data item is of a specific type. Second the Transform channel is used to transform the data item on the source end into an other data item using a user-defined function. All these channels can be combined with each other by using nodes.

2.2 Nodes

All nodes can be one of 3 types. Source, Sink and mixed. What type they belong to depends on what kind of ends of the connected channels are connected to the node. If all connected ends are source ends it is a sink node and vice versa. All source and sink nodes combined form the boundary of the Reo-connector and are used form communication with its environment.

3. Input/Output conformance (ioco) and mCRL2

3.1 mCRL2

mCRL2 is a process algebra that can be used to describe the semantics of a REO connector. A description of a REO connector in mCRL2 is formed by a set of actions. These actions describe anatomic events. There are several operators used to combine actions into multiactions. First of all actions can be synchronized. This is done with the | operator. So $a|b$ means that action a occurs simultaneously with action b. This action is commutative, so $a|b$ is equivalent to $b|a$. Another operator is the δ operator. This operator doesn't display any behavior. A third operator is the + operator which defined a choice between two actions. So $a + b$ means that either action a is done or action b. Fourthly there's the sequential composition $\cdot$. So $a \cdot b$ means that a is followed by b. Another operator is the if-then-else operator. This operator is of the form $c \rightarrow a \diamond b$. This means that if condition c evaluates to true action a happens and otherwise action b. There are some more operators which can be found in [2].

3.2 JTorX

JTorX is a program that can, given a specification and an implementation (both in Aldebaran format *.aut*), compare these two for ioco. For this project JTorX was connected to the REO extension for Eclipse.

3.3 ioco

In order to test whether what you've build is actually what you wanted to build when you specified the specifications, we need to have both reo model of the specification and implementation. In [2] it is explained that if we add extra actions to the mCRL2 definition of a REO connector for the boundary nodes, we can test whether two REO models are input output equivalent. For every boundary node (and thus action) A we need to add an action $?A$ and $!A$. The action $?A$ describes when the environment requests input or output from the node A. $!A$ describes the actual observation of data flow over node A. This changes the view mCRL2 gives on the semantics of a REO connector. Normally we couldn't distinguish between data being rejected and data being accepted but being lost in the connector. Both have actions on the input boundary node but neither has action on the output boundary node. But the REO connectors are not the same! With the addition of the ! and ? actions we can observe that if data is entered into the REO connector through an input boundary node A that the action $!A$ occurs. The same goes for output boundary nodes. If an output boundary node B has output ($!B$), then the action $?B$ must also have occured. Another benefit of this approach is that we don't need to know what happens between A and B.

3.1 and 3.2 is are excelent examples of how this new approach works. If input is observed on A then in the specification, flow on B and C could be observed simultaneously or indepentant of eachother. But in the implementation flow over B and C is always observed simultaneously and never independant (because of the extra sync channel). If we describe this using the ! and ? actions we get that if the action $?A$ and $?C$ have been observed, in the specification $!A$ and $!C$ can be observed whereas in the implementation $!C$ will never occur.

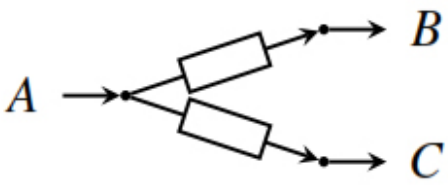

Figure 3.1: Specification

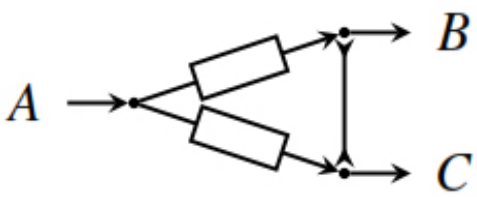

Figure 3.2: Implementation

4. Implementation

4.1 Eclipse

Eclipse is a cross-platform developement toolkit. With Eclipse it's possible to build extensions for Eclipse itself. In the appendices of this report extensive instructions can be found on how to install the various plugins and/or additional software needed to both run the developed REO Extension and how to start the developement environment to start modifying the REO extension yourself. Once all these instructions have been followed, the REO extension can be build and run. When you run your project, Eclipse will start a new version of itself with your home build plug-in as one of its plugins.

