D3.5

Version:	1.0
Date:	2010-12-30
Dissemination status:	PU
Document reference:	D3.5

Service Model Interpreter/Simulator Engine

Project acronym:	COMPAS	
Project name:	Compliance-driven Models, Languages, and Architectures for Services	
Call and Contract:	FP7-ICT-2007-1	
Grant agreement no.:	215175	
Project Duration:	01.02.2008 – 28.02.2011 (36 months)	
Co-ordinator:	TUV	Technische Universitaet Wien (AT)
Partners:	CWI	Stichting Centrum voor Wiskunde en Informatica (NL)
	UCBL	Université Claude Bernard Lyon 1 (FR)
	USTUTT	Universitaet Stuttgart (DE)
	TILBURG UNIVERSITY	Stichting Katholieke Universiteit Brabant (NL)
	UNITN	Universita degli Studi di Trento (IT)
	TARC-PL	Telcordia Poland (PL)
	THALES	Thales Services SAS (FR)
	PWC	Pricewaterhousecoopers Accountants N.V. (NL)

This project is supported by funding from the Information Society Technologies Programme under the 7th Research Framework Programme of the European Union.

Project no. 215175

COMPAS

Compliance-driven Models, Languages, and Architectures for Services

Specific Targeted Research Project

Information Society Technologies

Start date of project: 2008-02-01 Duration: 36 months

D3.5 Service Model Interpreter/Simulator Engine

Revision 1.0

Due date of deliverable: 2010-12-31

Actual submission date: 2010-12-30

Organisation name of lead partner for this deliverable:

CWI Stichting Centrum voor Wiskunde en Informatica, NL

Contributing partner(s):

Project funded by the European Commission within the Seventh Framework Programme		
	Dissemination Level	
PU	Public	X
PP	Restricted to other program participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

History chart

Issue	Date	Changed page(s)	Cause of change	Implemented by
0.1	2010-10-28	All sections	New document	CWI
0.2	2010-12-06	Section 3-4	Content added	CWI
0.3	2010-12-09	Section 5	Content added	CWI
0.4	2010-12-10	Section 6	Content added	CWI
0.5	2010-12-14	Section 7, all sections	Conclusion and minor revision of the whole document	CWI
0.6	2010-12-28	All sections	Comments from internal COMPAS reviewers addressed	CWI
1.0	2010-12-30		Approval	TUV

Authorisation

No.	Action	Company/Name	Date
1	Prepared	CWI	2010-12-28
2	Approved	TUV	2010-12-30
3	Released	TUV	2010-12-30

Contents

List of figures

Abstract

This deliverable presents tool prototypes for the simulation of service models specified using formal notations introduced in Deliverable D3.1 [D3.1]. The presented tools are integrated to the visual environment for service description described in Deliverable D3.2 [D3.2].
One of the simulation tools uses graphical animation of control flow in a formal process model represented in a form of a Reo network and collects statistical data about its performance. This tool is useful for the evaluation of non-functional parameters of processes such as response time or throughput. Such information is relevant to guarantee process compliance by design to QoS requirements and Service Level Agreements (SLA). We also present a plug-in for integrating two existing verification toolsets to our framework. Given a graphical process model, we automatically generate specifications that are processed by mCRL2 and CADP toolsets. Among many other facilities these toolsets offer means for process simulation. We outline the main functionalities of these tools, and illustrate their application to the simulation of service models extracted from the COMPAS case studies.

1. Introduction

1.1. Purpose and scope

In this deliverable, we present the recent progress in the development of *process verification tools*, more specifically, *service model simulation engines*. These tools allow developers of service compositions to check logical correctness of dataflow in their process models at the early stages of design and evaluate expected performance before the implementation and deployment of supporting software. We give a brief description of theoretical results behind the development of these tools, outline their main functionalities and explain how they can be used in the context of the COMPAS architecture.

