TECHNICAL REPORT

ISO/IEC TR 23002-6

First edition
2017-10

Information technology — MPEG video technologies —

Part 6: Tools for reconfigurable media coding implementations

Technologies de l'information — Technologies vidéo MPEG —

Partie 6: Outils d'implémentation du codage média reconfigurable

Reference number
ISO/IEC TR 23002-6:2017(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

A list of all the parts in the ISO/IEC 23002 series can be found on the ISO website.

Introduction

This document provides a description of a set of tools that are intended to be helpful for developing reconfigurable media coding implementations based on ISO/IEC 23001-4, ISO/IEC 23002-4 and ISO/IEC 23002-5. The description includes the following guidelines:

— guidelines on good practices to implement specifications based on ISO/IEC 23001-4, ISO/IEC 23002-4 and ISO/IEC 23002-5;

— guidelines on usage of a monitoring tool for specifications based on ISO/IEC 23001-4, ISO/IEC 23002-4 and ISO/IEC 23002-5.

— guidelines on usage of a design exploration and optimization tool for specifications based on ISO/IEC 23001-4, ISO/IEC 23002-4 and ISO/IEC 23002-5.

Information technology — MPEG video technologies —

Part 6: Tools for reconfigurable media coding implementations

1 Scope

This document provides a description of a set of tools that are intended to be helpful for developing reconfigurable media coding implementations based on ISO/IEC 23001-4, ISO/IEC 23002-4 and ISO/IEC 23002-5.

2 Normative references

There are no normative references in this document.

3 Terms, definitions and abbreviated terms

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— IEC Electropedia: available at http://www.electropedia.org/

— ISO Online browsing platform: available at http://www.iso.org/obp

3.1 Terms and definitions

3.1.1
Reconfigurable Video Coding
RVC
framework defined to support coding standards at the tool level while maintaining interoperability between solutions from different implementers

3.1.2
functional unit
FU
modular tool characterized by its input/output behaviour, consisting of a processing unit

Note 1 to entry: Also referred to as “actor”.

3.1.3
token
data entity exchanged among *functional units* (3.1.2), such that a functional unit performs operations on input tokens, produces output tokens and modifies its state

3.1.4
connection
link from output ports to input ports of *functional units* (3.1.2) that enable *token* (3.1.3) exchange between the corresponding functional units

3.1.5
functional unit network language
FU network language
FNL
language that describes a network of *functional units* (3.1.2)

3.1.6
functional unit network description
FU network description
FND
functional unit (3.1.2) *connection* (3.1.4) used to build a decoder and modelled using the *functional unit network language* (3.1.5)

3.1.7
RVC-CAL
dataflow specification language for specifying of *functional units* (3.1.2) and the reference software

3.1.8
video tool library
VTL
collection of *functional units* (3.1.2)

3.1.9
MPEG video tool library
video tool library (3.1.8) that contains *functional units* (3.1.2) drawn from existing MPEG standards

3.2 Abbreviated terms

API	Application Programming Interface
BXDF	Buffer-size XML Dataflow Format
CAL	CAL Actor Language
DSE	Design Space Exploration
ETG	Execution Trace Graph
FIFO	First-In, First-Out
FSM	Finite State Machine
HEVC	High-Efficiency Video Coding
IDE	Integrated Development Environment
L1_DCM	L1 Data Cache Misses
NoL	Number of Lines
PAPI	Performance API
PMC	Performance Monitoring Counter
RMC	Reconfigurable Media Coding
TOT_INS	TOTal Number of INStructions
XDF	XML Dataflow Format
XML	eXtensible Markup Language

4 Overview

This document describes a set of tools that are intended to be helpful for developing reconfigurable media coding implementations. Clause 5 describes how to implement an RVC-CAL specification using the Open RVC-CAL Compiler (ORCC)[1] tool. Clause 6 details how to use Papify[2] and Papify Viewer[3]. Papify is an implementation of an event-based performance monitoring tool integrated into ORCC. Papify Viewer is a visualization tool to monitor the actions of actors in RVC-CAL specifications. Fired actions can be analysed chronologically from different points of view. Finally, Clause 7 describes how to use TURNUS[4], an open-source system design exploration and optimization framework especially tailored for CAL dataflow programs.

5 RVC-CAL

5.1 General

RVC-CAL is a standardized version of the CAL Actor Language which implements a dataflow model of computation.

All the examples in this document are written for and tested with the ORCC compiler infrastructure tools.

Open RVC-CAL Compiler (ORCC) is an open-source Integrated Development Environment (IDE) based on Eclipse and dedicated to dataflow programming. Eclipse[5] is a software package which provides an integrated development environment for Java/C/C++ programs. The primary purpose of ORCC is to provide developers with a compiler infrastructure to allow software/hardware code to be generated from dataflow descriptions.

5.2 Installing ORCC tools

First, all the tools needed for compiling and running the provided examples should be installed. The Open RVC-CAL Compiler (ORCC) requires the Eclipse platform. Therefore, in order to use it, it is necessary to install the Java™ Runtime Environment (JRE)[6] and Eclipse IDE.

5.2.1 Java Runtime Environment

ORCC requires JRE version 1.6 or higher. The latest JRE release is available at https://java.com/en/download/. To use the Eclipse IDE for Java Developers edition, it is necessary to also install the Java™

1) Open RVC-CAL Compiler (ORCC) is an open-source IDE based on Eclipse. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named. Equivalent products may be used if they can be shown to lead to the same results.

2) Papify is an open-source tool available on GitHub. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named. Equivalent products may be used if they can be shown to lead to the same results.

3) Papify Viewer is an open-source visualization tool available on GitHub. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named. Equivalent products may be used if they can be shown to lead to the same results.

4) TURNUS is an open-source system published by IEEE.org. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named. Equivalent products may be used if they can be shown to lead to the same results.

5) Eclipse is an open-source software package published by the Eclipse Foundation (www.eclipse.org). This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named. Equivalent products may be used if they can be shown to lead to the same results.

6) Java Runtime Environment (JRE) is a trademark of a product supplied by Oracle. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named. Equivalent products may be used if they can be shown to lead to the same results.

Development Kit (JDK)[7], which can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/index.html.

5.2.2 Eclipse

ORCC is compatible with Eclipse versions 4.3 and higher. The Eclipse IDE can be downloaded from https://www.eclipse.org/downloads/ (for which the Eclipse IDE for Java Developers edition is suggested). To install it, extract the archive into a local directory.

Eclipse should be configured to allocate at least 512 MB of memory for the heap. This can be done by adding the -Xmx512m parameter in the eclipse.ini file.

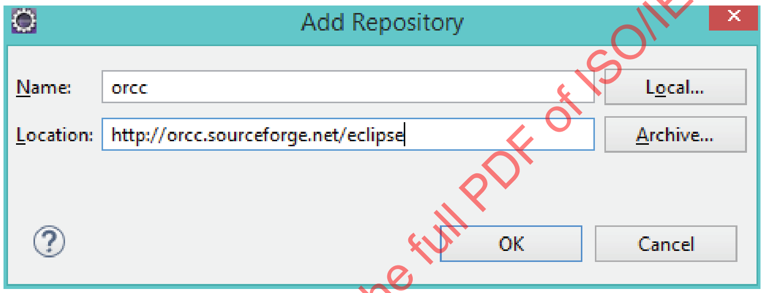
5.2.3 ORCC plug-in for Eclipse

The Eclipse Software Update Manager can be used to install ORCC. The steps are as follows.

— In Eclipse, go to Help > Install New Software.

— Click Add to add an update site.

— Set its name (e.g. ORCC) and its URL to http://orcc.sourceforge.net/eclipse (see Figure 1).



Name: orcc

Location: http://orcc.sourceforge.net/eclipse

Figure 1 — Repository setting during the installation of the ORCC plug-in for Eclipse

— Once this is done, select Open RVC-CAL Compiler or ORCC (see Figure 2).

7) Java Development Kit (JDK) is a trademark of a product supplied by Oracle. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named. Equivalent products may be used if they can be shown to lead to the same results.

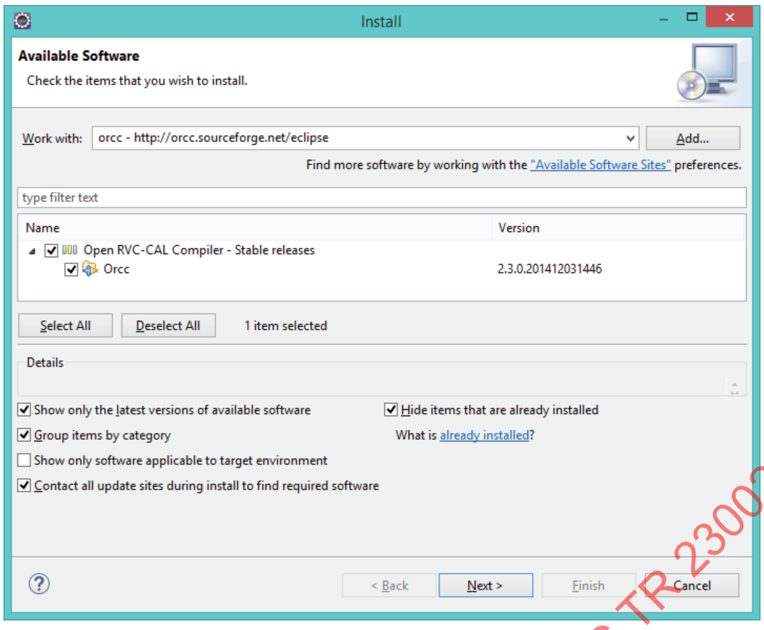

Install
Available Software
Check the items that you wish to install.
Work with: orcc - http://orcc.sourceforge.net/eclipse
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
Open RVC-CAL Compiler - Stable releases
Orcc
2.3.0.201412031446
Select All
Deselect All
1 item selected
Details
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
Finish
Cancel

Figure 2 — Selection of the Open RVC-CAL Compiler during the installation of the ORCC plug-in for Eclipse

— Click Next, check and accept the licenses and then click Finish.

— Eclipse will prompt for a confirmation to install an unverified feature. Accept and restart Eclipse.

5.3 "Hello world"

Make sure that the active perspective in Eclipse is Java™ EE[8]; otherwise, the menus will differ slightly and some menu items will not be in the illustrated places.

5.3.1 Creating a new project

A new ORCC project should be created. In the menu File > New > Other, select ORCC > Orcc Project (see Figure 3).

8) Java EE (JEE) is a trademark of a product supplied by Oracle. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named. Equivalent products may be used if they can be shown to lead to the same results.

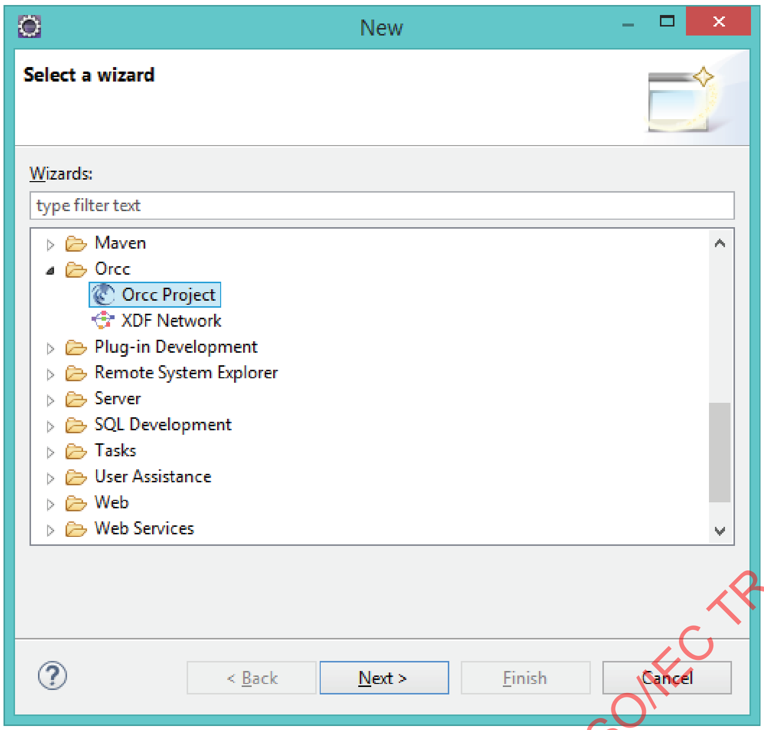
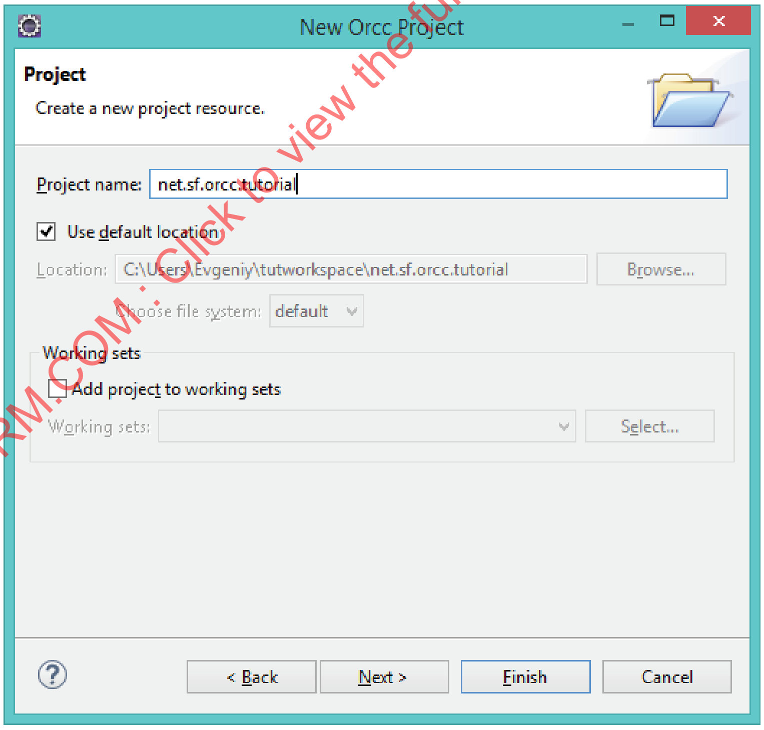


Figure 3 — Wizard selection to create a new project

Specify the name of the project and click Finish (see Figure 4).



Project name: net.sf.orcc.tutorial

[✓] Use default location

Location: C:\Users\Evgeniy\tutworkspace\net.sf.orcc.tutorial

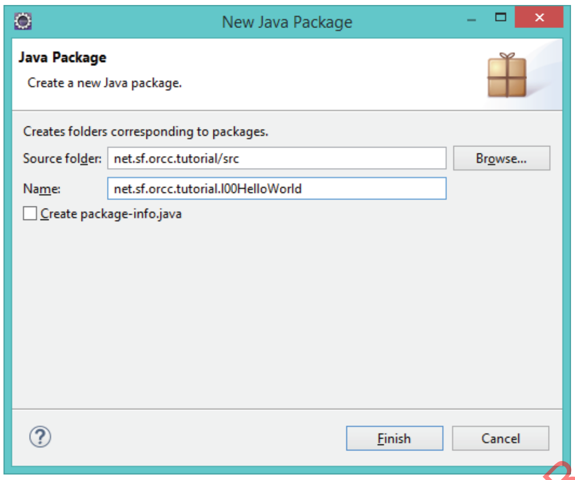
Choose file system: default

Figure 4 — Project name setting in ORCC

A default *src* directory will be added to the created project by this action.

5.3.2 Creating a new package

Right-click on the *src* folder in the *Project Explorer* pane. Select New > Package (see Figure 5).



Creates folders corresponding to packages.

Source folder: net.sf.orcc.tutorial/src

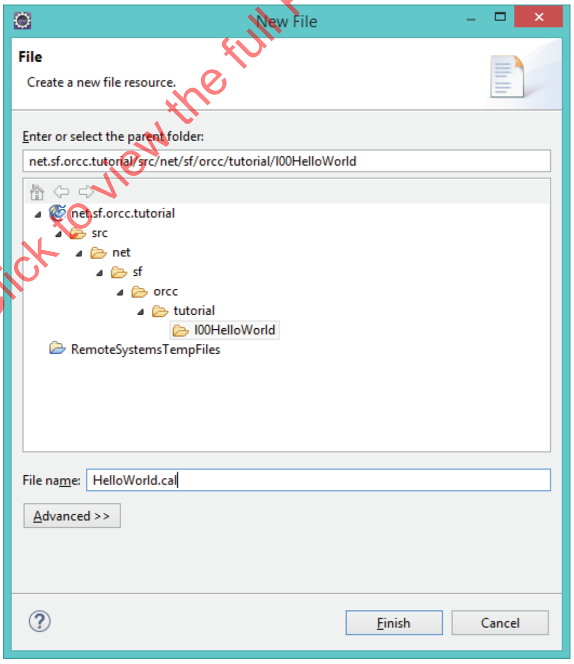
Name: net.sf.orcc.tutorial.I00HelloWorld

Create package-info.java

Figure 5 — Package setting in ORCC

5.3.3 Creating a new actor

In the *Project explorer* pane, select the package that was just created. Then click menu File > New > File. Specify the name and the extension *.cal* (see Figure 6).



Enter or select the parent folder:

net.sf.orcc.tutorial/src/net/sf/orcc/tutorial/I00HelloWorld

File name: HelloWorld.cal

Figure 6 — New CAL file setting in ORCC

After file is created, add the following code and save it. Eclipse will automatically compile the file.

```
package net.sf.orcc.tutorial.HelloWorld;

actor HelloWorld () int In ==> :
    action ==>
    do
        print("Hello World!\n");
    end
end
```

The code above implements the actor named **HelloWorld()**, which takes an input stream of tokens of type **int**.

The action within this actor always executes regardless of any input or other conditions and prints the string to the default output.

A detailed explanation of the syntax is given in the succeeding subclauses.

5.3.4 Creating a network

RVC-CAL is a language that implements the dataflow paradigm. This means that in order to run the application, it will be necessary to build a network of actors. In this case, the network will consist of only one actor.

To build a network, it will be necessary to create a new .xdf file. Go to File > New > Other, then select Orcc > XDF Network (see Figures 7 and 8).