4.2 REO extention code modifications

In order to implement the in chapter 3 mentioned implementation of Input/Output Conformance testing, a converter for the current mCRL2 implementation needed to be made. This converter uses the converter build for the I/O actions conversion of the mCRL2 code as a basis [4]. Most of the mCRL2 code that is modified in this converter is not needed for the ioco conversion. For all these non needed conversions, the implementation of the basic converter is used. The second modification made to the mCRL2 plug-in for Eclipse is the addition of some functionality buttons. With these buttons the current text of the specification window can be saved to a *.aut* file. The second button also saved the current specification window text to a dot out file. But because the mCRL2 converter isn't able to handle ! and ? characters, the input for the mCRL2 converter is done using i and o instead. When the second button is clicked the same *.aut* file is saved as with the first button, but after this the output is converted to have the ! and ? format. The third button is to start JTorX. JTorX is a program that used 2 descriptions of REO models (the *.aut* files) and does the testing for Input/Output Conformance. In order to be able to start JTorX, it's location on the users harddrive must be specified. The same goes for mCRL2 (see Appendix D). A text field in the settings was added to accommodate this.

4.3 Results

In this chapter some results will be shown. In 4.1 we see the mCRL2 interface tab as seen in the REO-extension before any modifcations were made to the source code. In 4.2 the new interface is shown. A new checkbox has been added, as well as 3 new buttons at the button.