1.2. Document overview

The rest of this report is structured as follows. Section 2 shows the place of service simulation tools in the overall COMPAS architecture. Section 3 describes updates in the process verification environment with respect to the Deliverable D3.2 [D3.2]. Section 4 introduces two external verification toolsets, mCRL2 and CADP, that we integrated into our verification environment. Section 5 presents service model simulation tools which are the main objective of this deliverable. Section 6 illustrates the application of some key functional elements of the updated process verification environment to the analysis of examples extracted from the COMPAS case studies.

1.3. Definitions and glossary

The most important terminology concerning the COMPAS project is listed on the public COMPAS Web-Site [D7.1] available at http://www.compas-ict.eu section terminology. This helps to make the overall COMPAS approach more comprehensive for the general public.

In the following the definitions of terms valid only in the scope of this deliverable and therefore not listed on the public COMPAS Web-Site [D7.1] are specified:

Service: an abstract entity consisting of a set of capabilities offered by one or more providers to consumers.

Reo process model: a business process model represented as a Reo circuit (also called network or connector).

1.4. Abbreviations and acronyms

Abbreviation	**Complete Term**
ACA	Action Constraint Automata
BCG	Binary Coded Graph
BPEL	Business Process Execution Language
CA	Constraint Automata
CADP	Construction and Analysis of Distributed Processes
ECT	Eclipse Coordination Tools
LPS	Linear Process Specification
LTL	Liner Temporal Logic
LTS	Labeled Transition System
mCRL2	micro Common Representation Language 2
OCIS	Open/Caesar Interactive Simulator
SLA	Service Level Agreement
WP	Work Package

2. Tool overview

This section shows the place of the presented prototypes in the overall COMPAS architecture.

Service simulation engines are part of *process verification tools* which constitute the *process verification environment* of the overall COMPAS architecture (see Figure 1). *Service simulation engines* complement *process verification environment* with additional functionality for visualization of process execution traces and analyzing compliance requirements that pose performance constraints on the delivered services.