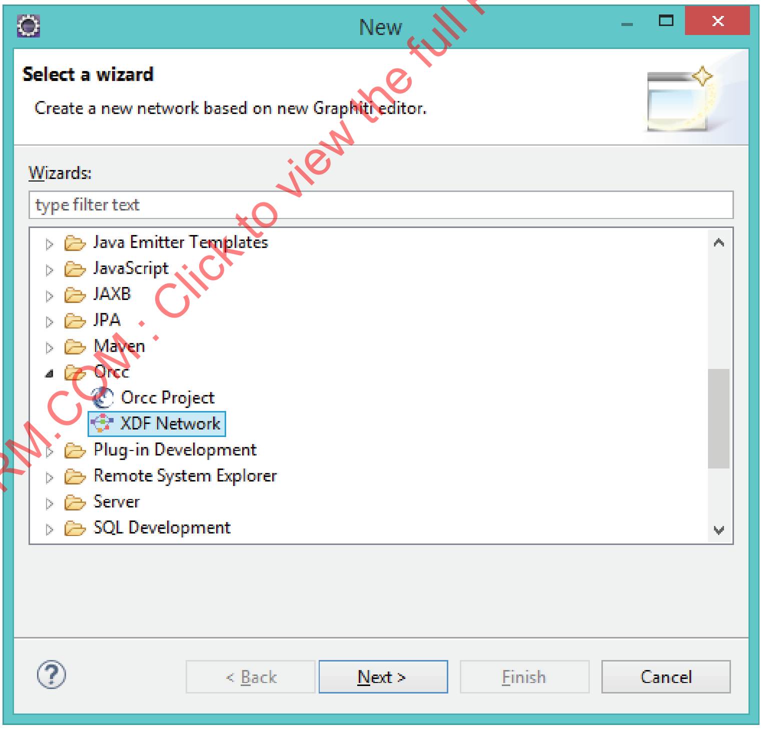
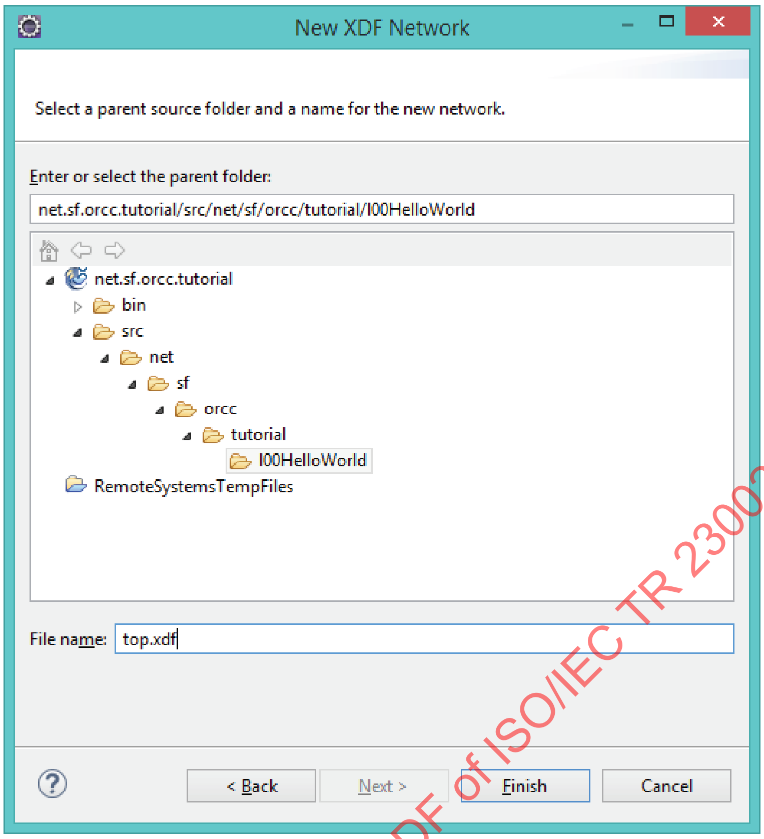


Figure 7 — Network setting in ORCC


New XDF Network

Select a parent source folder and a name for the new network.

Enter or select the parent folder:

net.sf.orcc.tutorial/src/net/sf/orcc/tutorial/I00HelloWorld

```
net.sf.orcc.tutorial
    bin
    src
        net
            sf
                orcc
                    tutorial
                        I00HelloWorld
RemoteSystemsTempFiles
```

File name: top.xdf

Figure 8 — Network name setting in ORCC

After the empty XDF is created, it will then be necessary to add an instance of an actor (see Figure 9).

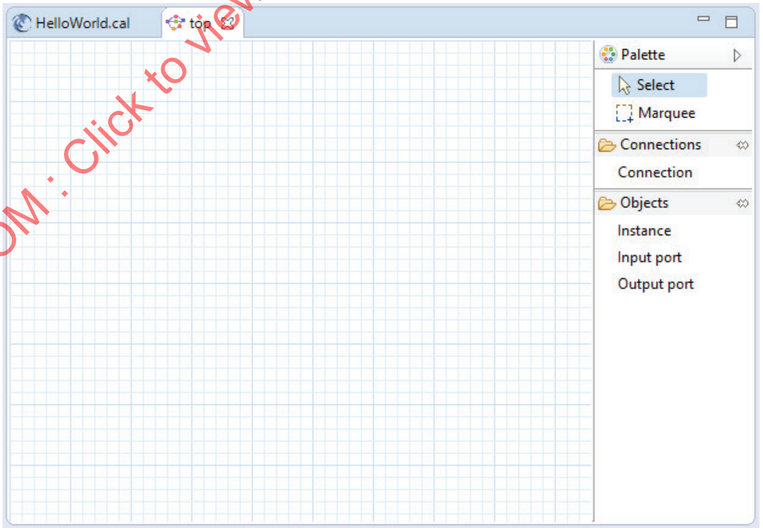


Figure 9 — Empty space for actor instance addition in ORCC

Click on Objects > Instance in the *Palette*, and then click on the .xdf file area to add an instance to the network (see Figure 10).

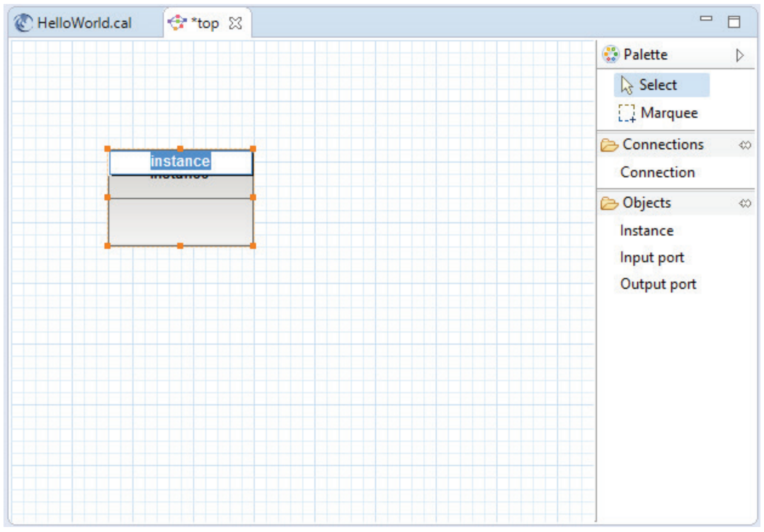


Figure 10 — Actor instance addition in ORCC

Name the instance “Hello”.

Now link this instance to the actor created. Right-click on the instance and select Set/Update Refinement (see Figure 11).

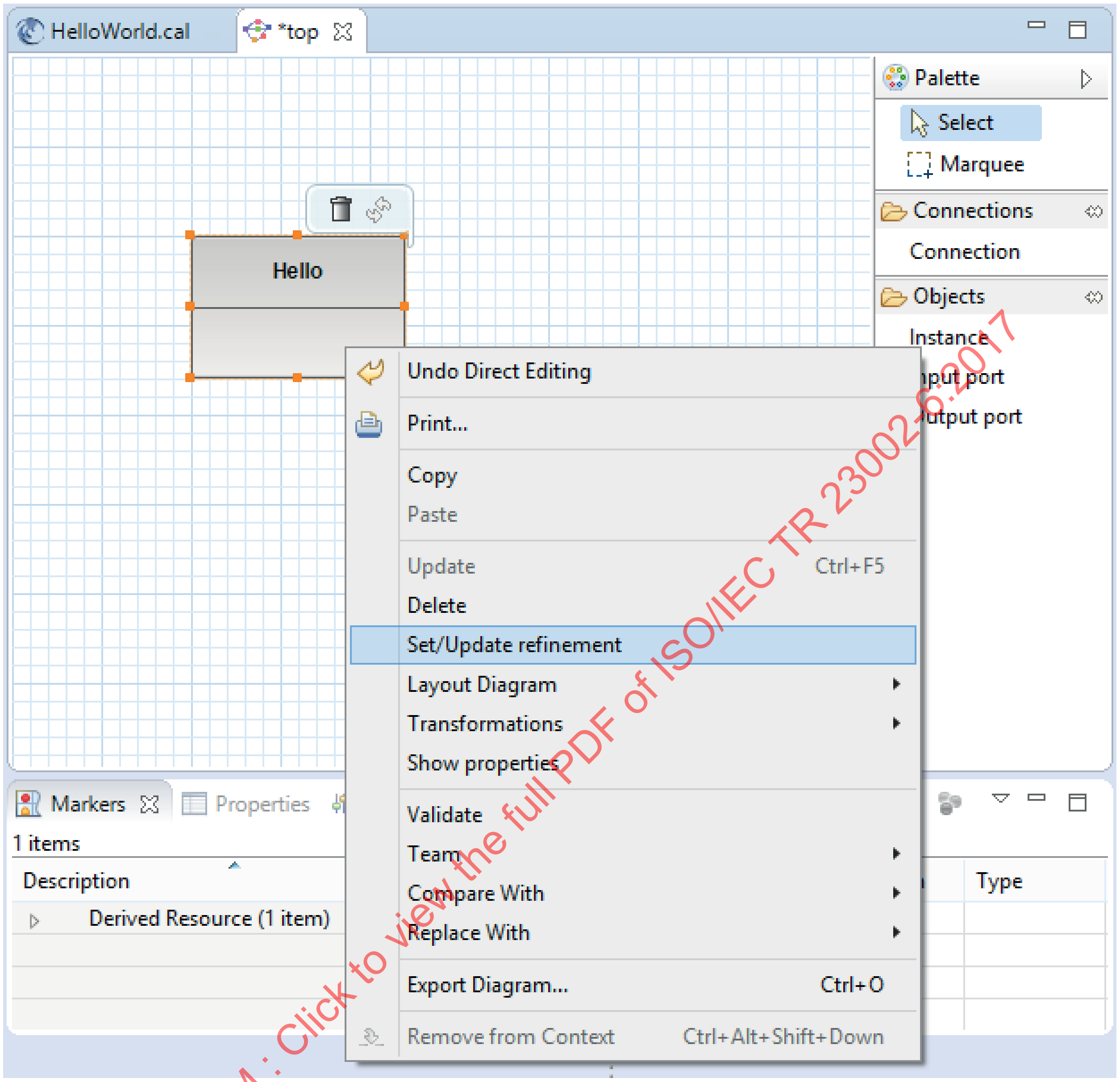


Figure 11 — Refinement of an actor instance to the corresponding CAL file in ORCC

Select the *HelloWorld* actor in the newly opened box (see Figure 12).

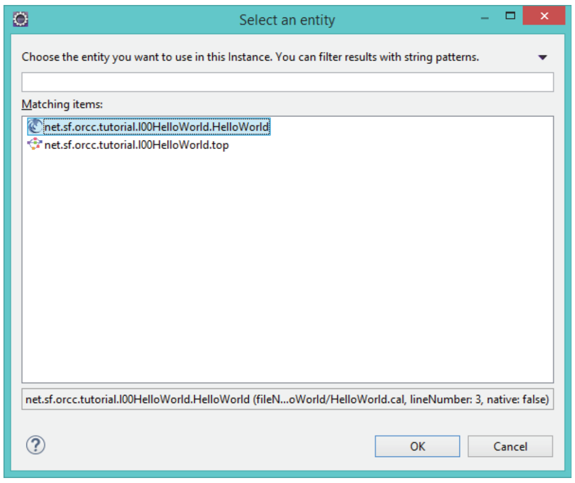

Select an entity
Choose the entity you want to use in this Instance. You can filter results with string patterns.
Matching items:
net.sf.orcc.tutorial.l00HelloWorld.HelloWorld
net.sf.orcc.tutorial.l00HelloWorld.top
net.sf.orcc.tutorial.l00HelloWorld.HelloWorld (fileN...oWorld/HelloWorld.cal, lineNumber: 3, native: false)
OK
Cancel

Figure 12 — Selection of a CAL file during an instance refinement process in ORCC

After validation, the *Hello* instance should be displayed in blue (meaning that the instance is assigned to an actor). See Figure 13.

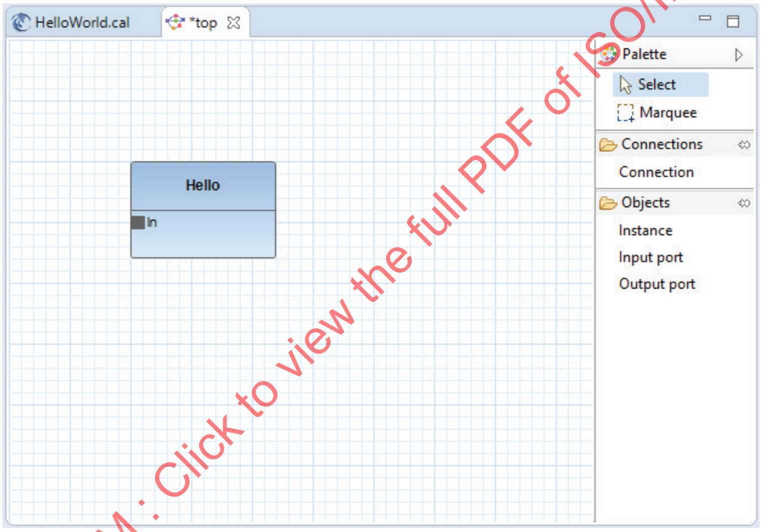


Figure 13 — Hello instance associated to the corresponding CAL file

5.3.5 Running simulation

Right-click on the .xdf file and select Run As > Orcc simulation (see Figure 14).

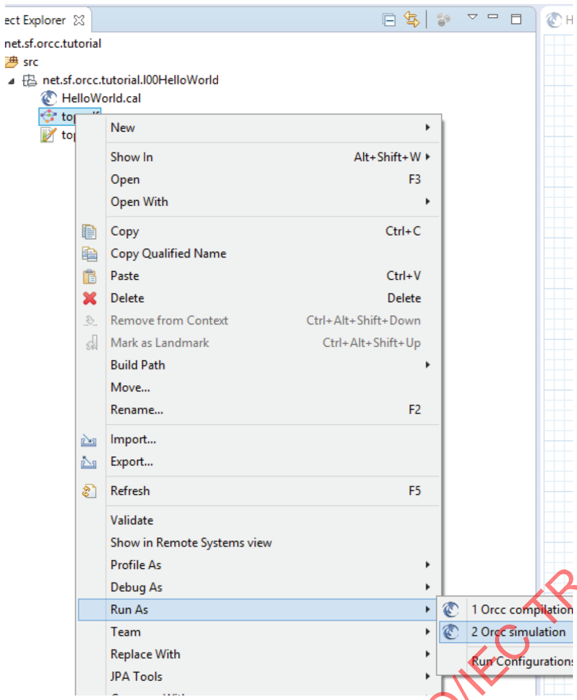


Figure 14 — Simulation selection in ORCC

In the *Select simulator* window, click OK (see Figure 15).

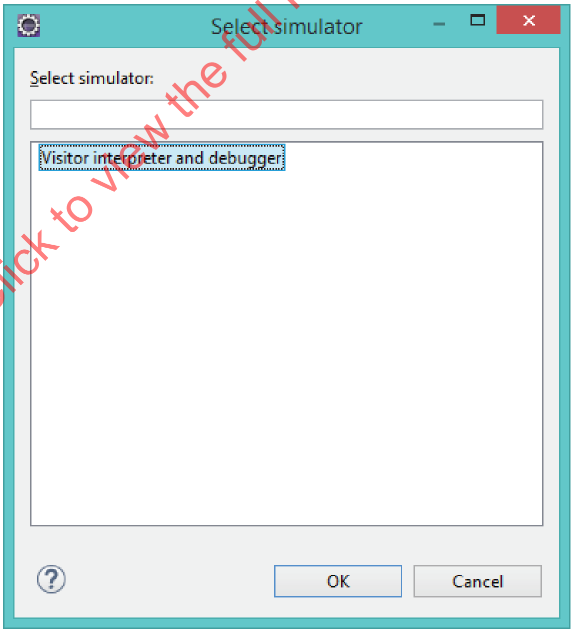
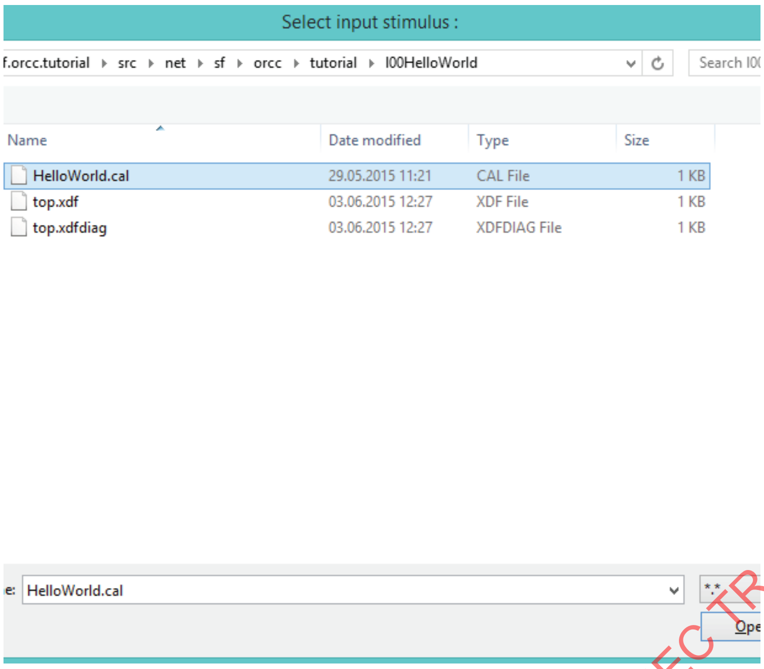


Figure 15 — Interpreter and debugger selection for simulation in ORCC

In the *Select input stimulus* window, select a random file (it will not actually be used by the example). See Figure 16.



Select input stimulus :

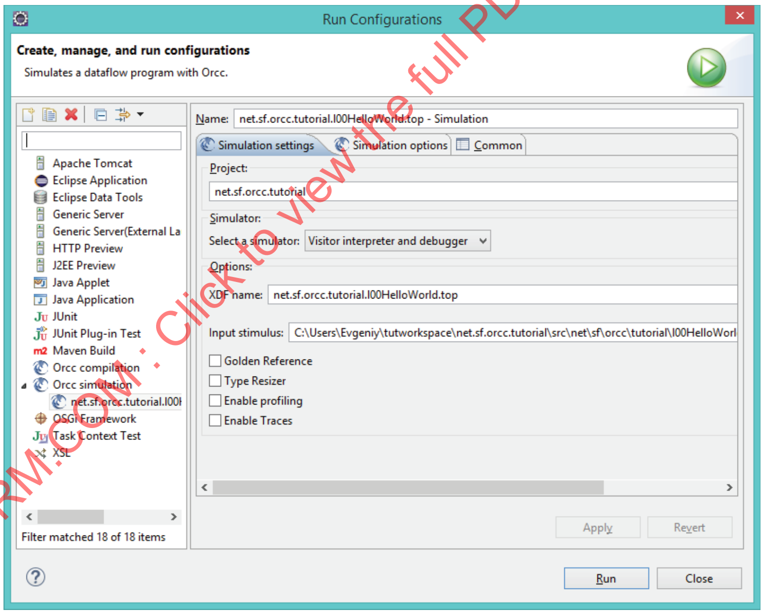
f.orcc.tutorial ▸ src ▸ net ▸ sf ▸ orcc ▸ tutorial ▸ I00HelloWorld

Name	Date modified	Type	Size
HelloWorld.cal	29.05.2015 11:21	CAL File	1 KB
top.xdf	03.06.2015 12:27	XDF File	1 KB
top.xdfdiag	03.06.2015 12:27	XDFDIAG File	1 KB

e: HelloWorld.cal

Figure 16 — Input stimuli selection for simulation in ORCC

In the *Run configuration wizard,* click on Run (see Figure 17).