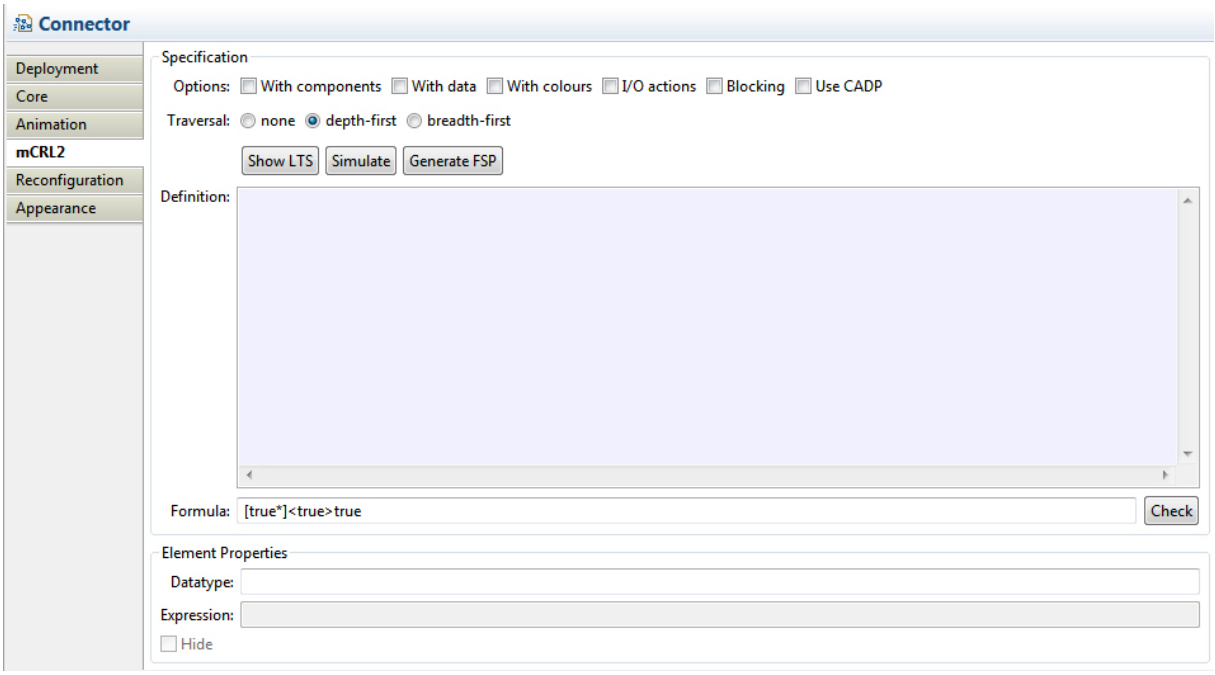

Figure 4.1: User Interface before modifications

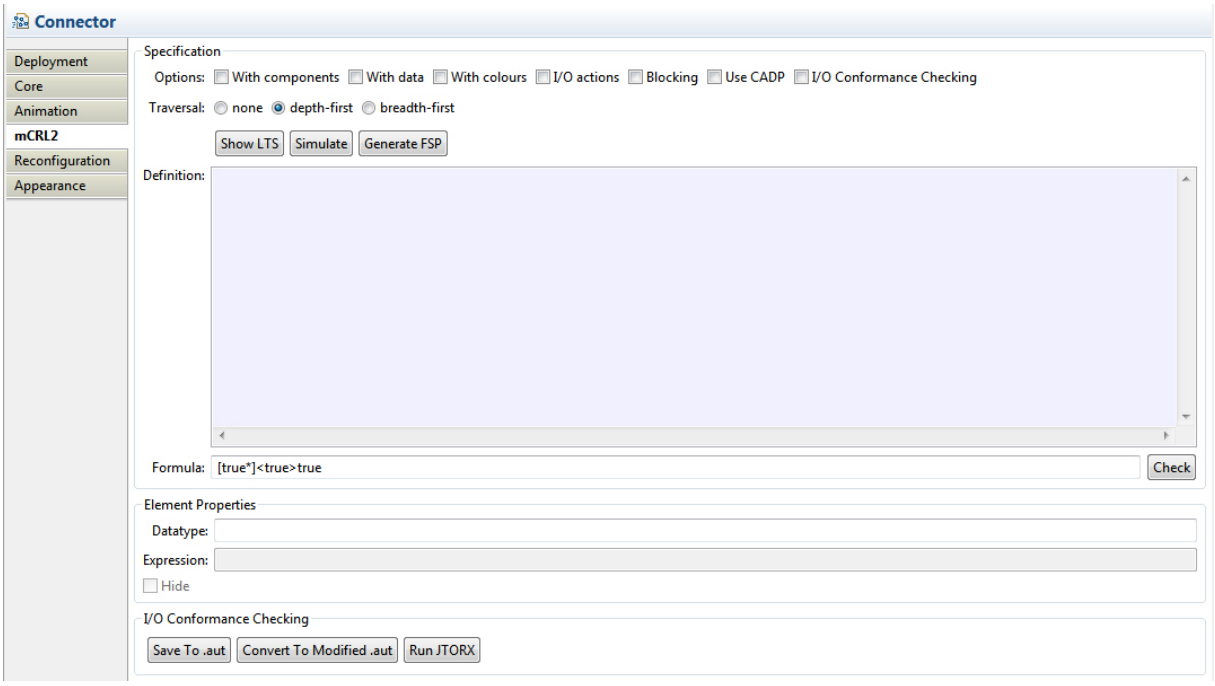

Figure 4.2: User Interface after modifications

The following page contains the mCRL2 beginings ot the definition of 3.1. On the left we have the begin of the mCRL2 definition without the input and output events on the boundary nodes. On the right we have the modified mCRL2 definition as described in section 4.2.

act
$A, B, C, M, M', M'', N, N', N'', O, O',$
$O'', P, P', P'', Q, Q', Q'', R, R', R'', S, S',$
$S'', T, T', T'';$

act
$M, M', M'', N, N', N'', O, O', O'', P, P',$
$P'', Q, Q', Q'', R, R', R'', S, S', S'', T, T',$
$T'', iA, iB, iC, oA, oB, oC;$