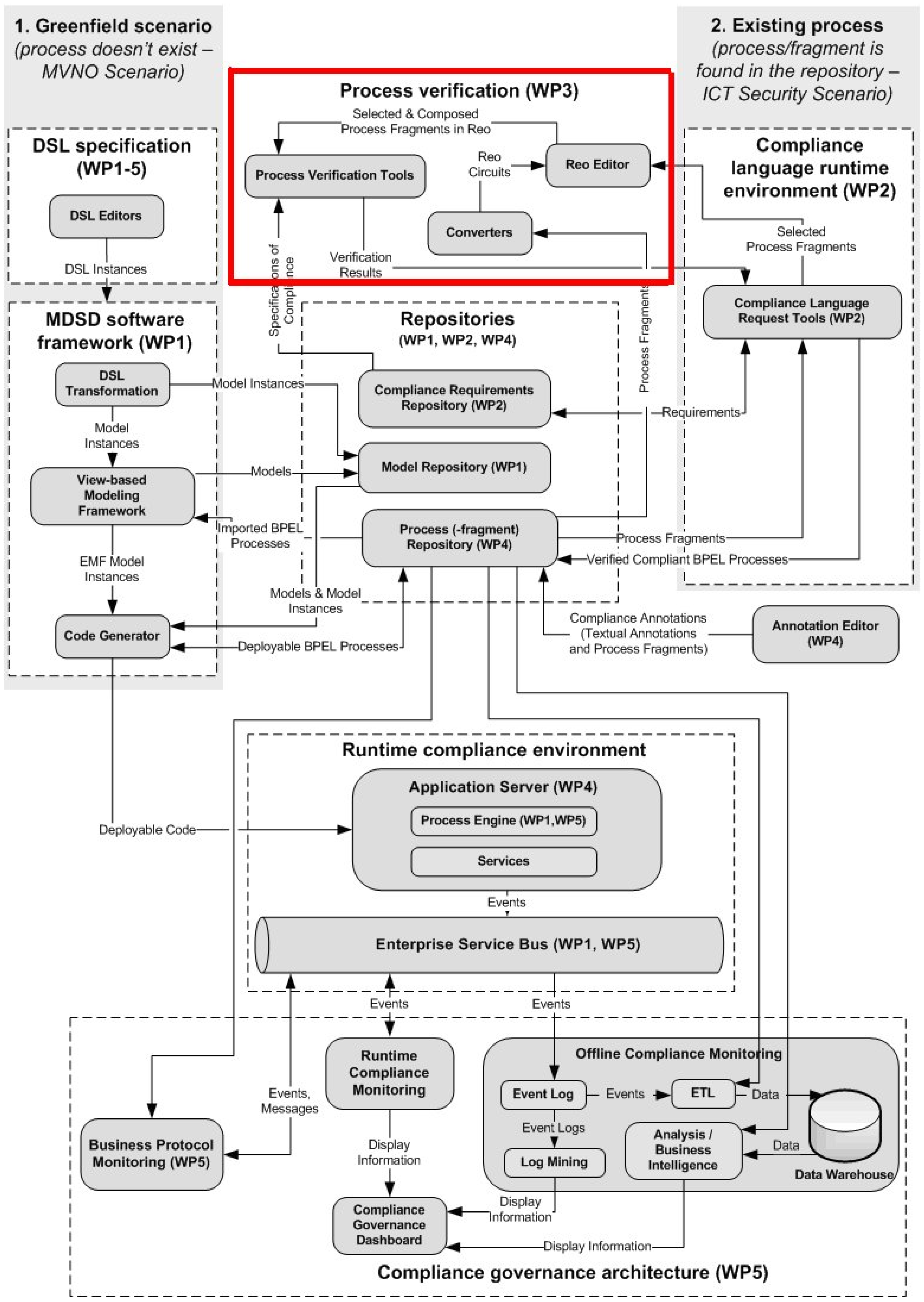

1. Greenfield scenario
(process doesn't exist – MVNO Scenario)
2. Existing process
(process/fragment is found in the repository – ICT Security Scenario)
Process verification (WP3)
Selected & Composed Process Fragments in Reo
Process Verification Tools
Reo Circuits
Reo Editor
Converters
Verification Results
Specifications of Compliance
Process Fragments
DSL specification (WP1-5)
DSL Editors
DSL Instances
MDSD software framework (WP1)
DSL Transformation
Model Instances
View-based Modeling Framework
EMF Model Instances
Code Generator
Repositories (WP1, WP2, WP4)
Compliance Requirements Repository (WP2)
Model Repository (WP1)
Process (-fragment) Repository (WP4)
Models
Imported BPEL Processes
Models & Model Instances
Deployable BPEL Processes
Compliance language runtime environment (WP2)
Selected Process Fragments
Compliance Language Request Tools (WP2)
Requirements
Verified Compliant BPEL Processes
Compliance Annotations (Textual Annotations and Process Fragments)
Annotation Editor (WP4)
Runtime compliance environment
Application Server (WP4)
Process Engine (WP1,WP5)
Services
Deployable Code
Events
Enterprise Service Bus (WP1, WP5)
Events, Messages
Business Protocol Monitoring (WP5)
Runtime Compliance Monitoring
Display Information
Compliance Governance Dashboard
Offline Compliance Monitoring
Event Log
ETL
Data
Event Logs
Log Mining
Analysis / Business Intelligence
Data Warehouse
Compliance governance architecture (WP5)

Figure 1 Overall COMPAS architecture

3. Visual Environment for Service Description

In this section, we describe updates in the visual environment for service description.