Run Configurations

Create, manage, and run configurations

Simulates a dataflow program with Orcc.

Apache Tomcat
Eclipse Application
Eclipse Data Tools
Generic Server
Generic Server(External La
HTTP Preview
J2EE Preview
Java Applet
Java Application
JUnit
JUnit Plug-in Test
Maven Build
Orcc compilation
Orcc simulation
net.sf.orcc.tutorial.I00I
OSGi Framework
Task Context Test
XSL

Filter matched 18 of 18 items

Name: net.sf.orcc.tutorial.I00HelloWorld.top - Simulation

Simulation settings | Simulation options | Common

Project:
net.sf.orcc.tutorial

Simulator:
Select a simulator: Visitor interpreter and debugger

Options:
XDF name: net.sf.orcc.tutorial.I00HelloWorld.top
Input stimulus: C:\Users\Evgeniy\tutworkspace\net.sf.orcc.tutorial\src\net\sf\orcc\tutorial\I00HelloWorl

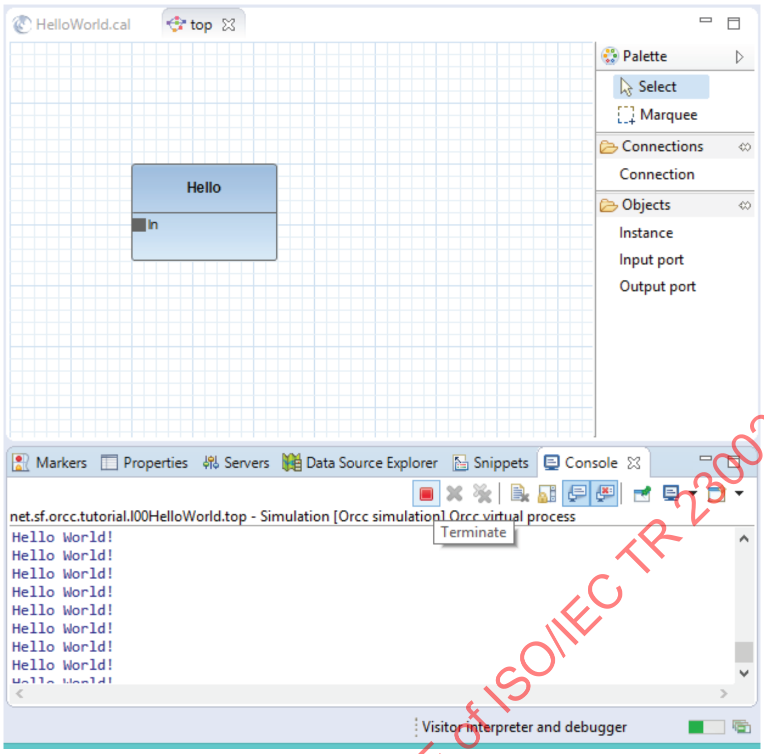
☐ Golden Reference
☐ Type Resizer
☐ Enable profiling
☐ Enable Traces

Apply | Revert

Run | Close

Figure 17 — Simulation configuration in ORCC

The following output (see Figure 18) should appear in the Eclipse console.



```
net.sf.orcc.tutorial.l00HelloWorld.top - Simulation [Orcc simulation] Orcc virtual process
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Figure 18 — Simulation console in ORCC

5.4 Simple actor

5.4.1 Structure of actors

Actors perform their computation in a sequence of steps called firings. In each of those steps, the actor

a) may consume tokens from its input ports,

b) may produce tokens at its output ports, and

c) may modify its internal state (this is described in further subclauses).

Describing an actor involves describing its interface, the ports, the structure of its internal state, as well as the steps it can perform, what these steps do (in terms of token production, consumption and actor state update) and how to select the step an actor will perform next.

5.4.2 Simplest actor

The simplest actor just copies a token from the input to the output without changing its value.

```
package net.sf.orcc.tutorial.SimpleActor;

actor ID () int In ==> int Out :
   first: action In: [a] ==> Out: [a] end
end
```

The first line specifies the package.

The main entity in RVC-CAL is an *actor*. In this example, the keyword **actor** is used followed by the name of the actor and the parameters in parentheses (an empty list in this example) to describe the actor. Input and output ports are specified before and after the ==> sign, respectively. RVC-CAL is a *statically typed* language, so it is necessary to explicitly define the type for each variable, i.e. in the line **actor ID () int In ==> int Out :** ports **In** and **Out** both are of type **int**. The colon at the end of the line marks the start of the actor body which is bounded by the keyword **end** from the other side.

Each actor has one or more actions within the body section, which execute (*fire*) at one step each. Actions may (or may not) *consume* input tokens and *produce* output tokens at each step. The syntax to describe an action in RVC-CAL is the following:

```
actor <ActorIdentifier> () <input ports> ==> <output ports> :
  [<ActionIdentifier>:] action <input pattern> ==> <output expression>
```

Input pattern specifies how many tokens to consume, from which ports the tokens are consumed and how to call these tokens in the rest of the action. The input pattern for the **ID** actor is **In: [a]**. It tells the action to consume one token from the input **In** and name it **a** within the action body. *Input patterns* realize the idea of *pattern matching*.

The expression following the **==>** sign is an *output expression*. It defines the number and values of output tokens which will be produced on each output port by each *firing* of the action. In this example, **Out: [a]** is an *output expression.* It defines that exactly one token will be produced on the output port **Out** and the value of that token will be **a**.

It is important to understand the difference between *input pattern* and *output expression.* In the input pattern, the local variable **a** is declared and assigned to the value of the input token whenever the action is fired. The output expression uses that variable and send the value of **a** as a produced token to the output port at the end of the action firing.

5.4.3 Running the examples

In the project, create a new package **net.sf.orcc.tutorial.SimpleActor**

Create new CAL file named *ID.cal* and copy the following code there.

```
package net.sf.orcc.tutorial.SimpleActor;

actor ID () int In ==> int Out :
   first: action In: [a] ==> Out: [a] end
end
```

In order to build a network, several additional actors are needed to produce data and print results to the console. This example uses **Source** and **Printer** actors as utilities. These can be downloaded from the GitHub repository (https://github.com/orcc/rvccaltut/tree/master/net.sf.orcc.tutorial/src/net/sf/orcc/tutorial/utils) and added to the project.

Source actor is a counter which produces a continuous sequence of numbers. Parameters can be specified for the starting number (default is 1) and the counter upper bound (default is 10). To know how to do that, see 5.4.4.4.

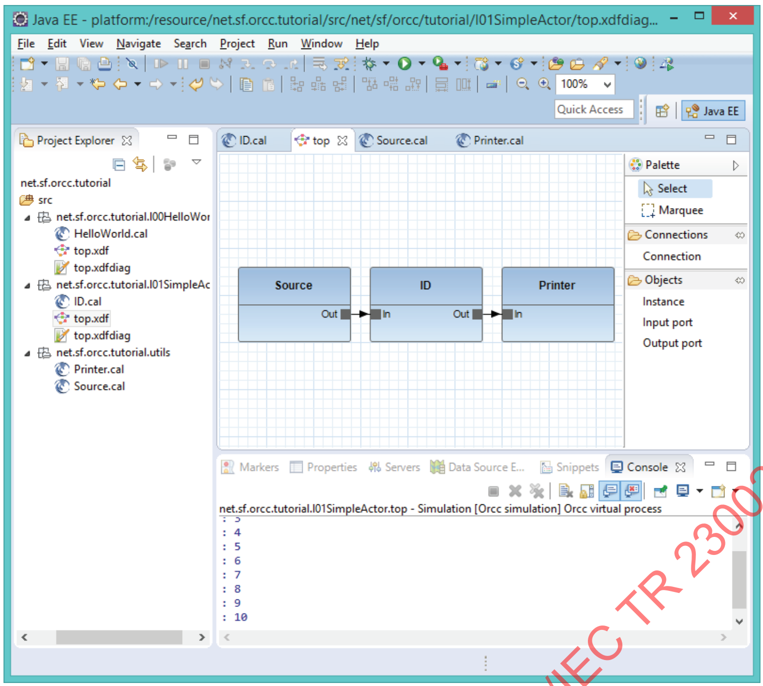
Printer prints all the consumed tokens to the console. The name can be specified for each instance. That is useful when there are several of them printing to the same console simultaneously.

The network can be created as indicated in 5.3.

After adding instances of actors **ID**, **Source** and **Printer**, the **Source** output should be connected to the **ID** input, and the **ID** output to the **Printer** input.

Note that it is possible to drag-and-drop an actor file from the Project explorer pane to the XDF network diagram to add an instance of an actor.

Now the example can be run as described in 5.3 and the result can be seen in the console (see Figure 19).



```
net.sf.orcc.tutorial.I01SimpleActor.top - Simulation [Orcc simulation] Orcc virtual process
: 3
: 4
: 5
: 6
: 7
: 8
: 9
: 10
```

Figure 19 — Simulation of a simple network of actors in ORCC

5.4.4 Other simple actors

5.4.4.1 Add

The next example shows how to make an actor which will be as simple as **ID** but at the same time will perform a real manipulation on the data.

```
package net.sf.orcc.tutorial.SimpleActor;

actor Add () int In1, int In2 ==> int Out :
    action In1: [a], In2: [b] ==> Out: [a+b]
    end
end
```

This package has two *input ports* separated by a comma, **int In1, int In2**, in the actor declaration. Also, *input pattern* changed to **In1: [a], In2: [b]** which means that action will be *fired* only when both ports, **In1** and **In2**, have a valid data on their inputs. Consumed tokens will then be assigned to **a** and **b**, respectively. Moreover, this example clarifies the difference between *input pattern* and *output expression*. **Out: [a+b]**, the *output expression*, includes an actual expression (the sum of two variables), which is calculated after the action is finished and the result is sent to the output port.

5.4.4.2 AddSeq

The example in 5.4.4.1 consumes two tokens from two input ports. The following example shows how to add two values with only one input port.

```
package net.sf.orcc.tutorial.SimpleActor;

actor AddSeq () int In1 ==> int Out :
    action In1: [a, b] ==> Out: [a+b]
    end
end
```

It can be observed that the input pattern **In1: [a, b]** consumes two tokens from the same input during single firing. It is important to note that the action will fire *only* when the data on the input will match the input pattern. And since the pattern consists of two tokens, action will fire only when there are two tokens available on the input.

It is also possible to put more than two tokens separated by commas in the input pattern.

5.4.4.3 AddSubSeq

The output expression, as illustrated in this example, can also produce more than one token. These expressions are separated by commas within square brackets.

```
package net.sf.orcc.tutorial.SimpleActor;

actor AddSubSeq () int In1 ==> int Out :
    action In1: [a, b] ==> Out: [a+b,a-b]
    end
end
```

5.4.4.4 Scale

The following example shows another operation the *output expression* can perform.

```
package net.sf.orcc.tutorial.SimpleActor;

actor Scale (int k=1) int In ==> int Out :
    action In: [a] ==> Out: [k*a]
    end
end
```

Note that the actor parameters field was not left empty. The expression **int k=1** has been used to introduce the parameter **k** which has the default value of **1**.

The parameters can be modified after adding an instance of an actor to the XDF network. This is done by right-clicking on the instance rectangle and then choosing the Show properties item (see Figure 20).

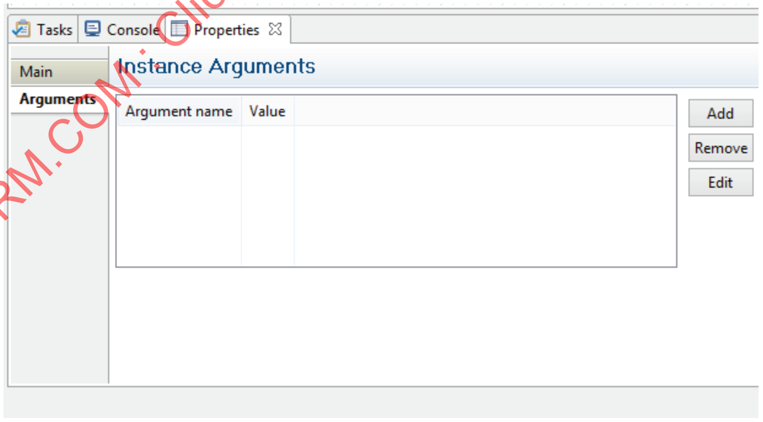


Figure 20 — Actor parameter list in ORCC

In the *Properties pane*, click on the Arguments on the left to display a list of arguments. Click on Add button and specify Name **k** and Value **7** (see Figure 21).

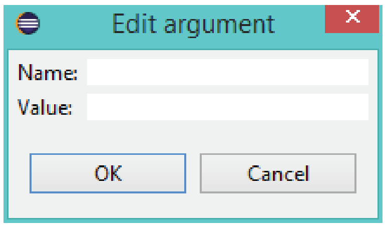


Figure 21 — Actor parameter setting in ORCC

When the network is run, this particular instance of the actor will multiply the input token by 7.

Note that the parameters for the **Source** and **Printer** actors can be specified using files downloaded from the GitHub repository (see 5.4.3).

5.4.5 Network of simple actors

After finishing all the examples in 5.4.4, a network similar to Figure 22 can be built.

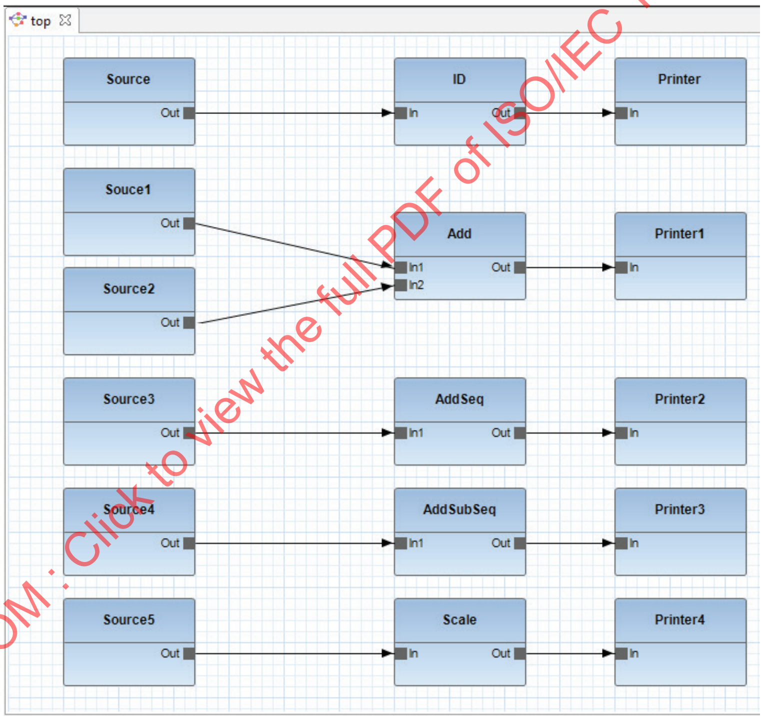


Figure 22 — Network of simple actors

If no parameter **name** of the actor **Printer** is specified (as described in 5.4.4), it cannot be determined from the console output which actor prints what. And since this parameter is a string, it should be surrounded by quotation marks, e.g. **“Printer1”**.

5.5 Non-determinism

As mentioned in 5.4.4.1 to 5.4.4.4, actors may have multiple actions. So in the following example, there are two.

```
package net.sf.orcc.tutorial.Nondeterminism;

actor NDMerge () int In1, int In2 ==> int Out :
   action In1: [x] ==> Out: [x] end
   action In2: [x] ==> Out: [x] end
end
```

This actor merges two input streams into one output. The first action takes a token from the input **In1** and sends it to the output **Out**. The second one does the same but for the input **In2**. However, from the description, one cannot know how the actor will behave if there are tokens available on both input ports at the same time. The order of output tokens will be undefined. This behaviour is called *non-deterministic*.

Generally, non-determinism means that the program can produce different output while processing the same input data. But in case of **NDMerge**, the output is actually defined by the timings of the input streams. The ability to leave this choice open was added to the CAL language on purpose. For example, if there is no available data on the first stream and there are data on the second stream, the actor does not have to wait. It will send further the token from the input whichever will have it first. And if the timings of the input data are known, this will help to avoid stalls and unnecessary delays.

However, an actor can be made which will be really non-deterministic even the timings of input data are known. The following example of **NDSplit** shows this.

```
package net.sf.orcc.tutorial.Nondeterminism;

actor NDSplit () int In1 ==> int Out1, int Out2 :
   action In1: [x] ==> Out1: [x] end
   action In1: [x] ==> Out2: [x] end
end
```

Here, there is one input and two outputs. Two actions always have a condition to fire at the same moment.

A network similar to Figure 23 can be built to simulate the non-deterministic behaviour. However, because of deterministic nature of simulator's algorithms, the results will not look random.

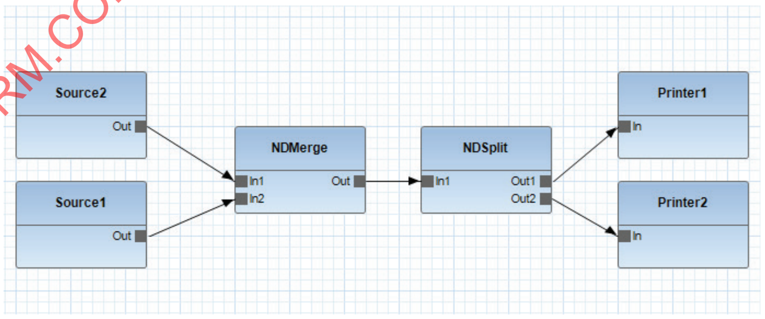


Figure 23 — Network of simple actors to explain non-determinism

5.6 Guarded actions

Non-determinism of multiple actions within one actor is introduced in 5.5. Even if one can exploit that property, it is generally undesirable to have such cases.

RVC-CAL provides options to restrict action firing conditions. One of them is using *guards*. This is a language construction which allows specifying additional requirements for an action to fire.

The following example illustrates how *guards* can be used.

```
package net.sf.orcc.tutorial.GuardedActions;

actor Split () int In ==> int P, int N :

   action In: [a] ==> P: [a]
   guard a >= 0 end

   action In: [a] ==> N: [a]
   guard a < 0 end

end
```

Line **guard a >= 0** in the definition of the first action defines a condition that fires the action only when the data on the input **In** is greater than or equal to zero. Similarly for the second action, **guard a < 0** means that the action will fire only when data is less than zero.

It is important to note that it is the user's responsibility to check that the guard conditions of all actions within an actor are exhaustive, i.e. that they cover all possible input. Otherwise, there will be cases when an actor will stall forever.

The next example (which is a wrong usage) shows what happens when not following this rule.