$proc$
$FIFO1 = M''.N''.FIFO1;$
$FIFO2 = O''.P''.FIFO2;$
$Sync3 = (Q''|R'').Sync3;$
$Sync4 = (S''|T'').Sync4;$
$Node1 = ((A|M'|O')).Node1;$
$Node2 = ((N'|S')).Node2;$
$Node3 = ((P'|Q')).Node3;$
$Node4 = ((T'|B)).Node4;$
$Node5 = ((R'|C)).Node5;$

$proc$
$FIFO1 = M''.N''.FIFO1;$
$FIFO2 = O''.P''.FIFO2;$
$Sync3 = (Q''|R'').Sync3;$
$Sync4 = (S''|T'').Sync4;$
$Node1 = ((iA.oA|M'|O')).Node1;$
$Node2 = ((N'|S')).Node2;$
$Node3 = ((P'|Q')).Node3;$
$Node4 = ((T'.iB|oB)).Node4;$
$Node5 = ((R'.iC|oC)).Node5;$

Below we have the same definitions as above, only now we have them for 3.2.

act
$A, B, C, M, M', M'', N, N', N'', O, O',$
$O'', P, P', P'', Q, Q', Q'', R, R', R'', S, S',$
$S'', T, T', T'', U, U', U'', V, V', V'';$

act
$M, M', M'', N, N', N'', O, O', O'', P, P',$
$P'', Q, Q', Q'', R, R', R'', S, S', S'', T, T',$
$T'', U, U', U'', V, V', V'', iA, iB, iC, oA, oB,$
$oC;$

$proc$
$FIFO1 = M''.N''.FIFO1;$
$FIFO2 = O''.P''.FIFO2;$
$Sync3 = (Q''|R'').Sync3;$
$Sync4 = (S''|T'').Sync4;$
$SyncDrain5 = (U''|V'').SyncDrain5;$
$Node1 = ((A|M'|O')).Node1;$
$Node2 = ((N'|S'|U')).Node2;$
$Node3 = ((P'|Q'|V')).Node3;$
$Node4 = ((T'|B)).Node4;$
$Node5 = ((R'|C)).Node5;$

$proc$
$FIFO1 = M''.N''.FIFO1;$
$FIFO2 = O''.P''.FIFO2;$
$Sync3 = (Q''|R'').Sync3;$
$Sync4 = (S''|T'').Sync4;$
$SyncDrain5 = (U''|V'').SyncDrain5;$
$Node1 = ((iA.oA|M'|O')).Node1;$
$Node2 = ((N'|S'|U')).Node2;$
$Node3 = ((P'|Q'|V')).Node3;$
$Node4 = ((T'.iB|oB)).Node4;$
$Node5 = ((R'.iC|oC)).Node5;$

The images displayed below are parhaps the most striking evidence that the ioco testing works. The first image is a graph representing 3.1 using the ioco setting in the reo-extension. The second image is a graph represention 3.2 using the same settings. On closer inspection of these graphs we find that in the implementation, there are some actions that don't occur in the specification graph. This points out that the specification 3.1 and 3.2 are not the same in relation to Input/Output Conformance. Example. After the action iA, oA and iB have been observed, in the specification oB can be observed. In the implementation this cannot happen before iC has been observed.