Since Deliverable D3.2 [D3.2] was released, we extended graphical Reo editor with the following functionalities:

- We re-implemented the BPEL to Reo converter and fully integrated it with the Eclipse Coordination Tools (ECT) described in Deliverable D3.2 [D3.2].
- As many compliance requirements are related to data exchanged between activities in a process, we extended graphical Reo editor with annotation tool that allows designers to specify dataflow by describing service input/output messages and defining branching conditions and functional transformations for service message exchange.
- We extended the initial set of Reo channels provided by the ECT with proper support of data manipulation primitives such as *filter*, *transformer*, *conditional synchronous drain* channels and *join* nodes. This includes their distinctive graphical notation in comparison to the previous version of the ECT and recognition by most of the analyzing tools.
- We added a possibility to create user defined channels by choosing its graphical notation and specifying semantics with the help of constraint automata (CA), other Reo connectors or components.

3.1. BPEL to Reo converter

The previous version of the tool [D3.2] generated huge output models for relatively small BPEL structures. For example, mapping of the widely used BPEL element *variable* led to a Reo circuit consisting of more than 20 channels, including 6 channels with asynchronous FIFO buffers that define the number of states in the model state space. For middle-size BPEL specifications containing dozens of variables, such a mapping already leads to a state explosion problem. We have resolved this issue by employing components. Thus, rather than dealing with complex connectors, we map some BPEL elements to components whose behavior is described by fairly small constraint automata. The components that correspond to basic BPEL structures such as *variable*, *receive* or *reply* operations are predefined at the design time, and then loaded and placed into a target Reo model at the runtime. Compound structures such as *sequence*, *while* and *pick* are transformed into a set of channels that glue the appropriate basic structures. In this way, we achieve a significant performance optimization compared to the previous version of the tool. More details about conversion tools and their application to the extracts from the THALES case study can be found in [CKA10].

The second update deals with the integration of the BPEL to Reo converter with the ECT. Now, for converting an arbitrary BPEL file to Reo, select it in the **Project Explorer** in the Eclipse environment with the installed Reo tools, right click and select **Conversion → BPEL to Reo.** The system will show a dialog box asking for the filename to save the output model.

3.2. Custom channels

A new connector file can be created using menu **File → New → Other → Reo → Connector.** After that one or more connectors and/or components can be added to the file using graphical **Reo Palette**. To create a new user-defined channel in a connector, choose menu **Custom** on the **Channels Palette** and draw a channel between any two nodes in the connector. The editor will ask you what kind of channel it should be. It is possible to create either a directed channel or a drain. The colours of the new custom channel can be changed in the **Appearance** tab of the channel **Properties** view. Custom channels work exactly like components. Thus, one can specify an animation definition in the **Animation** tab or set the **Type URI** property of a custom channel to point to the implementation of the channel by means of other Reo connectors, CA or Java files. If the channel is defined using another connector, make sure *writers* and *readers* are attached to it to show valid animations.

4. Integration with External Verification Tools

The tailored development of efficient verification tools requires substantial man power over an extensive time period. Due to the need to implement sophisticated optimization techniques, it may take a decade before a newly developed tool mature enough to be used for the analysis of large system specifications. Therefore, along with the development of native verification tools for Reo, we tried to find and adopt existing tools for analyzing Reo networks. The main obstacle to the application of popular model checking tools such as SPIN and NuSMV is the incompatibility between their input formats and semantics of Reo in terms of extended constraint automata. Of course, we could simulate the behavior of constraint automata describing the behavior of a Reo process model using finite state machines, but this would reduce the benefits of Reo, most notably, its compositionality, and leave unsolved some technical problems such as processing of data constraints posed by data-aware Reo channels. What we need is a tool that preserves compositional construction of process models, accept automata with multiple transition labels as input and support full-featured data analysis. mCRL2 is a verification toolset with a specification language that allows for compositional construction of distributed programs by synchronizing actions of communicating processes. It also supports algebraic data types and ideally suits our goal. Moreover, the toolset provides a tool for state-space generation which is compatible with the input format for another popular verification toolset, CADP. In the rest of this section, we briefly introduce these toolsets and explain how we integrated them with the ECT.