```
package net.sf.orcc.tutorial.GuardedActions;

actor SplitDead () int In ==> int P, int N :

   action In: [a] ==> P: [a]
   guard a > 0 end

   action In: [a] ==> N: [a]
   guard a < 0 end

end
```

The guard in the first action covers all the positive numbers and the guard in the second action covers all the negative ones. But if the input **In** is zero, this token will not cause any action to fire and it (therefore) will not be consumed, so no other tokens will come to the input **In**. The actor will stall forever.

Moreover, besides being exhaustive, *guards* in an actor should not have overlapped ranges. It can cause the errors explained in the following example.

```
package net.sf.orcc.tutorial.GuardedActions;

actor SplitND () int In ==> int P, int N :

   action In: [a] ==> P: [a]
   guard a >= 0 end

   action In: [a] ==> N: [a]
   guard a <= 0 end

end
```

Here, there are two guards, one is **guard a >= 0** and another is **guard a <= 0**. So the first action fires when there is a non-negative number on the input **In** and the second action fires when there is a non-positive one. It can be observed that zero satisfies conditions for both. This means that in case of zero on the input, the same non-determinism problem which was described in 5.5 can occur.

A final and important fact about the *guards* is that when guarding conditions is not fulfilled, the action does not fire, i.e. *the token is not consumed* and remains on the input so it could be consumed by the next firing or another action. It can be illustrated in the following example.

```
package net.sf.orcc.tutorial.GuardedActions;

actor Select () bool S, int A, int B ==> int Out :

    action S: [sel], A: [x] ==> Out: [x]
    guard sel end

    action S: [sel], B: [x] ==> Out: [x]
    guard not sel end

end
```

The code above is similar to the **NDMerge** in 5.5 and has an additional input **bool S**, data from which is used to select action to fire. Thus, in the example here, when there are tokens available on all three inputs and token from input **S** is **false**, the first action checks it but does not consume it, then the second action can consume it, fire and send the data from input **A** to the output.

The following network can be built. For the Boolean input of the actor from the last example, a special source generator **BoolGen** will be needed. This can be downloaded from the GitHub repository (https://github.com/orcc/rvccaltut/blob/master/net.sf.orcc.tutorial/src/net/sf/orcc/tutorial/utils/BoolGen.cal).

The **BoolGen** actor generates an infinite sequence of **[true, false, true, false,...]** (see Figure 24).

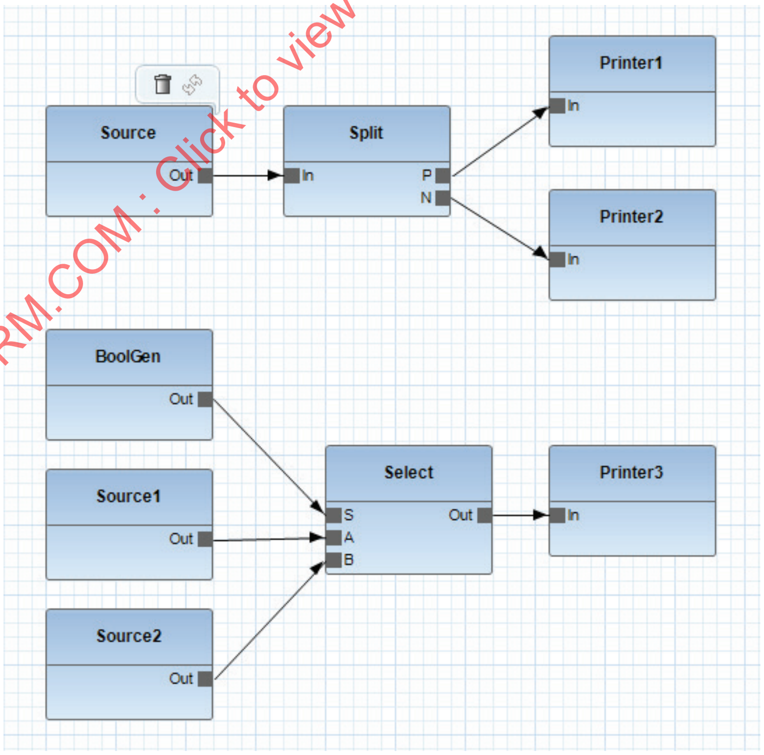


Figure 24 — Network of simple actors to explain guarded actions

5.7 State variables

Previous sections have illustrated how actions are fired on external conditions, but there is nothing inside the specification of an actor which can affect subsequent firings.

The *state variables* represent the internal memory of an actor. Actions within an actor can change its internal state and thereby alternate subsequent firings.

The simplest example of using state variable is a **Sum** actor.

```
package net.sf.orcc.tutorial.States;

actor Sum () int In ==> int Out :
   int sum := 0;
   action In: [a] ==> Out: [sum]
   do
      sum := sum + a;
   end
end
```

In the line **int sum := 0;**, the state variable **sum** is declared and initialized to zero. In this example, it can also be seen that an action can manipulate data within its body. In this case, the code between **do** and **end** updates the state variable **sum**, adding consumed token to it. Constructions like that are usually called accumulators. So here, it can be seen that an action not only consumes input token and produces output, but also modifies internal state of the actor, which will affect the output of the next firing.

It is important to notice here (although it was mentioned in previous subclauses) that the output expression is evaluated after action has been fired. The value of **sum** in output expression **Out: [sum]** is the one which has been updated by the action.

The previous example does not clearly represent the meaning of state variables. To explain this, an actor will be introduced which selects the input stream according to its internal state and sends a consumed token to the output (also recall the **Select** actor in 5.6).

```
package net.sf.orcc.tutorial.States;

actor IterSelect () bool S, int A, int B ==> int Out :
   int state := 0;
   action S: [sel] ==> guard state=0
   do
      if sel then
         state := 1;
      else
         state := 2;
      end
   end

   action A: [x] ==> Out: [x]
   guard state=1
   do
      state := 0;
   end

   action B: [x] ==> Out: [x]
   guard state=2
   do
      state := 0;
   end

end
```

Here, in the actor **IterSelect**, the state variable declaration **int state:= 0;** is found.

The first action consumes a token from the input **S** and does not produce any output. It only modifies the internal state. (Notice that there is no output expression after the **==>** sign but a guard.)

So this action is fired if current state is **0** and changes the state value to **1** if there is **true** on the input **S** or to **2** if there is **false**.

The second action changes **state** to zero and copies a token from input **A** to the output but only fires when current value of the internal state variable **state** is **1**. The third action does the same but only when **state** value is **2**.

Select and **IterSelect** are almost, but not entirely, equivalent. First, **IterSelect** makes twice as many steps in order to process the same number of tokens. Second, it actually reads, and therefore consumes, the token from input **S**, irrespective of whether a matching data token is available on **A** or **B**. And unlike the previous examples, the **IterSelect** actor uses guards that depend on an actor state variable rather than on an input token.

It is possible to use combinations of state variables and input tokens in guards, which is illustrated in the following example.

```
package net.sf.orcc.tutorial.States;

actor AddOrSub () int In ==> int Out :

   int sum := 0;

   action In: [a] ==> Out: [sum]
   guard a > sum
   do
      sum := sum + a;
   end

   action In: [a] ==> Out: [sum]
   guard a <= sum
   do
      sum := sum - a;
   end

end
```

Here, there are two actions: one of them adds the input token value to the state variable **sum** and another one subtracts the input from **sum** token, depending on whether or not the token is smaller than the value of **sum** itself.

A network similar to Figure 25 can be built to experiment with these actors.

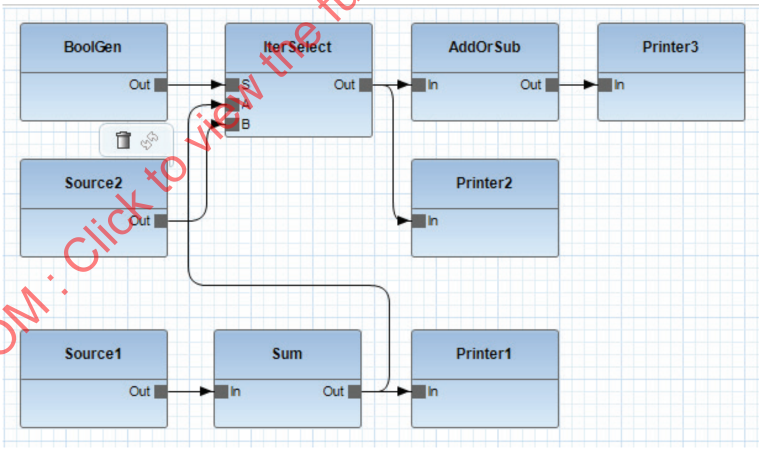


Figure 25 — Network of simple actors to explain state variables

5.8 Scheduling

The **InterSelect** example from 5.7 implements a commonly used software design pattern called finite state machines.

RVC-CAL provides special syntax to describe finite state machines. It is called *schedules*. The following example **IterSelectFSM** illustrates the use of *schedules.*

```
package net.sf.orcc.tutorial.Schedules;

actor IterSelectFSM () bool S, int A, int B ==> int Out :
   readT: action S: [sel] ==> guard sel end
   readF: action S: [sel] ==> guard not sel end
   copyA: action A: [x] ==> Out: [x] end
   copyB: action B: [x] ==> Out: [x] end

   schedule fsm init :
      init  (readT)  --> waitA;
      init  (readF)  --> waitB;
      waitA (copyA) --> init;
      waitB (copyB) --> init;
   end
end
```

Recall that every action can have an identifier or label, e.g. here **readT: action S: [sel] ==> guard sel end** the name of the action is **readT**. These labels are called *action tags*.

The block of code

```
schedule fsm init :
   init  (readT)  --> waitA;
   init  (readF)  --> waitB;
   waitA (copyA) --> init;
   waitB (copyB) --> init;
end
```

describes the automaton. Basically, it is a textual representation of a finite state machine given as a list of possible state transitions. The states of that finite state machine are the first and the last identifiers **(init**, **waitA** and **waitB**) in those transitions represented with the **-->** sign. Relating this back to the original version of **IterSelect**, these states are the possible values of the state variable, i.e. **0**, **1**, and **2**. The initial state of the schedule is the one following **schedule fsm**. In this example, it is **init**.

Each state transition consists of three parts: the *original state*, a *list of action tags* in parenthesis, and the *following state*. For instance, in the transition **init (readT) --> waitA;**, there is **init** as the original state, **readT** as the action tag, and **waitA** as the following state. The way to read this is that if the schedule is in state **init** and an action tagged with **readT** occurs, the schedule will subsequently be in state **waitA**.

The states **init**, **waitA** and **waitB** are imagined as circles, and the action tags **readT**, **readF**, **copyA** and **copyB** are imagined as arrows; an FSM diagram can be envisioned in the code, as illustrated in Figure 26.

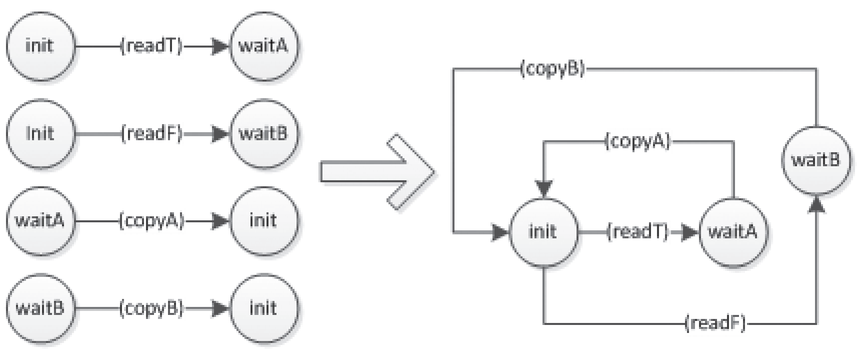


Figure 26 — State transitions and the equivalent finite state machine

The example above appears to show how to make implementation simpler and more readable. But in fact, it complicates the computation. In the original **IterSelect** actor, there were only three actions but here there are four.

A simpler example can be used to show how to avoid increasing complexity using *schedules*.

Actor **AlmostFairMerge** merges two streams almost fairly, as it is biased with respect to which input it starts reading from. But once it is running, it will strictly alternate between the two.

```
package net.sf.orcc.tutorial.Schedules;

actor AlmostFairMerge () int In1, int In2 ==> int Out :

   int state := 0;

   action In1: [x] ==> Out: [x]
   guard state=0
   do
      state := 1;
   end

   action In2: [x] ==> Out: [x]
   guard state=1
   do
      state := 0;
   end

end
```

The actor clearly has two states. It can be implemented using schedules as follows:

```
package net.sf.orcc.tutorial.Schedules;

actor AlmostFairMergeFSM () int In1, int In2 ==> int Out :

   A: action In1: [x] ==> Out: [x] end
   B: action In2: [x] ==> Out: [x] end

   schedule fsm S1 :
      S1 (A) --> S2;
      S2 (B) --> S1;
   end

end
```

Here, two actions **A**, **B** and two states **S1**, **S2** can be seen, which is the same as in the original actor.

A network can be created according to Figure 27.

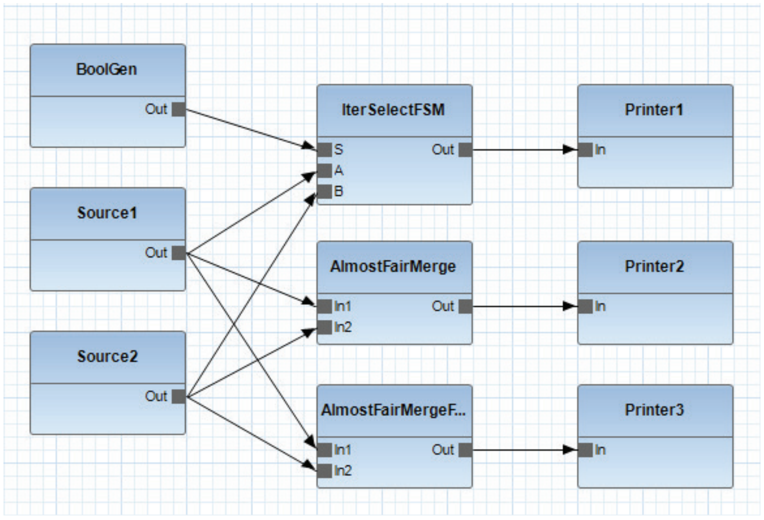


Figure 27 — Network of simple actors to explain finite state machines

5.9 Priorities

The previous subclauses have discussed *guards*, *states* and *schedules*. There is one more way to manage action firings in RVC-CAL.

In the case when conditions have been met for more than one action to fire, a higher priority can be given to some actions against others.

The following example illustrates how to use this in RVC-CAL.

```
package net.sf.orcc.tutorial.Priorities;

actor BiasedMerge () int A, int B ==> int Out :

    InA: action A: [x] ==> Out: [x] end
    InB: action B: [x] ==> Out: [x] end

    priority
        InA > InB;
    end

end
```

Here, there are two actions labelled **InA** and **InB**. And the line **InA > InB;** in the **priority ... end** block tells the actor that **InA** has a higher priority than **InB**. So in the case when tokens will be available on both inputs **A** and **B**, the token from input **A** will always go to the output first.

The following example illustrates how to give equal priorities to groups of actions.

```
package net.sf.orcc.tutorial.Priorities;

actor FairMerge () int A, int B ==> int Out :

   One.a:  action A: [x] ==> Out: [x] end
   One.b:  action B: [x] ==> Out: [x] end
   Both.a: action A: [x], B: [y] ==> Out: [x,y] end
   Both.b: action A: [x], B: [y] ==> Out: [y,x] end

   priority
      Both > One;
   end

end
```

It is necessary to pay attention to the action tagging. Actions can be grouped by labelling them as **One.a**, **One.b**. So here, there is a group **One**. Similarly, two actions can be tagged to the group **Both**.

And finally, a higher priority is given to the group **Both** (see Figure 28).

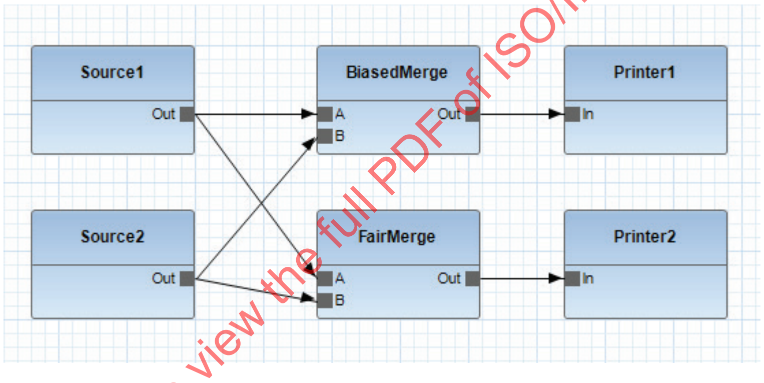


Figure 28 — Network of simple actors to explain priorities

5.10 Repeat clause

This subclause discusses *input patterns* and *output expressions* anew.

Earlier, it was shown that an *input pattern* of an action has two main functions: a) the *input pattern* defines the requirements for the action to be fired based on tokens available on the input; and b) the *input pattern* declares variables which will be used in the body of an action.

Simple cases that were discussed in the previous subclauses did not cover all the possible variants of input data. Another form of input is discussed in the following example.

```
package net.sf.orcc.tutorial.Repeat;

actor Reduce () int In1 ==> int Out :

   action In1: [a1,a2,a3,a4,a5,a6,a7] ==> Out: [a1,a2]
   guard a1 mod 2=0
   end

end
```

Here, seven tokens are read from the input and if the first one is an even number, the first two tokens are transferred further to the output **Out**.

It can be seen that having seven elements in the input pattern looks quite excessive. Larger numbers of elements, such as 64 or 1 024, would become unwieldy.

RVC-CAL has a special language construction called *repeat clause*.

```
package net.sf.orcc.tutorial.Repeat;

actor ReduceRP () int In1 ==> int Out :

   action In1: [a] repeat 15 ==> Out: [a] repeat 8
   guard a[0] mod 2=0
   end

end
```

Using the keyword **repeat**, the action can be set up to consume a defined pattern multiple times. Here, 15 tokens are read if the first one is even.

The consumed token **a** within the action's body can be referred to in an array, e.g. **a[2]**. So actually, **In1: [a] repeat 15** here defines an array **int a[15]** and initializes it with 15 input tokens of type **int**.

Similar construction can be used with output expression. As discussed in 5.3, output expressions contain a list of expressions to compute the values of output tokens. In the example above, a *repeat clause* is added, the defined output token will not be just reproduced n times. But the first **n** elements of array **a[15]** are sent to the output.

Using a *repeat clause* can produce a complicated output behaviour. The following example shows one of these cases.