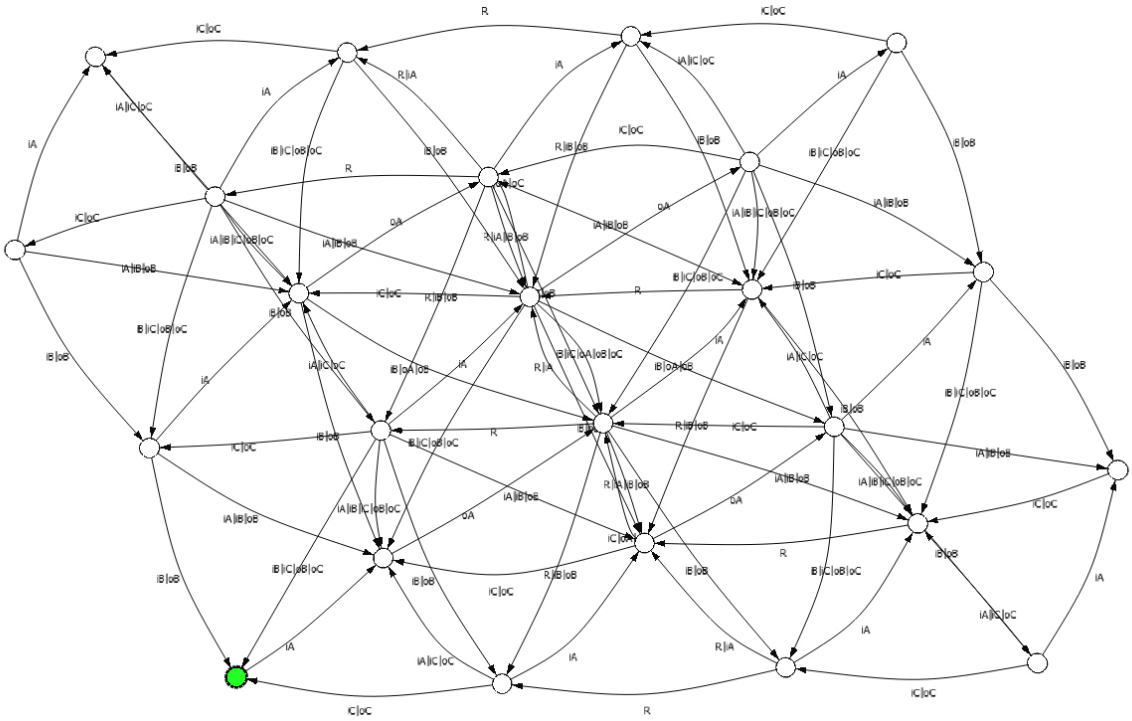

Figure 4.3: Graph representing 3.1 using the ioco mCRL2 definition

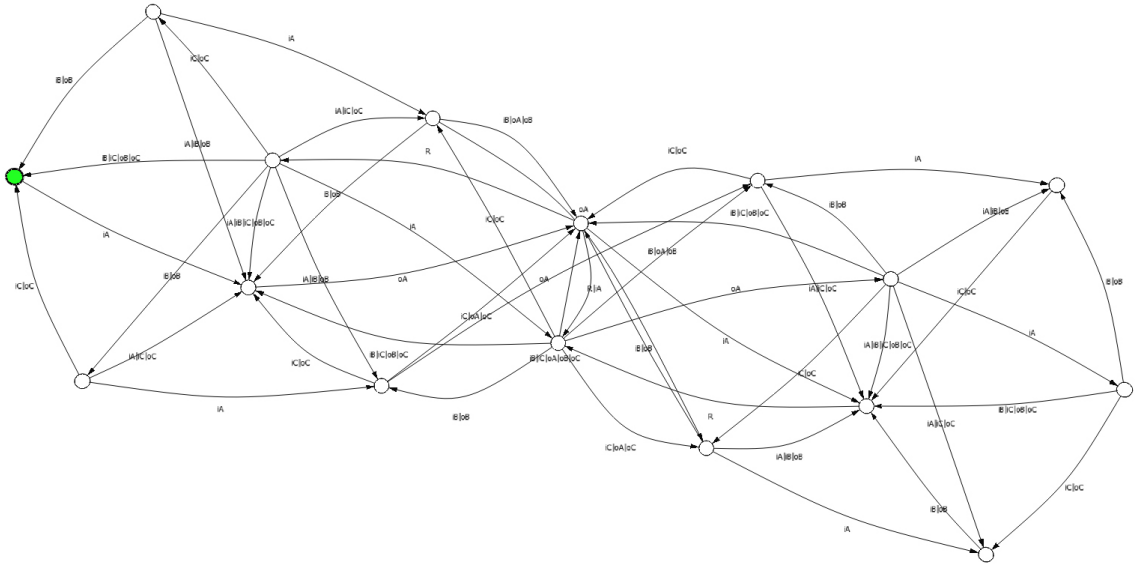

Figure 4.4: Graph representing 3.2 using the ioco mCRL2 definition

Below are the two graphs represention the same REO connectors. Though these graphs are not isomorphic, we can see on closer inspection that every action in the implementation (only one NB. $A|B|C$) occurs in the specification. This is where the ioco and non-ioco graphs differ.