4.1. mCRL2 toolset

Deliverable 3.2 [D3.2] outlined our initial idea on the integration of the mCRL2 (micro Common Representation Language 2) toolset http://www.mcrl2.org/ with the ECT. During the past period, we completed this integration by

- Developing mapping rules for basic Reo primitives (channels and nodes) to the mCRL2 specification language according to the extended CA semantics for Reo.
- Extending graphical Reo editor with annotation tool that allows designers to specify expected input and output constraints as well as filter conditions and transformation functions using the mCRL2 functional language.
- Implementing the translation plug-in that automatically generates mCRL2 code for a given graphical Reo circuit.
- Formally proving the behavioral equivalence of the generated mCRL2 specifications to the initial semantics of the circuit given in terms of extended CA.
- Integrating mCRL2 utility calls to the graphical Reo editor.

Each Reo channel and each Reo node are mapped to simple mCRL2 processes behaviorally equivalent to channel and node semantics expressed in terms of extended CA. Channels are connected through nodes by synchronizing process actions which occur when data flow is observed on channel/node ends. Actions in mCRL2 represent atomic events and can be parameterized with data and synchronized using the synchronization operator |. Processes are defined by process expressions, which are compositions of actions using a number of operators. These operators include:

- *alternative composition* $\mathbf{p + q}$,
- *sequential composition* $\mathbf{p \cdot q}$,
- *conditional operator* or if-then-else construct $\mathbf{c \rightarrow p \lozenge q}$, where c is a boolean expression,
- *summation* $\mathbf{\Sigma_{d:D} (p)}$ used to quantify over a data domain D,
- *at* operator **a@t** indicating that the action **a** happens at time **t**.

A distributed system can be specified using a number of processes and the following operators:

- *parallel composition* $\mathbf{p \parallel q}$ yielding interleaving of the actions in **p** and **q**,
- *encapsulation* $\mathbf{\partial_H(p)}$, where **H** is a set of action names that are not allowed to occur,
- *renaming operator* $\mathbf{\rho_R(p)}$, where **R** is a renaming of the form $\mathbf{a \rightarrow b}$,
- *communication* operator $\mathbf{\Gamma_C(p)}$, where **C** is a set of communications of the form $\mathbf{a_0 \mid \ldots \mid a_n \rightarrow c}$, which means that every group of actions $\mathbf{a_0 \mid \ldots \mid a_n}$ is replaced by **c**.

We employ the mCRL2 toolset to generate state spaces for graphical Reo circuits and further model-check them. The mCRL2 language provides a number of built-in data types (e.g., **Bool**, **Nat**, **Int**) with predefined standard arithmetic operations and a data type definition mechanism to declare custom types (called also **sorts**). A global custom sort **Data** and the mCRL2 summation operator are used to model the input data domain and iterate over it while specifying service input/output constraints and data constraints and transformation functions imposed by channels.

The mCRL2 models for Reo circuits are generated in the following way: observable actions (i.e., dataflow on the channel ends in the basic CA model) are represented as atomic actions, while data items observed at these ports are modeled as parameters of these actions. Analogously, we introduce a process for every node and actions for all channel ends meeting at the node. Several variants of Reo to mCRL2 translations have been developed: a model that relies on CA [KKV10a], a model that relies on Timed CA [KKV10b], a model that relies on coloring semantics for Reo and deals with context-depend channels [KKV10c], and a model that relies on Action Constraint Automata (ACA) [KCA10] with four types of actions for port blocking and sequential dataflow modeling within synchronous regions of the circuit.

Using the conversion plug-in of the ECT, the mCRL2 script can be obtained automatically for any Reo circuit by selecting the **mCRL2** tab in the channel/connector **Properties** view of the ECT. In this view, a user can switch between several options such as **with components**, which means that the generated mCRL2 code will define processes for components/services that provide data for connector input ports and consume data from their output ports, **with data** option, which indicates whether encoding is data-aware, or **with colours** option, which uses special encoding to deal with context-dependent channels. Alternatively, **blocking encoding** can be selected, to switch between mappings based on constraint automata or on action constraint automata. For more details on the latter encoding, see Section 6.3.