```
package net.sf.orcc.tutorial.Repeat;

actor SplitRP () int In1 ==> int Out1, int Out2, int Out3 :

   action
      In1: [a,b,c] repeat 8
      ==>
      Out1: [a] repeat 8,
      Out2: [b] repeat 8,
      Out3: [c] repeat 8
   end

end
```

Here, the input pattern is **[a,b,c] repeat 8**. This means that action will consume these three tokens eight times. So if the input has

[1,2,3,4,5,6,7,8,9,...,24]

then in the action, there will be arrays

a[8] = [1,4,7,...,22]

b[8] = [2,5,8,...,23]

c[8] = [3,6,8,...,24]

The output expressions of the example send all three arrays to three different outputs.

Networks can be built for all these examples, and their behaviour can be observed using the actors **Source** and **Print** from the **utils** package.

5.11 Control flow

5.11.1 General

Previous subclauses illustrated dataflow abstractions in RVC-CAL such as actors, actions and XDF networks. But the language itself also contains elements of a procedural paradigm.

Control flow constructions and mutable variables can be used within each action.

5.11.2 Data types

Before proceeding with imperative elements of the language, data types used in RVC-CAL are introduced.

Table 1 shows all predefined data types used in RVC-CAL.

Table 1 — List of predefined datatypes in RVC-CAL

Data type	Example	Description
Bool	true	Boolean
int	–21	Integer
Uint	42	Unsigned integer
Float	237.2	Floating point numbers
String	“Hello”	Strings of characters
List(type: T, size = N)	[1,2,3]	Finite lists of N elements of type T

The first five of these predefined RVC-CAL data types should require no further explanation but the last one, **List**, requires some further discussion. RVC-CAL supports elements of a functional programming paradigm, which will be discussed in the following subclauses. Within the imperative paradigm, a **List** can be treated as arrays. Furthermore, RVC-CAL provides an alternative syntax for lists:

List(type: int, size = 64) foo is equivalent to **int foo[64]**

This format resembles that of an array.

5.11.3 Assignments

A mutable variable can be defined within an action and it can be assigned an initial value and changed during the action execution.

```
action ==> Out: [m,a[0]]
var
    int m := 0,
    int a[8]
do
    m := 10;
    a := [0,1,2,3,4,5,6,7];
    a[3] := m;
end
```

Every action can contain local variables which should be introduced in the block preceded by the keyword **var** just after the action declaration. Variables can also be initialized with values using operator **:=**. Definitions of different variables should be separated by commas.

The values of variables can be changed using assignment statements within the body of action which is bounded by keywords **do ... end**. Each statement should be terminated with a semicolon.

A working example with assignments is shown here:

```
package net.sf.orcc.tutorial.ControlFlow;

actor Fibonacci () ==> int Out :
   int fib[2] := [0,1];
   int counter := 0;

   action ==> Out: [fib[0]]
   guard
      counter < 20
   var
      int tmp
   do
      // Evaluate next Fibonacci number
      tmp := fib[1];
      fib[1] := fib[0] + fib[1];
      fib[0] := tmp;

      // Increment counter
      counter := counter + 1;
   end

end
```

The actor in this example produces a sequence of Fibonacci numbers.

The following subclauses describe the control flow constructions of RVC-CAL.

5.11.4 If statement

The first control flow statement that will be discussed is an "if" statement.

An if statement in RVC-CAL has the following syntax:

```
   if m > 0
   then
      m := m + n;
   else
      n := m + n;
   end
```

The else part can be omitted when not needed, as in:

```
   if m != 0
   then
      m := m + n;
   end
```

Table 2 provides a list of logical operators.

Table 2 — List of logical operators in RVC-CAL

Operator	Description
=	equal
!=	not equal
>	greater then
<	less then
>=	greater or equal
<=	less or equal

5.11.5 While statement

The syntax for a "while" statement is as follows.

```
    while n < 10
    do
        n := n + m;
    end
```

And the following example illustrates how to use while and if statements in actors:

```
package net.sf.orcc.tutorial.ControlFlow;

actor SatDotProduct (int level=1024) int In ==> int Out :

    action In: [x,y] repeat 8 ==> Out: [sum]
    var
        int i := 0,
        int sum := 0
    do
        while i < 8
        do
            sum := sum + x[i]*y[i];
            if sum > level
            then
                sum := level;
            end
            i := i + 1;
        end
    end
end
```

The actor **SatDotProduct** computes saturated dot-product of two vectors. This is using *repeat clause* in the input pattern to read two arrays of tokens as was explained in 5.10.

5.11.6 Foreach statement

A "foreach" statement has the following syntax:

```
        foreach int i in 0 .. 7
        do
            sum := sum + n + i;
        end
```

The next example shows how a dot product can be calculated using a foreach statement in RVC-CAL.

```
package net.sf.orcc.tutorial.ControlFlow;

actor DotProduct () int In ==> int Out :

   action In: [x,y] repeat 8 ==> Out: [sum]
   var
      int sum := 0
   do
      foreach int i in 0 .. 7
      do
         sum := sum + x[i]*y[i];
      end
   end
end
```

And the last example shows how to use nested foreach statements:

```
package net.sf.orcc.tutorial.ControlFlow;

actor MatrixProduct () int In1, int In2 ==> int Out :

   action In1: [x] repeat 8*8, In2: [y] repeat 8 ==> Out: [z] repeat 8
   var
      int z[8]
   do
      foreach int i in 0 .. 7 do
      z[i] := 0;
         foreach int j in 0 .. 7
         do
            z[i] := z[i] + x[i*8+j]*y[j];
         end
      end
   end
end
```

Here, a *repeat clause* is used in the output expression in order to send an array of tokens to the output.

6 Papify and Papify Viewer

6.1 General

This subclause provides a detailed tutorial on how to use Papify, a tool that implements an event-based performance monitoring in RVC-CAL. Papify integrates the Performance API (PAPI) into the Open-Source RVC-CAL Compiler (ORCC). Papify analyses in detail the performance of an implementation in a processor-based platform. Papify Viewer is a visualization tool to monitor the actions of actors in RVC-CAL specifications. Fired actions can be analysed chronologically from either an actor or a partition point of view. In addition, Papify Viewer can also generate event histograms.

6.2 Using Papify

Papify employs the annotation syntax defined in ISO/IEC 23001-4 to signal the instrumented actors and actions. Annotations are a common mechanism in the RVC-CAL language to drive the compiler behaviour. In addition, Papify can be employed to profile the video tool library (VTL) defined in ISO/IEC 23002-4.

In order to profile an actor, annotations of the form "@papify(ListOfEvents)" are employed, where the *ListOfEvents* is a non-empty, comma-separated sequence that comprises any of the *preset* events of the Performance API (PAPI).

To use Papify, installation of the Open RVC-CAL Compiler (ORCC) and PAPI are required. For installation of ORCC, see http://orcc.sourceforge.net/getting-started/install-orcc/. For installation of PAPI, which can be found at http://icl.cs.utk.edu/papi/, see the PAPI installation instructions at http://icl.cs.utk.edu/projects/papi/wiki/Installing_PAPI. No additional software needs to be installed for Papify, as it is already included with ORCC.

6.2.1 Papify activation

To activate Papify, the ordinary compilation process of ORCC needs to be followed. On the selected XDF network, right-click to get the menu Run As > Run Configurations (see Figure 29).

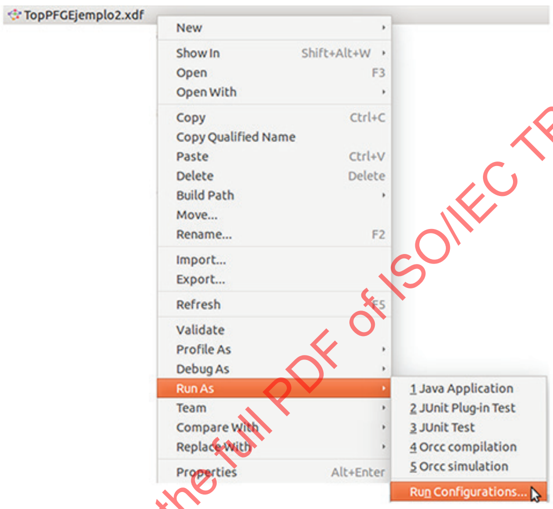


Figure 29 — Configuration setting to activate Papify in ORCC

Once the Run Configurations window is open, the C backend needs to be selected (see Figure 30).

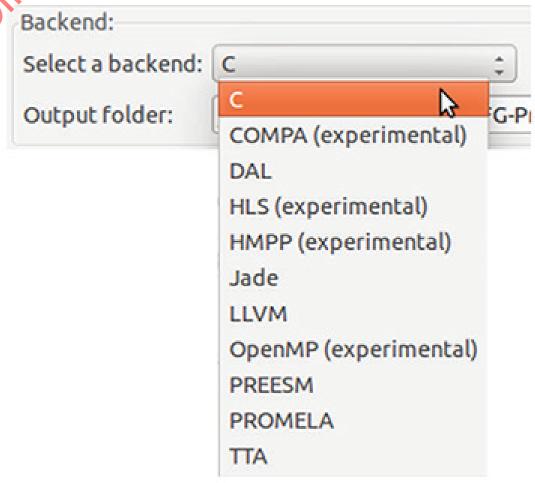


Figure 30 — C backend selection to activate Papify in ORCC

Within the backend options, the last one needs to be checked: *Papify: profile actors using PAPI*. Additionally, the option *Enable multiplex* may be selected in case the number of PMC counters available on the objective platform is less than the number of configured events (see Figure 31).

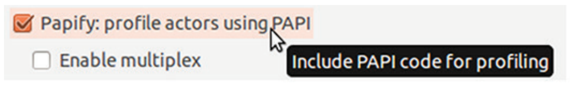

Papify: profile actors using PAPI
Enable multiplex
Include PAPI code for profiling

Figure 31 — Multiplex option selection to activate Papify in ORCC

Finally, click *Apply* and then *Run* to generate the C code with the PAPI function calls for the instrumented actors and actions.

6.2.2 Actor assessment

To assess all actions of an actor, an annotation should be included before the actor interface declaration. The format of the annotation is the following:

`@papify([event`$_1$`], [event`$_2$`], …, [event`$_n$`])`

where each:

`[event`$_i$`]`

is a PAPI preset event. At least one event should always be included.

Figure 32 shows a network of three actors employed to rotate an image. The functionality of the specification is as follows:

— Actor **Source** begins to send luminance samples to actor **Rotate**, who changes the sample orientation to 90° and stores the rotated values internally.

— Once **Source** sends all samples of the image file, **Rotate** passes the rotated luminance pixels to actor **Writer**, who writes the samples into the rotated image file.

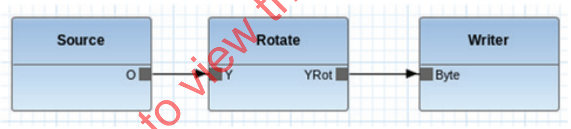


Figure 32 — Network of actors employed to rotate an image

In case the performance of the actor **Rotate** wants to be measured with, for instance, the events PAPI_TOT_INS and PAPI_L1_DCM, the following statement should be included above the actor declaration:

```
@papify(PAPI_TOT_INS, PAPI_L1_DCM)
actor Rotate()
uint(size=8) Y ==> uint(size=8) YRot :
/*(...)*/
```

To review the available events in a system, the following PAPI command can be utilized:

```
papi_avail
```

As a consequence of the available hardware resources in the objective platform, some events may not be mutually compatible during the execution of the implementation of an RVC-CAL specification. The command:

```
papi_event_chooser
```

This is a way to detect these conflicts. For instance, the command:

```
papi_event_chooser PRESET PAPI_TOT_INS PAPI_L1_DCM
```

will provide a list of events that can be merged in the same execution with PAPI_TOT_INS and PAPI_L1_DCM.

6.2.3 Action assessment

With Papify, it is possible to assess specific actions of an actor and remove the rest of them from the evaluation. To do so, the following annotation shall be included above the actions whose performance is required to measure:

```
@papify
```

Furthermore, to add the events to measure, the same annotation employed for the actor assessment shall also be included.

@papify([event$_1$], [event$_2$], …, [event$_n$])

In this way, the annotated actions can be instrumented with the events indicated at the actor level. For instance, below shows the performance measurement of the action Image_load of the actor Rotate with the events PAPI_TOT_INS and PAPI_L1_DCM.

```
@papify(PAPI_TOT_INS, PAPI_L1_DCM)
actor Rotate()
uint(size=8) Y ==> uint(size=8) YRot :