Figure 4.5: Graph representing 3.1 using the mCRL2 definition without ioco

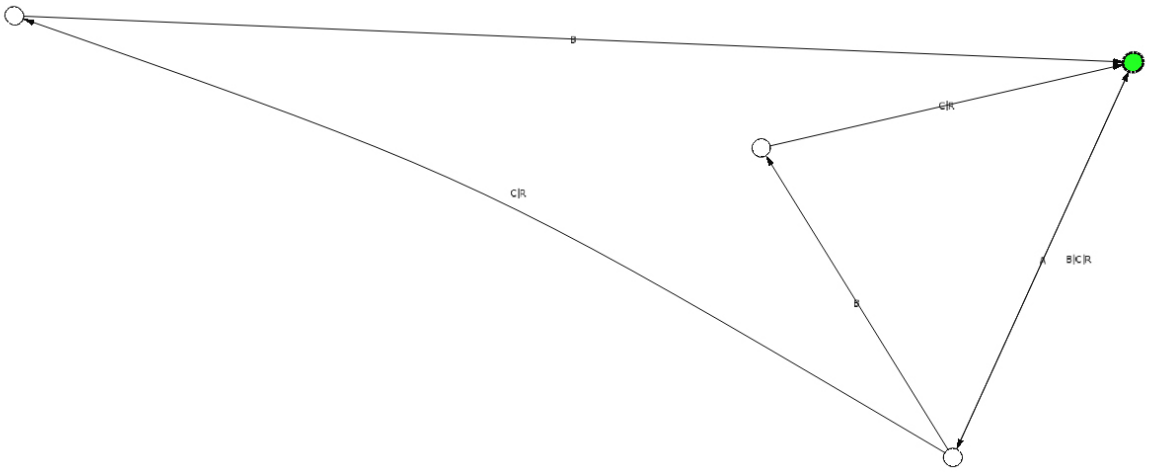

Figure 4.6: Graph representing 3.2 using the mCRL2 definition without ioco

4.4 Changelog

Following is a list of all modified files and a short description of the changes made

4.4.1 Changed Files

org.ect.reo.ui/src/org/ect/reo/prefs/ReoPreferenceConstants.java
Added JTorX home
org.ect.reo.ui/src/org/ect/reo/prefs/ui/ExternalProgramsPage.java
Added field for the JTorX home directory
org.ect.reo.mcrl2/src/org/ect/reo/mcrl2/Atom.java
Added function removeDouble
org.ect.reo.mcrl2/src/org/ect/reo/mcrl2/properties/MCRL2PropertySection.java
Added function addIOConfChk, added checkbox for ioco checking.
org.ect.reo.mcrl2/src/org/ect/reo/mcrl2/conversion/Reo2MCRL2Preferences.java
Added constants and functions to handle check/non-check of ioco checkbox
org.ect.reo.mcrl2/src/org/ect/reo/mcrl2/conversion/ElementConverters.java
Added new converter (Channel and component converter are basic, node converter is new)
org.ect.reo.ui/src/org/ect/reo/prefs/ReoPreferences.java
Added function to return JTORX home directory.
org.ect.reo.mcrl2/src/org/ect/reo/mcrl2/Sequence.java
Removed the additional brackets
org.ect.reo.mcrl2/src/org/ect/reo/mcrl2/Specification.java
Added call to function removeDouble

4.4.2 New Files

org.ect.reo.mcrl2/src/org/ect/reo/mcrl2/converters/IOCONodeConverter.java
This file implements the converter for the ioco conversion of the specification text field.
org.ect.reo.mcrl2/src/org/ect/reo/mcrl2/properties/SaveAUT.java
This file adds the job that needs to be run if the "Save To .aut" button is pressed
org.ect.reo.mcrl2/src/org/ect/reo/mcrl2/properties/SaveConvAUT.java
This file adds the job that needs to be run if the "Save To Modified .aut" button is pressed
org.ect.reo.mcrl2/src/org/ect/reo/mcrl2/properties/RunJTorX.java
This file adds the job that needs to be run if the "Run JTorX" button is pressed