One also has an option to choose an optimization strategy for the state space generation by choosing a network traversal type. The most straightforward approach (**traversal**: **none)**, is to describe any circuit as a parallel composition of all involved channels and nodes. However, a state explosion may occur since many nondeterministic unfoldings with unmatched internal actions will be generated at intermediate stages that will be considered unreachable in the final model. To overcome this problem, we create a process for a Reo circuit consequently synchronizing actions corresponding to the connected channel and node ends. Thus, **depth-first traversal** means that a process graph will be traversed using the depth-first strategy, white **breadth-first traversal** means that the graph will be traversed using the breadth-first strategy, and actions representing data flow on connected channel ends will be synchronized in the corresponding order. Our experiments show that the depth-first traversal strategy yields the best performance results [KKV10c], significantly reducing the time for the state apace generation.

Sorts of data generated/accepted by components or services coordinated by Reo can be defined in the field **Datatype**. Field **Expression** can be used to set constraints on the input or output structures. For example,

sort *request(amount:Pos, salary:Pos, period:Pos)*

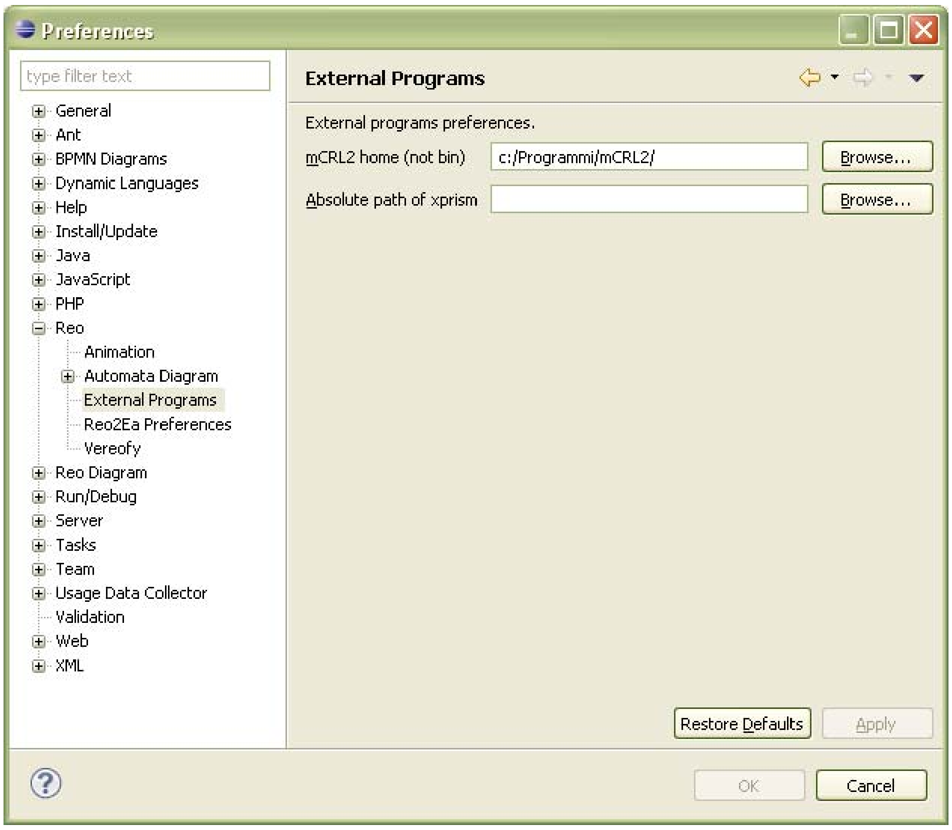
defines a data structure representing a user loan request. Here *amount* is a positive integer that refers to the requested loan amount, *salary* refers to the user salary while *period* refers to the time period (number of months) the loan is requested for. An expression, e.g.,

$$(amount \leq 10) \wedge (salary \leq 5) \wedge (period = 1),$$

can be used to define the acceptable amount, salary and period ranges.