/* (...) */

 @papify
 Image.load: action Y:[pixel] ==>

 guard LoadUnload=true

 /* (...) */
```

6.2.4 Output folder

Once the execution of a specification instrumented with Papify has finished, a folder with the name papi-output is created. This folder is located in the folder /bin of the ORCC generated folders. Within the papi-output folder, the output files of each of the instrumented actors are written. These files are the input to the Papify Viewer tool.

6.3 Papify Viewer

Papify Viewer is a tool written in Processing[4], a programming language for visual applications that helps in the analysis of the activity file created with Papify. Due to the enormous amount of information typically generated in the activity file, the use of visual tools is recommended to get an insight of the action traces obtained during the execution of an RVC-CAL specification. Papify Viewer can additionally generate per-actor, per-action and per-partition histograms of events.

6.3.1 Chronological visualization

6.3.1.1 Actor point of view

Papify Viewer offers the possibility to have a chronological view per actor of the activity of a specification. Figure 33 shows a network of three actors employed to rotate an image. The generated per-actor chronological trace can be seen. Time is shown advancing from left to right. The meaning of the trace can be interpreted as follows: actor **Source** (in blue) begins to send luminance samples to actor **Rotate** (in red), which changes the sample orientation to 90° and stores the rotated values internally. Once **Source** sends all samples of the image file, **Rotate** passes the rotated luminance pixels to actor **Writer** (in green), which writes the samples into the rotated image file.

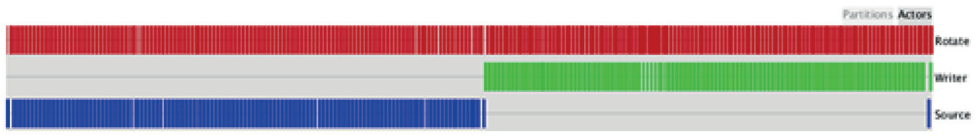


Figure 33 — Per-actor chronological trace

6.3.1.2 Partition point of view

In case an actor to core mapping file is provided to Papify Viewer (in an .xcf file), the tool can generate a per-partition chronological trace as shown in Figure 34. It can be seen that the **Source** and **Writer** actors are being executed in one core (Partition 1), while the **Rotate** actor is being executed in a different one (Partition 2).

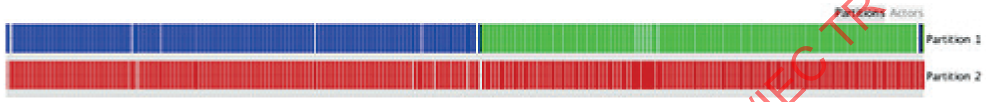


Figure 34 — Per-partition chronological trace

6.3.1.3 Zooming

Another possibility of Papify Viewer is to get a zoom in or out along the time axis. Figure 35 shows the zooming in on an interval in which the **Source** and **Writer** actors are being executed is applied to unveil the actual interleaving of actor actions. The time duration of these executions basically depends on the size of the FIFO queue and the number of actions needed to fill it or empty it.

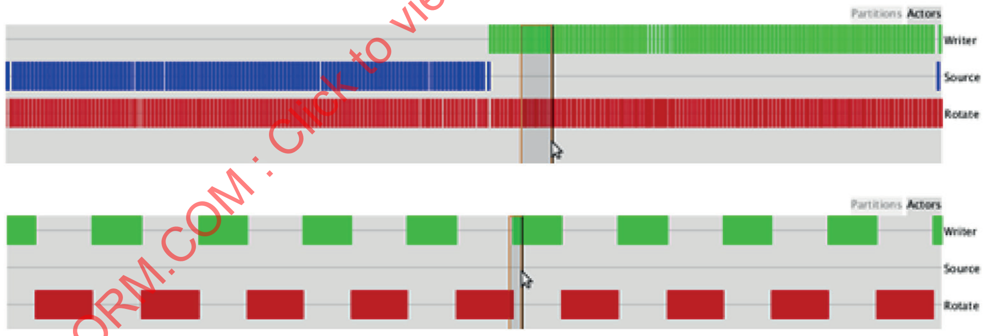


Figure 35 — Per-actor zooming

6.3.1.4 Action visualization

Papify Viewer can determine from a per-actor chronological trace the actions fired at each instant. As can be seen in Figure 36, when the cursor is located on one of the **Rotate** execution *blocks*, the corresponding executed action, *Image_unload*, is signalled.

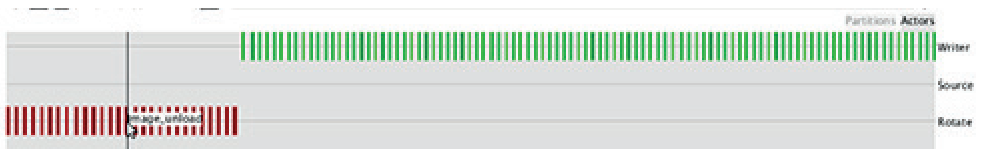


Figure 36 — Actions fired in a per-actor chronological trace

In case a per-partition chronological trace is available, the cursor indicates the actor being executed instead of the action, as shown in Figure 37.

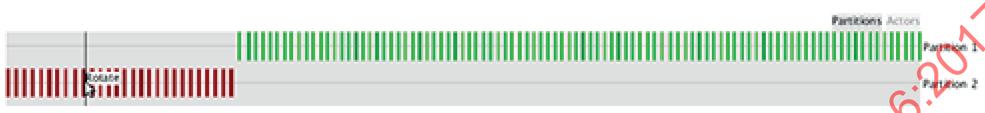


Figure 37 — Actions fired in a per-partition chronological trace

6.3.1.5 Complex specifications

In complex specifications, the visualization of any of the two traces (per-actor or per-partition) can help to better understand the behaviour of the system.

Figure 38 shows a per-actor chronological trace.

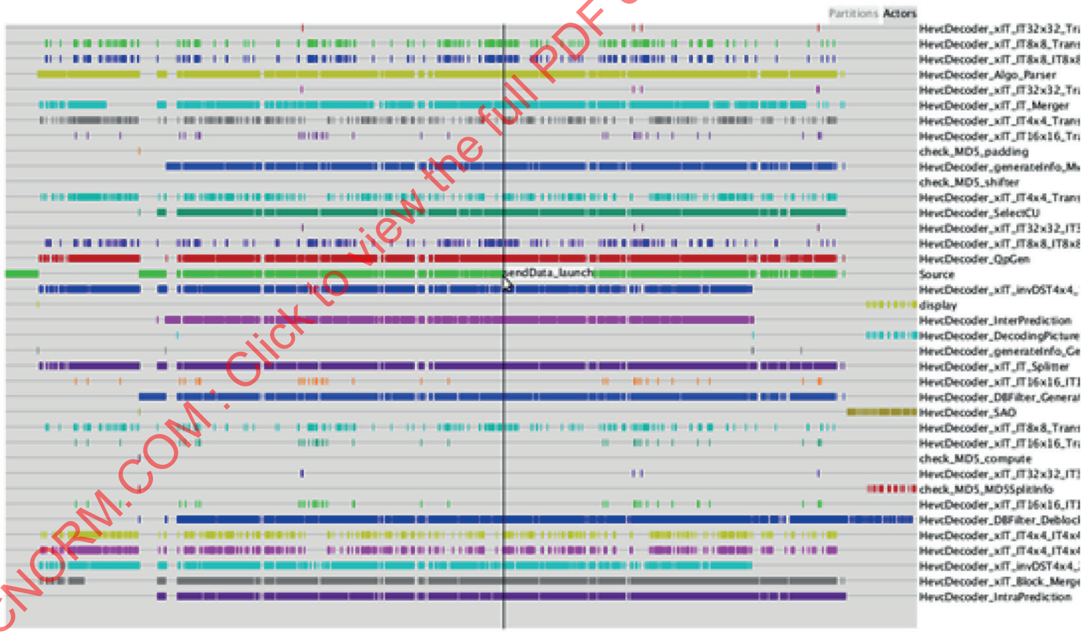

Partitions
Actors
sendData_launch
HevcDecoder_xIT_IT32x32_Tr
HevcDecoder_xIT_IT8x8_Trans
HevcDecoder_xIT_IT8x8_IT8x8
HevcDecoder_Algo_Parser
HevcDecoder_xIT_IT32x32_Tr
HevcDecoder_xIT_IT_Merger
HevcDecoder_xIT_IT4x4_Trans
HevcDecoder_xIT_IT16x16_Tr
check_MD5_padding
HevcDecoder_generateInfo_M
check_MD5_shifter
HevcDecoder_xIT_IT4x4_Trans
HevcDecoder_SelectCU
HevcDecoder_xIT_IT32x32_IT3
HevcDecoder_xIT_IT8x8_IT8x8
HevcDecoder_QpGen
Source
HevcDecoder_xIT_invDST4x4_
display
HevcDecoder_InterPrediction
HevcDecoder_DecodingPicture
HevcDecoder_generateInfo_Ge
HevcDecoder_xIT_IT_Splitter
HevcDecoder_xIT_IT16x16_IT1
HevcDecoder_DBFilter_General
HevcDecoder_SAO
HevcDecoder_xIT_IT8x8_Trans
HevcDecoder_xIT_IT16x16_Tr
check_MD5_compute
HevcDecoder_xIT_IT32x32_IT3
check_MD5_MD5SplitInfo
HevcDecoder_xIT_IT16x16_IT1
HevcDecoder_DBFilter_Debloc
HevcDecoder_xIT_IT4x4_IT4x4
HevcDecoder_xIT_IT4x4_IT4x4
HevcDecoder_xIT_invDST4x4_
HevcDecoder_xIT_Block_Merge
HevcDecoder_IntraPrediction

Figure 38 — Per-actor chronological trace for an HEVC decoder specification

Figure 39 shows the per-partition chronological trace of a frame decoding in an example sequence “Kristen and Sara” (1280 × 720, 60 fps, qp = 27) using a Main profile HEVC decoder.

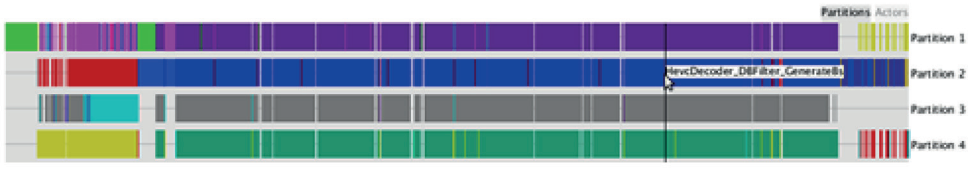


Figure 39 — Per-partition chronological trace for an HEVC decoder specification

6.3.2 Event histograms

6.3.2.1 Actors

Figure 40 shows the per-actor PAPI_TOT_INS histogram resulting from the instrumentation of the actor network given in 6.2.3. It is worth noting that when the cursor is located on a bin of the histogram, the actual value of the event count is shown at the y-axis. Either a linear or log scale can be used to represent the data. Furthermore, the total event count of an actor is shown when the cursor is located on the corresponding bin of the histogram.

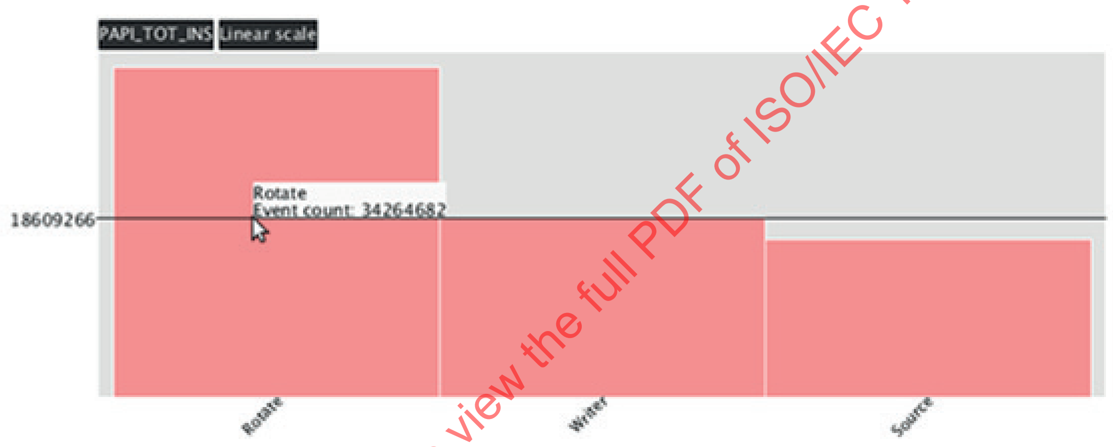


Figure 40 — Actor histogram for the PAPI_TOT_INS event

6.3.2.2 Actions

Papify Viewer can show event histograms of the actions of a selected actor as shown in Figure 41. With Papify Viewer beside the event count shown in Figure 40, the average number of events per action firing can be shown. For instance, the action *Image_unload* is fired 65 536 times with an average number of instructions per fire (occurrences of the PAPI_TOT_INS event) of 262.

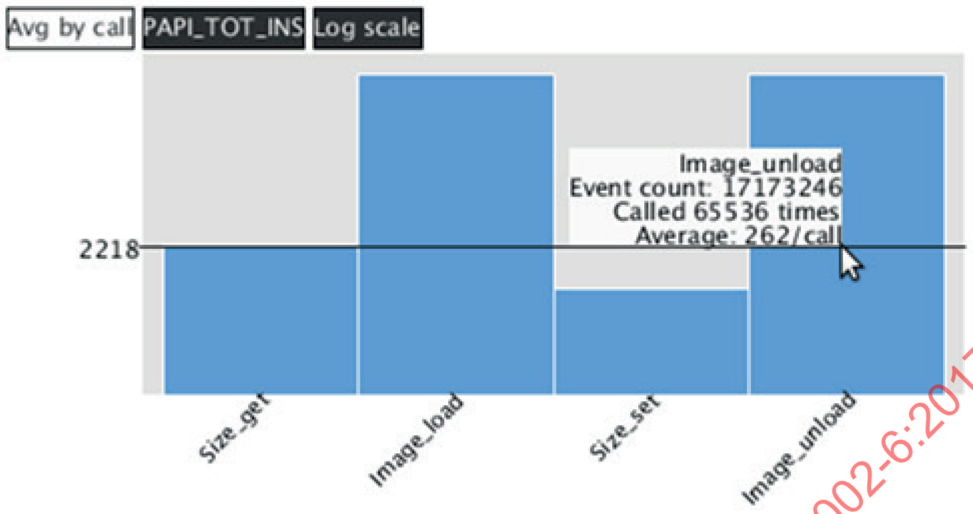


Figure 41 — Action histogram for the Rotate actor and the PAPI_TOT_INS event

6.3.2.3 Partitions

Additionally, Papify Viewer can show per-partition event histograms. Figure 42 shows the PAPI_TOT_INS event histogram of the partition discussed in 6.3.1.2 (**Source** and **Writer** in Partition 1 and **Rotate** actor in Partition 2).

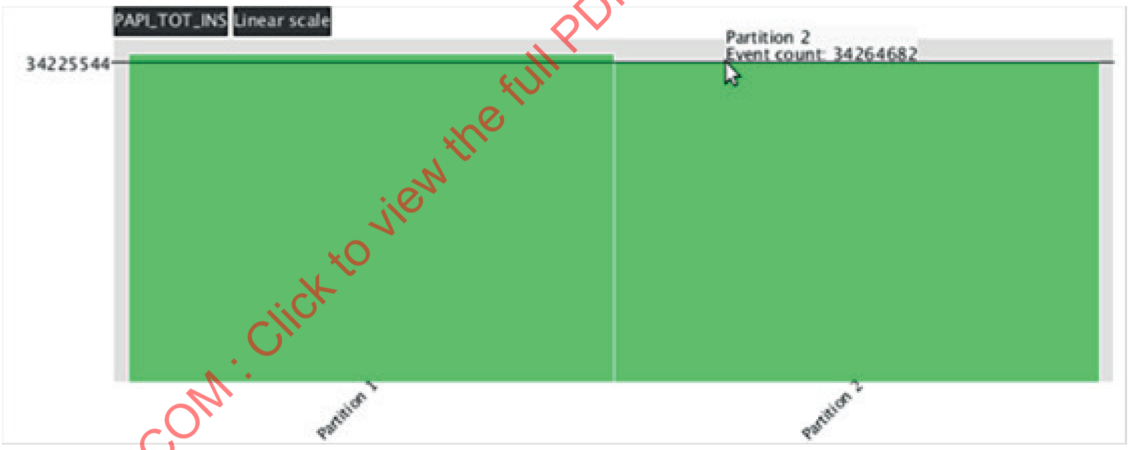


Figure 42 — Partition histogram

7 TURNUS

7.1 General

TURNUS is an open-source system design exploration and optimization framework tailored for CAL dataflow programs. TURNUS provides an application programming interface (API) to profile CAL programs that is usable by third-party dataflow compilers, such as ORCC, with the capability to model heterogeneous parallel platforms and a collection of multidimensional design spaces explorations (DSE) optimization heuristics based on the execution trace graph (ETG) post-mortem scheduling. The TURNUS profiler is used in the first stages of the DSE for evaluating both the ETG and the high-level profiling information of a CAL program. Successively, the ETG is post-mortem scheduled in order to estimate the design performance and explore and optimize the design space of the program.

Installation and use of the TURNUS framework is described in 7.2. An MPEG HEVC video decoder implemented with RVC-CAL dataflow language is used to describe the main design space exploration and optimization capabilities of the TURNUS framework.

7.2 Installing the TURNUS framework

7.2.1 General

The TURNUS framework is based on Java™[9] 1.8 and its graphical user interface (GUI) is integrated within the Eclipse IDE platform.

7.2.2 Java Runtime Environment