A. Installing Eclipse

In order to work with the REO extention for Eclipse, one first needs to install Eclipse itself. A clean installation of Eclipse can be found at `http://www.eclipse.org/downloads`. To install Eclipse the only thing that has to be done is to extract the content of the downloaded zip-file. There's no need for an actual installation.

B. REO extention

To work with the REO extention in Eclipse it first needs to be installed. Following is a step by step description of how this is done.

1. Open Eclipse.
2. Go to `Help`→`Install New Software...`
3. In the `Work with:` field type `http://reo.project.cwi.nl/update`
4. Under `Extensible Coordination Tools` select the `Reo Core Tools (required)`
5. Click `Next`
6. Click `Next`
7. Accept the terms of the license agreement
8. Click `Finish`

C. REO extention developement

To start developement on the REO extension two things need to be done: First the source code for the REO extension is needed, and second, these files need to be imported into eclipse

C.1 Getting the source code

A clean version of the REO extension source code can be found at `http://code.google.com/p/extensible-coordination-tools/source/checkout`. In order to get this we need to use an SVN-application. For windows a good SVN program is TortoiseSVN. This can be found at `http://tortoisesvn.net`. Once you have downloaded the source code it can be imported into Eclipse.

C.2 Importing the source code

To import the source code follow the following steps:

1. Open Eclipse.
2. Go to `File`→`Import...`
3. Under `Plug-in Developement` select `Plug-ins and Fragments`
4. Under `Import From` select the directory where you've downloaded the source code files
5. Under `Import As` select `Projects with source folders`
6. Click `Next`
7. Add all REO plug-ins
8. Click `Finish`

C.3 Building the source code

In order to be able to run the source code we need to install some additional software in reo. Use the following steps to do this:

1. Open Eclipse.
2. Go to `Help`→`Install New Software...`
3. Under `Work with:` select the website for your current release of Eclipse (Juno (3.8) was used in this project).
4. Type GMF into the filter field and select all the packages that are filtered out.

5. Click `Next`
6. Click `Next`
7. Accept the terms of the license agreement
8. Click `Finish`
9. Do step 2-8 again but instead of typing GMF (step 4) type EMF.

Now the REO Extension for Eclipse should compile without fault.

C.4 Running the source code

In order to run the source code a run configuration needs to be made. The following steps will explain how this is done

1. Open Eclipse.
2. Go to `Run`→`Run Configurations...`
3. Under `Plug-ins` select the workspace plugin.
4. Click `Run`

D. Install mCRL2

mCRL2 can be downloaded from `http://www.mcrl2.org`. Once you have downloaded and installed this, it still needs to be connected to the REO extension in Eclipse. Use the Following steps to do this.

1. Open Eclipse.
2. Go to `Window`→`Preferences`
3. Select `Reo`→`External Programs`
4. Locate your installation of mCRL2 in the `mCRL2 home (not bin)` field.

Bibliography

[1] F. Arbab (2004): *Reo: A Channel-based Coordination Model for Component Composition*. Mathematical Structures in Computer Science 14, pp. 1-34, doi: 10.1017/S0960129504004153

[2] N. Kokash, F. Arbab, B. Changizi (2011): *Input-output Conformance Testing for Channel-based Service Connectors*. PACO

[3] J. Tretmans (2008): *Model Based Testing with Labelled Transition Systems*. In: Formal Methods and Testing, LNCS 4949, Springer, pp. 1-38, doi: 10.1007/978-3-540-78917-8_1

[4] N. Kokash, C. Krause, E.P. de Vink (2011):*REO + mCRL2: A Framework for Model-checking Dataflow in Service Compositions*. Formal Aspects of Computing doi: 10.1007/s00165-011-0191-6