For a correct processing, datatype and expression definitions must be valid mCRL2 definitions. The details of the functional language used for data annotations can be found in the mCRL2 language reference [mCRL2-d]. The generated specification can be model-checked using the **lps2pbes** tool from the mCRL2 toolset which is invoked by pressing the **Check** button. A formula for verification is specified in the field **Formula** provided by the ECT plug-in. For finite data domains, one can visualize the process state space by generating a Labelled Transition System (LTS) which can be obtained by pressing the **Show LTS** button. The LTS can be exported as well. The detailed description of the mCRL2 utilities that can be used for process model analysis in the ECT can be found on the mCRL2 manual web pages [mCRL2-t].

The mCRL2 framework is an open source project and is distributed under GPL licence. To use the tool from the ECT environment, one has to install it from the mCRL2 download web page http://www.mcrl2.org/mcrl2/wiki/index.php/Download and indicate its home directory in the ECT menu **Preferences → Reo → External programs → mCRL2 home** (see Figure 2).



External Programs

External programs preferences.

mCRL2 home (not bin) c:/Programmi/mCRL2/

Absolute path of xprism

Figure 2 ECT integration with external verification tools

4.2. CADP toolset

CADP http://www.inrialpes.fr/vasy/cadp/ stands for the *Construction and Analysis of Distributed Processes* and represents a toolbox for the design of communication protocols and distributed systems. CADP is developed at INRIA Rhone-Alpes and is connected to various complementary tools. CADP offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking and has been used in many industrial projects. In the nutshell, the toolset provides the following functionalities:

- Compilers for several input formalisms (e.g. LOTOS, finite state machines running in parallel and synchronized together using process algebra operators or synchronization vectors).
- Several equivalence checking tools (minimization and comparisons modulo bisimulation relations), such as BCG_MIN and BISIMULATOR.
- Several model-checkers for various temporal logic and µ-calculus, such as EVALUATOR and XTL.
- Several verification algorithms combined together: enumerative verification, on-the-fly verification, symbolic verification, compositional minimization, partial order reduction, distributed model checking, etc.
- Advanced tools for process visual checking and performance evaluation.

CADP is designed in a modular way and puts the emphasis on intermediate formats and programming interfaces such as the BCG (Binary Coded Graph) and OPEN/CAESAR software environments which allow the CADP tools to be combined with other tools and adapted to various specification languages. In fact, we employed several CADP tools for the analysis of Reo process models by converting LTS specifications obtained from the mCRL2 code into the BCG format.

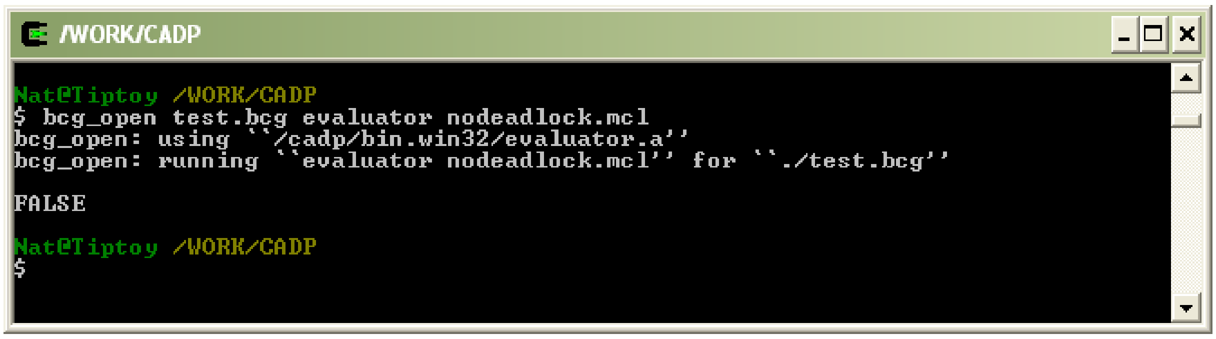
CADP provides powerful tools suitable for the verification of rather large process models. The main disadvantages of the toolset is its license and complicated installation procedure. To use CADP as part of the ECT, one has to obtain and install the toolset following the instructions from the CADP project web site http://www.inrialpes.fr/vasy/cadp/registration/.