TURNUS requires the Java 1.8 (or higher) JRE. The latest JRE release can be downloaded from https://java.com/en/download/. To use Eclipse IDE for the Java Developers edition, the Java Development Kit will also need to be installed. The JDK can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/index.html.

7.2.3 Eclipse

TURNUS is compatible with Eclipse versions 4.4 (or higher). The Eclipse IDE can be downloaded from https://www.eclipse.org/downloads/ (Eclipse IDE for the Java Developers edition is suggested). To install, extract the archive into a local directory.

7.2.4 TURNUS plug-in for Eclipse

To install TURNUS, the Eclipse Software Update Manager tool should be used. From the Eclipse ID:

a) Go to Help > Install New Software.

b) Click Add to add an update site.

c) Set its name (e.g. TURNUS) and its URL to http://eclipse.turnus.co (see Figure 43).

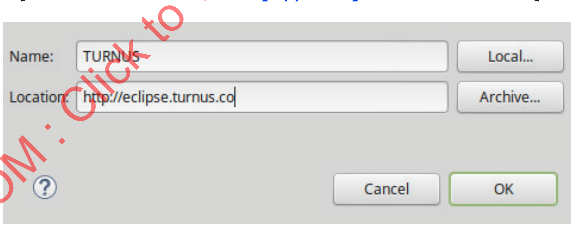
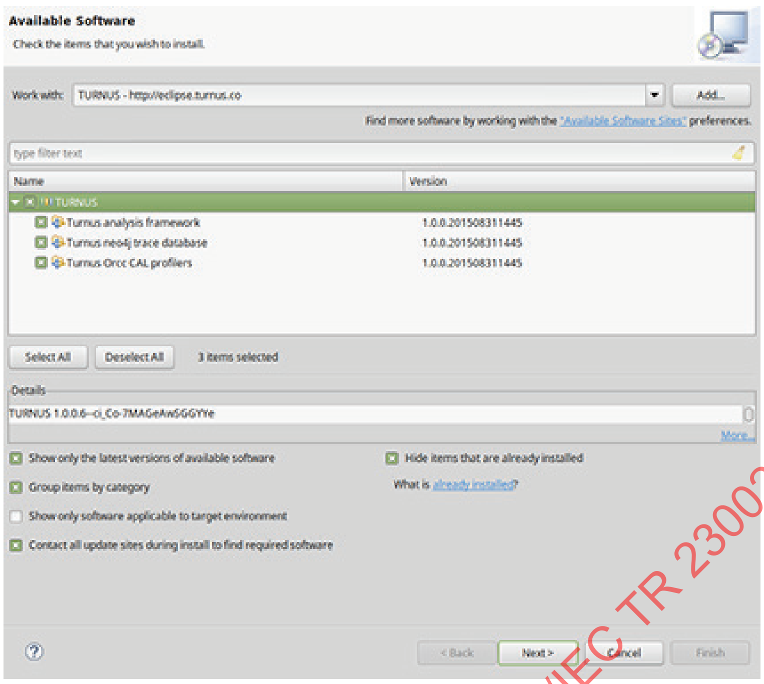

Name: TURNUS
Location: http://eclipse.turnus.co

Figure 43 — Repository setting during the installation of the TURNUS plug-in for Eclipse

d) Once done, select all the plug-ins within the TURNUS plug-ins group (i.e. TURNUS analysis framework, TURNUS neo4j trace database and TURNUS ORCC CAL profilers). See Figure 44.

9) Java is a trademark of a product supplied by Oracle. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named. Equivalent products may be used if they can be shown to lead to the same results.


Available Software

Check the items that you wish to install.

Work with: TURNUS - http://eclipse.turnus.co

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☒ TURNUS	
☒ Turnus analysis framework	1.0.0.201508311445
☒ Turnus neo4j trace database	1.0.0.201508311445
☒ Turnus Orcc CAL profilers	1.0.0.201508311445

3 items selected

Details

TURNUS 1.0.0.6--ci_Co-7MAGeAwSGGYYe

☒ Show only the latest versions of available software

☒ Group items by category

☐ Show only software applicable to target environment

☒ Contact all update sites during install to find required software

☒ Hide items that are already installed

What is already installed?

Figure 44 — Selection of the TURNUS framework during the installation of the TURNUS plug-in for Eclipse

e) Click Next, check and accept the licenses and then click Finish.

At this point, after restarting the Eclipse IDE, TURNUS and its required dependencies are installed.

7.3 Profiling an RVC-CAL HEVC video decoder

7.3.1 General

The HEVC decoder used in this document is the one available in the official RVC-CAL open-source applications (Orc-apps) repository. In the following subclauses, it is illustrated how the application can be statically and dynamically profiled. The following profilers are presented:

— TURNUS ORCC static code profiler;

— TURNUS ORCC dynamic code profiler.

7.3.2 Download the design and the conformance bit-streams

The link to this repository is https://github.com/orcc/orc-apps. To download the source code, clone this Git[10] repository:

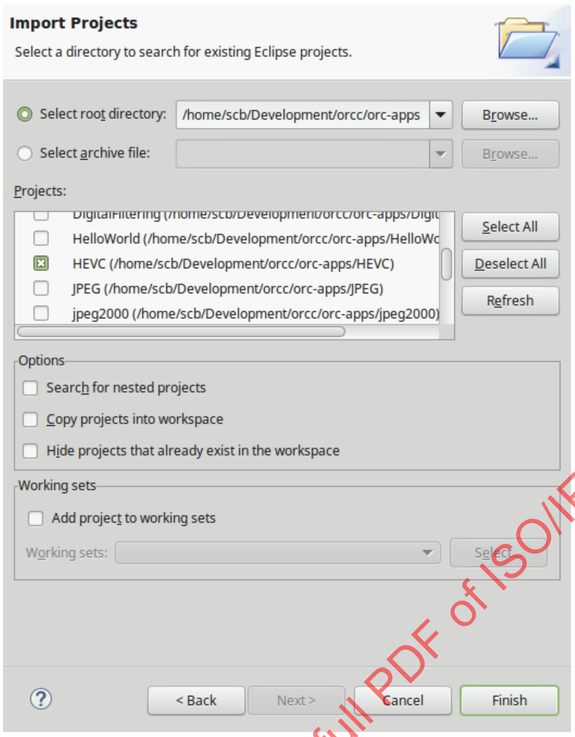
```
$ git clone https://github.com/orcc/orc-apps.git
```

An input sequence is also needed in order to effectively simulate the design. Some of the supported conformance bit streams can be downloaded from ftp://ftp.kw.bbc.co.uk/hevc/hm-10.0-anchors/bitstreams/lp_main/.

10) Git is a software configuration management package available on GitHub. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named. Equivalent products may be used if they can be shown to lead to the same results.

7.3.3 Import the HEVC design project in the Eclipse IDE workspace

From the Eclipse IDE, import the HEVC RVC-CAL project. In the menu File > Import, select General > Existing Projects into Workspace and select the path where the Orc-apps Git repository was previously cloned (see Figure 45).



Import Projects

Select a directory to search for existing Eclipse projects.

● Select root directory: /home/scb/Development/orcc/orc-apps [Browse...]

○ Select archive file: [Browse...]

Projects:

☐ DigitalFiltering (/home/scb/Development/orcc/orc-apps/Digit
☐ HelloWorld (/home/scb/Development/orcc/orc-apps/HelloWo
☒ HEVC (/home/scb/Development/orcc/orc-apps/HEVC)
☐ JPEG (/home/scb/Development/orcc/orc-apps/JPEG)
☐ jpeg2000 (/home/scb/Development/orcc/orc-apps/jpeg2000)

Options

☐ Search for nested projects
☐ Copy projects into workspace
☐ Hide projects that already exist in the workspace

Working sets

☐ Add project to working sets

Working sets:

Figure 45 — How to import the HEVC design project into the Eclipse IDE workspace

The following projects need to be selected:

— HEVC;

— System;

— RVC.

Click on the Finish button.

(Make sure that the active perspective in Eclipse is Java EE; otherwise, the menus could be different.)

At this point, all the imported projects are available in the *Project Explorer* viewer (see Figure 46).

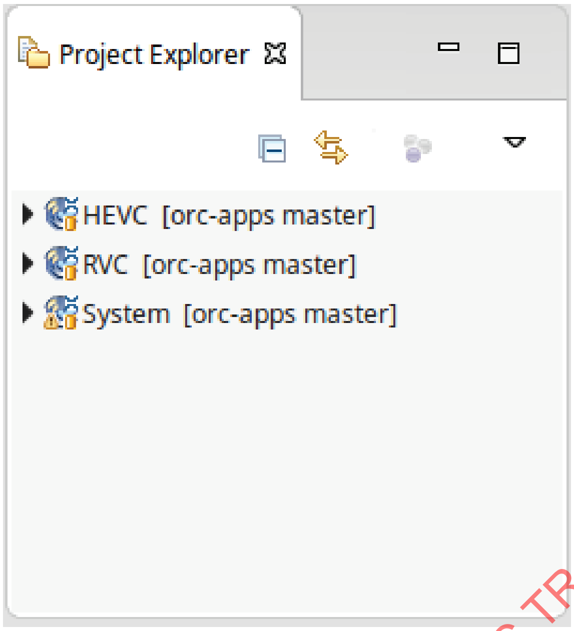


Figure 46 — Project explorer viewer

The design that is used in the following subclauses is

RVC/src/ org/sc29/wg11/mpegh/part2/Top_mpegh_part2_main_no_md5.xdf (see Figure 47).

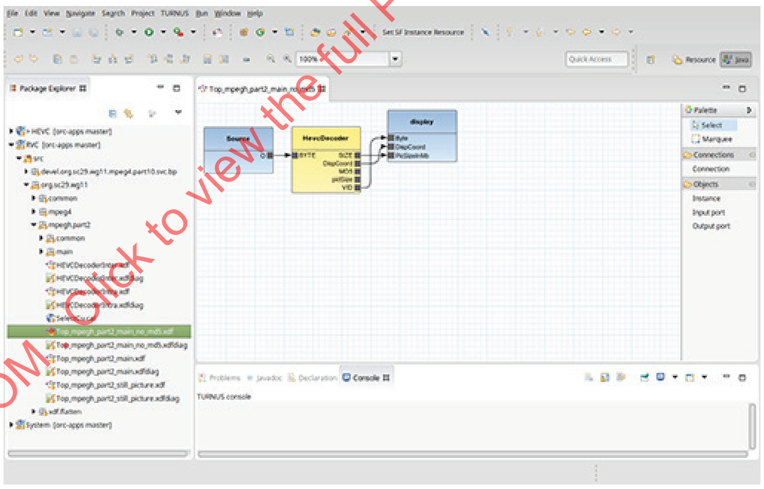


Figure 47 — Top HEVC network of actors

7.3.4 Static code profiling

With the TURNUS ORCC static profiler, basic information about the code complexity and maintainability of the design can be retrieved. This analysis is made without actually executing the program. The CAL code is analysed in order to provide information about its quality and maintainability according to some well-known static code analysis metrics (e.g. Halstead complexity measures).

To run this analysis, select and right-click on the .xdf file, then select Profile as > TURNUS static code analysis (see Figure 48).

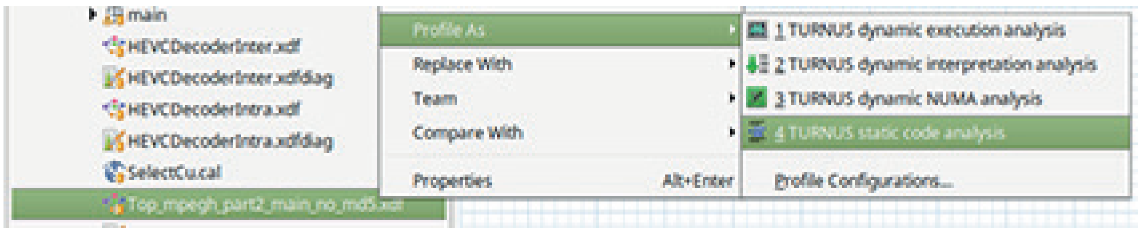
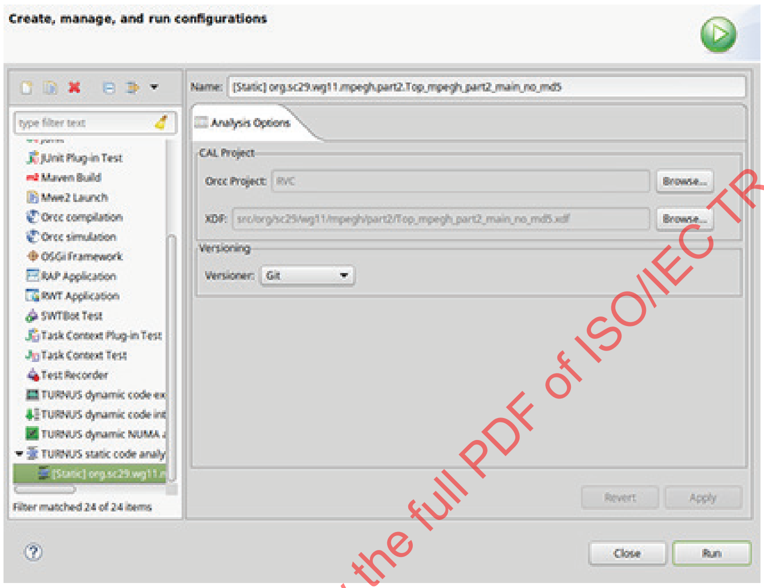

main
HEVCDecoderInter.xdf
HEVCDecoderInter.xdfdiag
HEVCDecoderIntra.xdf
HEVCDecoderIntra.xdfdiag
SelectCu.cal
Top_mpegh_part2_main_no_md5.xdf
Profile As
Replace With
Team
Compare With
Properties
Alt+Enter
1 TURNUS dynamic execution analysis
2 TURNUS dynamic interpretation analysis
3 TURNUS dynamic NUMA analysis
4 TURNUS static code analysis
Profile Configurations...

Figure 48 — Static code analysis setting in TURNUS

Successively, the configuration options table is shown (see Figure 49).


Create, manage, and run configurations

Name: [Static] org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main_no_md5

Analysis Options

CAL Project

Orcc Project: RVC

XDF: src/org/sc29/wg11/mpegh/part2/Top_mpegh_part2_main_no_md5.xdf

Versioning

Versioner: Git

Figure 49 — Configuration setting of the static analysis code in TURNUS

The program can be executed by clicking the Run button and the options described in Table 3 can be changed.

Table 3 — List of configuration options for static code analysis in TURNUS

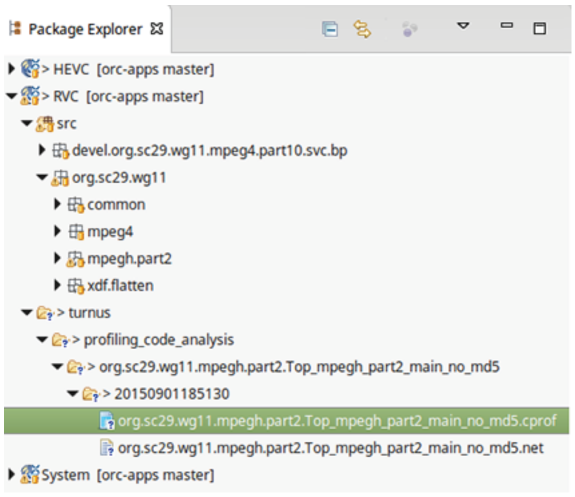
Option	Description
ORCC Project	The CAL project which contains the design under test.
XDF	The .xdf file which contains the top network description of the design under test.
Versioner	The versioner used for versioning the CAL files of the design under test. By default, a Git versioner is used, which collects the following information: the Git commit identifier, the Git repository and the last modification date of each .cal file used in the design.

After running the analysis, a report file is generated containing all the profiling information (see Figure 50). This file is located in the project path under

<project>/turnus/profiling_code_analysis/<design_name>/<current_date>/<xdf_name>.cprof.

In this case, results are stored under

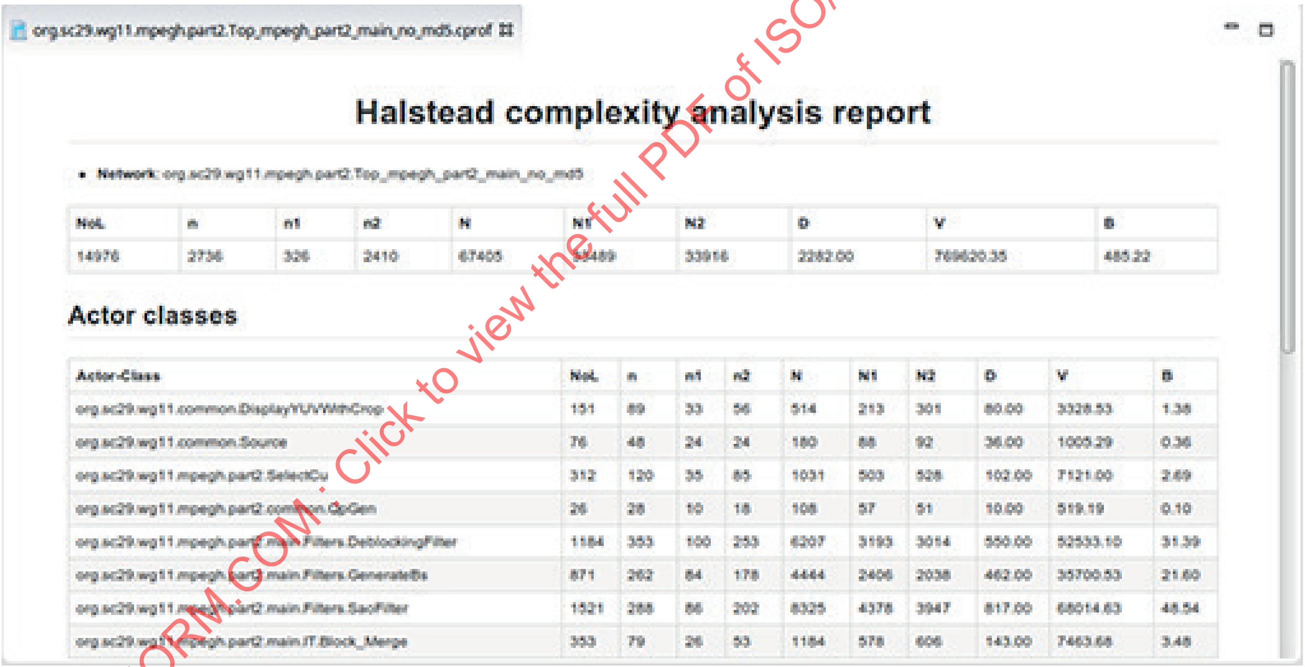
RVC/turnus/profiling_code_analysis/org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main_no_md5/20150901185130/org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main_no_md5.cprof.



```
Package Explorer
▸ > HEVC [orc-apps master]
▾ > RVC [orc-apps master]
  ▾ src
    ▸ devel.org.sc29.wg11.mpeg4.part10.svc.bp
    ▾ org.sc29.wg11
      ▸ common
      ▸ mpeg4
      ▸ mpegh.part2
      ▸ xdf.flatten
  ▾ > turnus
    ▾ > profiling_code_analysis
      ▾ > org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main_no_md5
        ▾ > 20150901185130
            org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main_no_md5.cprof
            org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main_no_md5.net
▸ System [orc-apps master]
```

Figure 50 — Static code analysis data path (.cprof file)

The .cprof file can be opened with the TURNUS report viewer where a summary of the analysis data is available (see Figure 51).



org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main_no_md5.cprof

Halstead complexity analysis report

- **Network:** org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main_no_md5

NoL	n	n1	n2	N	N1	N2	D	V	B
14976	2736	326	2410	67405	33489	33916	2282.00	769620.35	485.22

Actor classes

Actor-Class	NoL	n	n1	n2	N	N1	N2	D	V	B
org.sc29.wg11.common.DisplayYUVWithCrop	151	89	33	56	514	213	301	80.00	3328.53	1.38
org.sc29.wg11.common.Source	76	48	24	24	180	88	92	36.00	1005.29	0.36
org.sc29.wg11.mpegh.part2.SelectCu	312	120	35	85	1031	503	528	102.00	7121.00	2.69
org.sc29.wg11.mpegh.part2.common.QpGen	26	28	10	18	108	57	51	10.00	519.19	0.10
org.sc29.wg11.mpegh.part2.main.Filters.DeblockingFilter	1184	353	100	253	6207	3193	3014	550.00	52533.10	31.39
org.sc29.wg11.mpegh.part2.main.Filters.GenerateBs	871	262	84	178	4444	2406	2038	462.00	35700.53	21.60
org.sc29.wg11.mpegh.part2.main.Filters.SaoFilter	1521	288	86	202	8325	4378	3947	817.00	68014.63	48.54
org.sc29.wg11.mpegh.part2.main.IT.Block_Merge	353	79	26	53	1184	578	606	143.00	7463.68	3.48

Figure 51 — TURNUS report viewer

The information available in the TURNUS report viewer is summarized in Table 4.

Table 4 — List of items in the TURNUS report viewer

Data	Description
NoL	Number of Lines — indicates how many lines are contained in the file (or in the network). Lines with comments are not counted.
n	Program vocabulary — defined as n = n1 + n2.
n1	The number of distinct operators.
n2	The number of distinct operands.

Table 4 *(continued)*

Data	Description
N	Program length — defined as N = N1 + N2.
N1	The total number of operators.
N2	The total number of operands.
D	The program difficulty.
V	The program volume.
B	The number of (potential) delivered bugs.

A .cprof can be exported to an .xls file. Select the .cprof file, right-click and select Export to XLS file. The exported file is fully compatible with the open-source tool available at https://www.openoffice.org (see Figure 52).

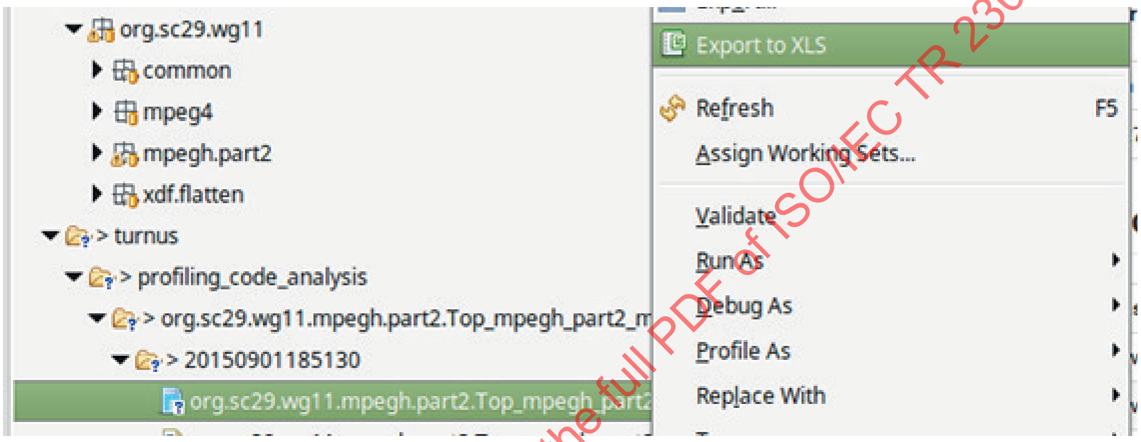
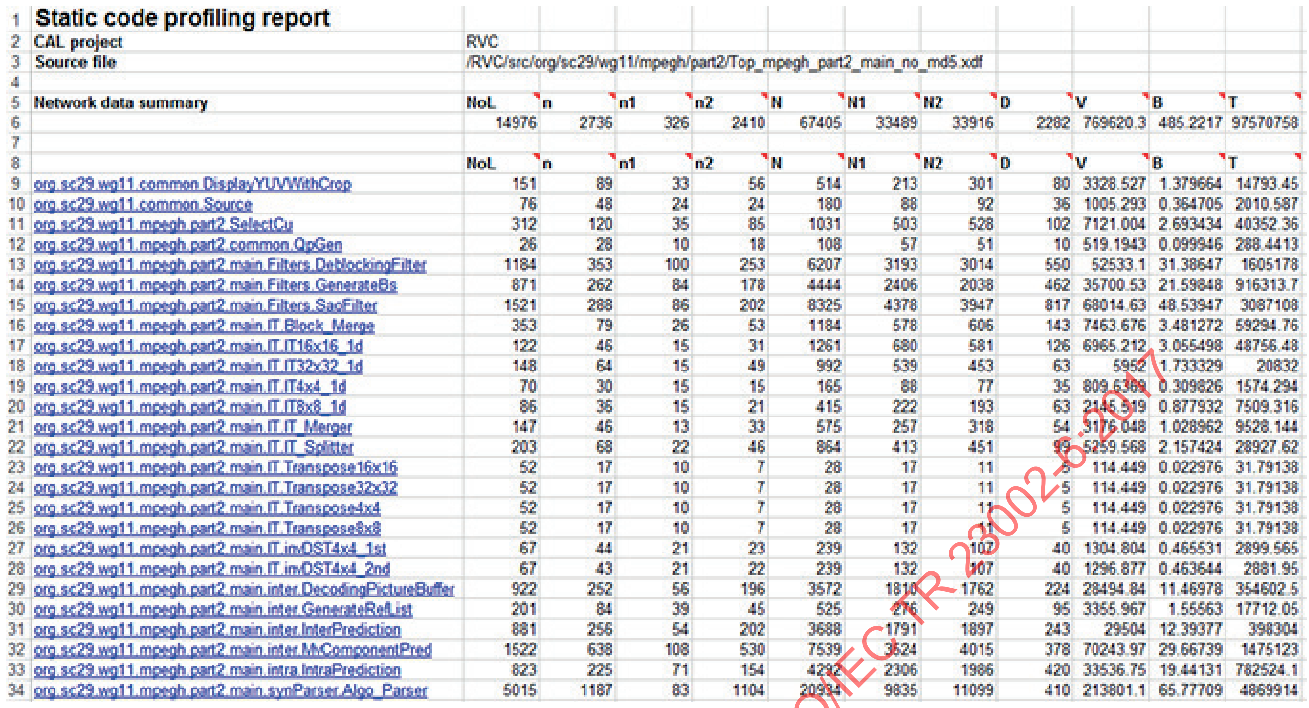


Figure 52 — Settings to export a .cprof file to an XLS file

The .xls file will appear as in Figure 53.



1	**Static code profiling report**											
2	**CAL project**	RVC										
3	**Source file**	/RVC/src/org/sc29/wg11/mpegh/part2/Top_mpegh_part2_main_no_md5.xdf										
4												
5	**Network data summary**	**NoL**	**n**	**n1**	**n2**	**N**	**N1**	**N2**	**D**	**V**	**B**	**T**
6		14976	2736	326	2410	67405	33489	33916	2282	769620.3	485.2217	97570758
7												
8		**NoL**	**n**	**n1**	**n2**	**N**	**N1**	**N2**	**D**	**V**	**B**	**T**
9	org.sc29.wg11.common.DisplayYUVWithCrop	151	89	33	56	514	213	301	80	3328.527	1.379664	14793.45
10	org.sc29.wg11.common.Source	76	48	24	24	180	88	92	36	1005.293	0.364705	2010.587
11	org.sc29.wg11.mpegh.part2.SelectCu	312	120	35	85	1031	503	528	102	7121.004	2.693434	40352.36
12	org.sc29.wg11.mpegh.part2.common.QpGen	26	28	10	18	108	57	51	10	519.1943	0.099946	288.4413
13	org.sc29.wg11.mpegh.part2.main.Filters.DeblockingFilter	1184	353	100	253	6207	3193	3014	550	52533.1	31.38647	1605178
14	org.sc29.wg11.mpegh.part2.main.Filters.GenerateBs	871	262	84	178	4444	2406	2038	462	35700.53	21.59848	916313.7
15	org.sc29.wg11.mpegh.part2.main.Filters.SaoFilter	1521	288	86	202	8325	4378	3947	817	68014.63	48.53947	3087108
16	org.sc29.wg11.mpegh.part2.main.IT.Block_Merge	353	79	26	53	1184	578	606	143	7463.676	3.481272	59294.76
17	org.sc29.wg11.mpegh.part2.main.IT.IT16x16_1d	122	46	15	31	1261	680	581	126	6965.212	3.055498	48756.48
18	org.sc29.wg11.mpegh.part2.main.IT.IT32x32_1d	148	64	15	49	992	539	453	63	5952	1.733329	20832
19	org.sc29.wg11.mpegh.part2.main.IT.IT4x4_1d	70	30	15	15	165	88	77	35	809.6369	0.309826	1574.294
20	org.sc29.wg11.mpegh.part2.main.IT.IT8x8_1d	86	36	15	21	415	222	193	63	2145.519	0.877932	7509.316
21	org.sc29.wg11.mpegh.part2.main.IT.IT_Merger	147	46	13	33	575	257	318	54	3176.048	1.028962	9528.144
22	org.sc29.wg11.mpegh.part2.main.IT.IT_Splitter	203	68	22	46	864	413	451	99	5259.568	2.157424	28927.62
23	org.sc29.wg11.mpegh.part2.main.IT.Transpose16x16	52	17	10	7	28	17	11	5	114.449	0.022976	31.79138
24	org.sc29.wg11.mpegh.part2.main.IT.Transpose32x32	52	17	10	7	28	17	11	5	114.449	0.022976	31.79138
25	org.sc29.wg11.mpegh.part2.main.IT.Transpose4x4	52	17	10	7	28	17	11	5	114.449	0.022976	31.79138
26	org.sc29.wg11.mpegh.part2.main.IT.Transpose8x8	52	17	10	7	28	17	11	5	114.449	0.022976	31.79138
27	org.sc29.wg11.mpegh.part2.main.IT.invDST4x4_1st	67	44	21	23	239	132	107	40	1304.804	0.465531	2899.565
28	org.sc29.wg11.mpegh.part2.main.IT.invDST4x4_2nd	67	43	21	22	239	132	107	40	1296.877	0.463644	2881.95
29	org.sc29.wg11.mpegh.part2.main.inter.DecodingPictureBuffer	922	252	56	196	3572	1810	1762	224	28494.84	11.46978	354602.5
30	org.sc29.wg11.mpegh.part2.main.inter.GenerateRefList	201	84	39	45	525	276	249	95	3355.967	1.55563	17712.05
31	org.sc29.wg11.mpegh.part2.main.inter.InterPrediction	881	256	54	202	3688	1791	1897	243	29504	12.39377	398304
32	org.sc29.wg11.mpegh.part2.main.inter.MvComponentPred	1522	638	108	530	7539	3524	4015	378	70243.97	29.66739	1475123
33	org.sc29.wg11.mpegh.part2.main.intra.IntraPrediction	823	225	71	154	4292	2306	1986	420	33536.75	19.44131	782524.1
34	org.sc29.wg11.mpegh.part2.main.synParser.Algo_Parser	5015	1187	83	1104	20934	9835	11099	410	213801.1	65.77709	4869914

Figure 53 — XLS file report view

For each CAL file, it is possible to obtain fine-grained information about the operand and operators that contribute to the static code complexity of the design.

7.3.5 Dynamic code programming

With the TURNUS ORCC dynamic code profiler, the design is executed and the following profiling information is retrieved.

— The number of executed actions of each actor.

— The number of firings of each action.

— The operators executed by each actor and action.

— The number of produced and consumed tokens of each actor and action.

— The number of tokens that have been passed through a buffer of the design.

— The number of peeks, read misses, write misses for each actor and action and buffer.

Furthermore, during profiling, the execution trace graph (ETG) is generated, which is used during the design space exploration and optimization stages as described in 7.3.6.

The TURNUS framework provides two different ORCC dynamic code profilers.

— A code interpreter which collects the profiling information directly from the interpretation of the ORCC intermediate representation (IR) of the design.

— A profiled code generation and execution which generate a C++ representation of the design incorporating some profiling directives automatically captured by the TURNUS CAL code profiler during the binary execution.

The first approach (called TURNUS ORCC dynamic interpretation) is platform-independent and it is the most accurate way of profiling a CAL program. With the second one (called TURNUS ORCC dynamic execution), the profiling information can be biased by low-level source code optimization. However, with both approaches, the generated ETGs are identical (i.e. using the same design and the same input sequence).

7.3.6 TURNUS ORCC dynamic interpreter profiler

7.3.6.1 General

To run this analysis, select and right-click on the .xdf file then select Profile as > TURNUS dynamic interpretation analysis (see Figure 54).

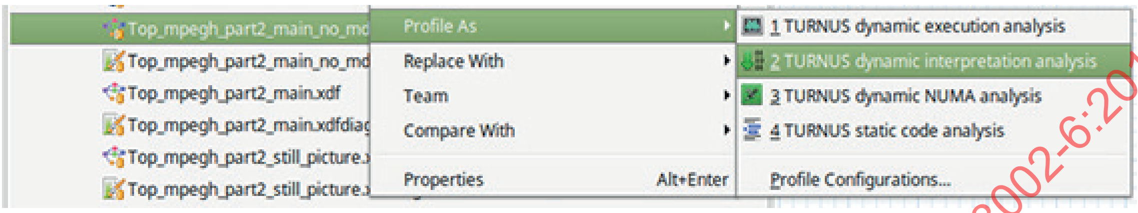
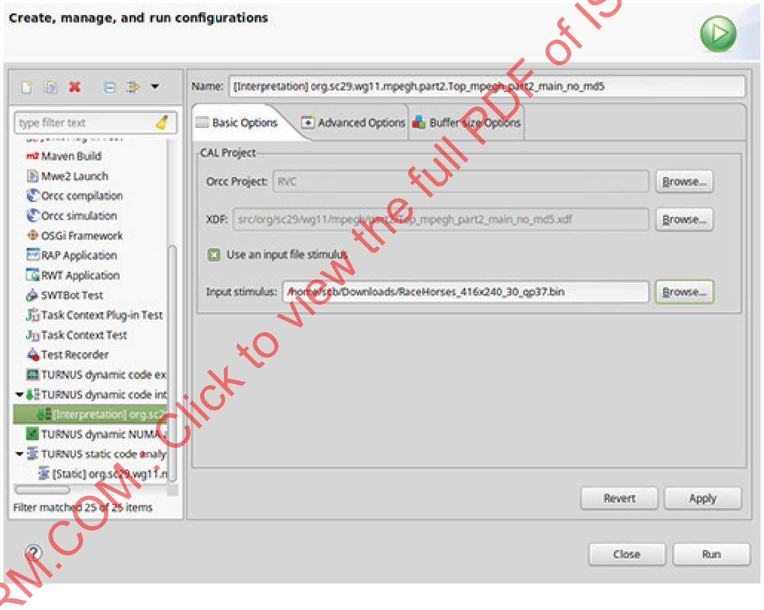

Top_mpegh_part2_main_no_md
Top_mpegh_part2_main_no_md
Top_mpegh_part2_main.xdf
Top_mpegh_part2_main.xdfdiag
Top_mpegh_part2_still_picture.
Top_mpegh_part2_still_picture.
Profile As
Replace With
Team
Compare With
Properties
Alt+Enter
1 TURNUS dynamic execution analysis
2 TURNUS dynamic interpretation analysis
3 TURNUS dynamic NUMA analysis
4 TURNUS static code analysis
Profile Configurations...

Figure 54 — Setting of the dynamic interpretation analysis in TURNUS

Figure 55 displays configuration options while Table 5 provides a list of configuration options for dynamic interpretation analysis in TURNUS.


Name: [Interpretation] org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main_no_md5

CAL Project

Orcc Project: RVC

XDF: src/org/sc29/wg11/mpegh/part2/Top_mpegh_part2_main_no_md5.xdf

[✓] Use an input file stimulus

Input stimulus: /home/ssb/Downloads/RaceHorses_416x240_30_qp37.bin

Figure 55 — Configuration settings of the dynamic interpretation analysis in TURNUS

Table 5 — List of configuration options for dynamic interpretation analysis in TURNUS

Option	Description
ORCC Project	The CAL project that contains the design under test.
XDF	The .xdf file that contains the top network description of the design under test.
Use an input stimulus	This checkbox should be checked if an input sequence needs to be used.
Input stimulus	This option appears when the "Use an input stimulus" option is checked. It is used to define the input file used by the network design.

The second step is to define the location of the input stimulus sequence. First of all, the Use an input file stimulus checkbox should be ticked and the input stimulus should be selected using the Browse button on the right of the Input stimulus option. The input sequence used in this example is the

RaceHorses_416x240_30_qp37.bin conformance sequence retrieved from the following link: ftp://ftp.kw.bbc.co.uk/hevc/hm-10.0-anchors/bitstreams/lp_main/.

Now the program can be executed by clicking on the Run button and the other options available in the two configuration tables can be changed: Advanced Options and Buffer size Options.

In the Advanced Options window, it is possible to specify the following configuration options (see Table 6).

Table 6 — List of advanced configuration options for dynamic interpretation analysis in TURNUS

Option	Description
Scheduler	This is used during the interpretation. By default, round robin is used.
Versioner	This is used for versioning the CAL files of the design under test. By default, a Git versioner is used, which includes the following collected information: the Git commit identifier, the Git repository and the last modification date of each .cal file used in the design.
Export the execution trace graph	This checkbox should be ticked to generate the execution trace graph file during the profiled simulation.
Compress the execution trace graph	This checkbox should be ticked to export the execution trace graph in a compressed file. If checked, the file will have a .trycez extension; otherwise, it will have .trace. Both are compatible with the analyses provided by the TURNUS framework.
Export the Gantt chart	This checkbox should be ticked to export the execution Gantt chart. If checked, a .vcd file is exported.
Stack protection	This checkbox should be ticked to enable the stack protection.
Shared variables support	This checkbox should be ticked to enable the shared variables support.
Constant folding	This checkbox should be ticked to enable the constant folding IR transformation.
Constant propagation	This checkbox should be ticked to enable the constant propagation IR transformation
Dead code elimination	This checkbox should be ticked to enable the dead code elimination IR transformation.
Expression evaluation	This checkbox should be ticked to enable the expression evaluation IR transformation.
Variable initializer	This checkbox should be ticked to enable the variable initializer IR transformation.
To N bit	This checkbox should be ticked to enable the type resize to N-bit IR transformation of all the variables.
To 32 Bit	This checkbox should be ticked to enable the type resize to 32-bit IR transformation of all the variables.

In the Buffer size Options window, it is possible to specify the following configuration options (see Table 7).

Table 7 — List of buffer size options for dynamic interpretation analysis in TURNUS

Option	Description
Default buffer size	The default buffer size (i.e. in terms of tokens) is used for each buffer when a specific size is not defined.
Buffer size	The actual buffer size (i.e. in terms of tokens) of each buffer.
Load from XDF	Load the actual buffer size configuration from the .xdf file.
Load from BXDF	Load the actual buffer size configuration from a BXDF file configuration (i.e. generated by the TURNUS buffer size analysis).
Clear all	Clear all buffer size configurations and use the default one.

During the program interpretation, a display frame is opened where the decoded frames are displayed. In case of errors or warnings, messages are printed in the Eclipse IDE output console (see Figure 56).

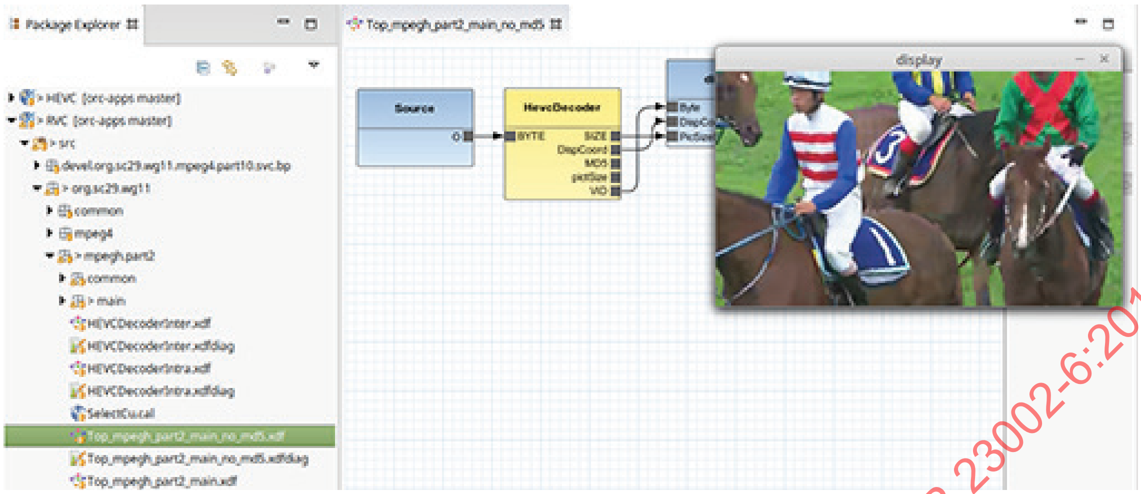


Figure 56 — Execution of the HEVC decoder project for dynamic interpretation analysis

After running the analysis, the resulting output files are located in the project path under

<project>/turnus/profiling_dynamic_analysis/<design_name>/<current_date>/.

In this case, results are stored under

RVC/turnus/profiling_dynamic_analysis/org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main_no_md5/20150901185130/.

The files contained in this folder are discussed in 7.3.6.2.

7.3.6.2 TURNUS ORCC dynamic execution profiler

To run this analysis, select and right-click the .xdf file, then select Profile as > TURNUS dynamic execution analysis (see Figure 57).

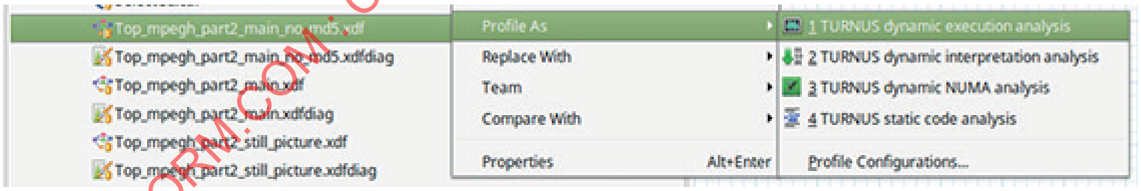
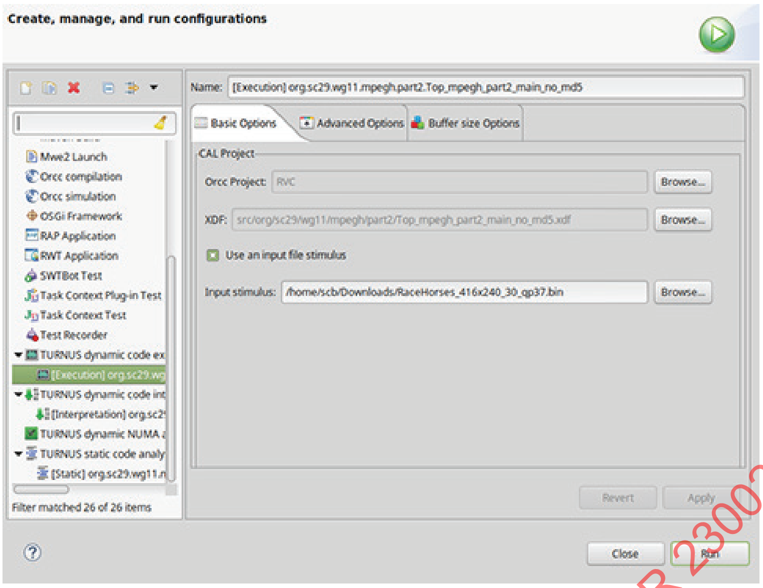

Top_mpegh_part2_main_no_md5.xdf
Top_mpegh_part2_main_no_md5.xdfdiag
Top_mpegh_part2_main.xdf
Top_mpegh_part2_main.xdfdiag
Top_mpegh_part2_still_picture.xdf
Top_mpegh_part2_still_picture.xdfdiag
Profile As
Replace With
Team
Compare With
Properties Alt+Enter
1 TURNUS dynamic execution analysis
2 TURNUS dynamic interpretation analysis
3 TURNUS dynamic NUMA analysis
4 TURNUS static code analysis
Profile Configurations...

Figure 57 — Setting of the dynamic execution analysis in TURNUS

Figure 58 displays the configuration setting, while Table 8 provides a list of configuration options for the dynamic execution analysis in TURNUS.



Name: [Execution] org.sc29.wg11.mpegh.part2.Top_mpegh_part2_main_no_md5

Orcc Project: RVC

XDF: src/org/sc29/wg11/mpegh/part2/Top_mpegh_part2_main_no_md5.xdf

☑ Use an input file stimulus

Input stimulus: /home/scb/Downloads/RaceHorses_416x240_30_qp37.bin

Figure 58 — Configuration setting of the dynamic execution analysis in TURNUS

The second step is to define the location of the input stimulus sequence. First of all, the *Use an input file stimulus* checkbox should be ticked and the *input stimulus* should be selected using the Browse button on the right of the *Input stimulus* option. The input sequence used in this example is the RaceHorses_416x240_30_qp37.bin conformance sequence retrieved from the following link: ftp://ftp.kw.bbc.co.uk/hevc/hm-10.0-anchors/bitstreams/lp_main/.

Table 8 — List of configuration options for dynamic execution analysis in TURNUS

Option	Description
ORCC Project	The CAL project that contains the design under test.
XDF	The .xdf file that contains the top network description of the design under test.
Use an input stimulus	This checkbox should be ticked if an input sequence needs to be used.
Input stimulus	This option appears when the “Use an input stimulus” option is checked. It is used to define the input file used by the network design.

The program can be executed by clicking the Run button and the other options available in the two configuration tables can be changed: Advanced Options and Buffer size Options.

In the Advanced Options configuration window, it is possible to specify the following options (see Table 9).

Table 9 — List of advanced configuration options for dynamic execution analysis in TURNUS

Option	Description
Versioner	This is used for versioning the CAL files of the design under test. By default, a Git versioner is used, which includes the following collected information: the Git commit identifier, the Git repository and the last modification date of each .cal file used in the design.
Export the execution trace graph	This checkbox should be ticked to generate the execution trace graph file during the profiled simulation.
Compress the execution trace graph	This checkbox should be ticked to export the execution trace graph in a compressed file. If checked, the file will have a .tracez extension; otherwise, it will have .trace. Both are compatible with the analyses provided by the TURNUS framework.