To obtain LTS specification for a given Reo network in a format processable by the CADP tools, generate a mCRL2 code for this network and convert it to the (LPS) specification using the mCRL2 utility **lps2lts.** After that, convert the LPS specification to the LTS specification and save it in the Aldebaran format (a file with the extension .aut, say *fileName1.aut*). An LTS in the Aldebaran format can be converted to the BCG format (a file with the extension .bcg, say *fileName2.bcg*) using the CADP tool **bcg_io**:

bcg_io *-aldebaran fileName1.aut fileName2.bcg*.

This file then can be analyzed by any CADP tool accepting input in the BCG format. For example, to verify a μ-calculus property defined in the file *formulaFile.mcl* using the EVALUATOR tool, one can run the following command:

bcg_open *fileName2.bcg* **evaluator** *formulaFile.mcl.*



```
Nat@Tiptoy /WORK/CADP
$ bcg_open test.bcg evaluator nodeadlock.mcl
bcg_open: using ``/cadp/bin.win32/evaluator.a''
bcg_open: running ``evaluator nodeadlock.mcl'' for ``./test.bcg''

FALSE

Nat@Tiptoy /WORK/CADP
$
```

Figure 3 Integration with the CADP: Model checking Reo networks with the EVALUATOR tool

To use EVALUATOR tool from the ECT, activate the check box **Use CADP** in the mCRL2 view of the **Properties** tab for a given connector, specify formula in the **Formula** field and press **Check** button. Note that in order to find the CADP root folder, an environment variable CADP should be specified.

Figure 3 shows the results of deadlock detection (due to a wrong branching condition in the dataflow model) for a didactic Loan Request scenario where the decision to approve or reject a client's application for a loan depends on the details of the request, say, the client's salary, the required loan amount and the period. A Reo model for this scenario is shown in Figure 4. In this model, a client's request is specified as a data item *request(amount:Pos, salary:Pos, period:Pos)* provided by a writer component. FIFO channels represent basic (atomic) activities that constitute the process. The request is denied if the salary during the loan period is half of the required loan amount, and approved if it is three times bigger than this amount. These conditions are modelled using two filter channels. Another writer represents a bank clerk (Alice or Bob) who, in principle, may process the request. However, the condition of the filter channel *filter(login, authorized)* allows only Alice to login in the system and process the request.

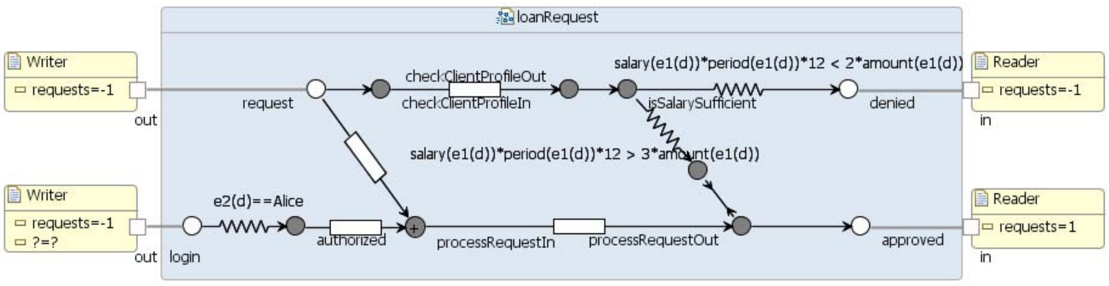


Figure 4 Dataflow model of the Loan Request case study

5. Service Model Simulation Tools

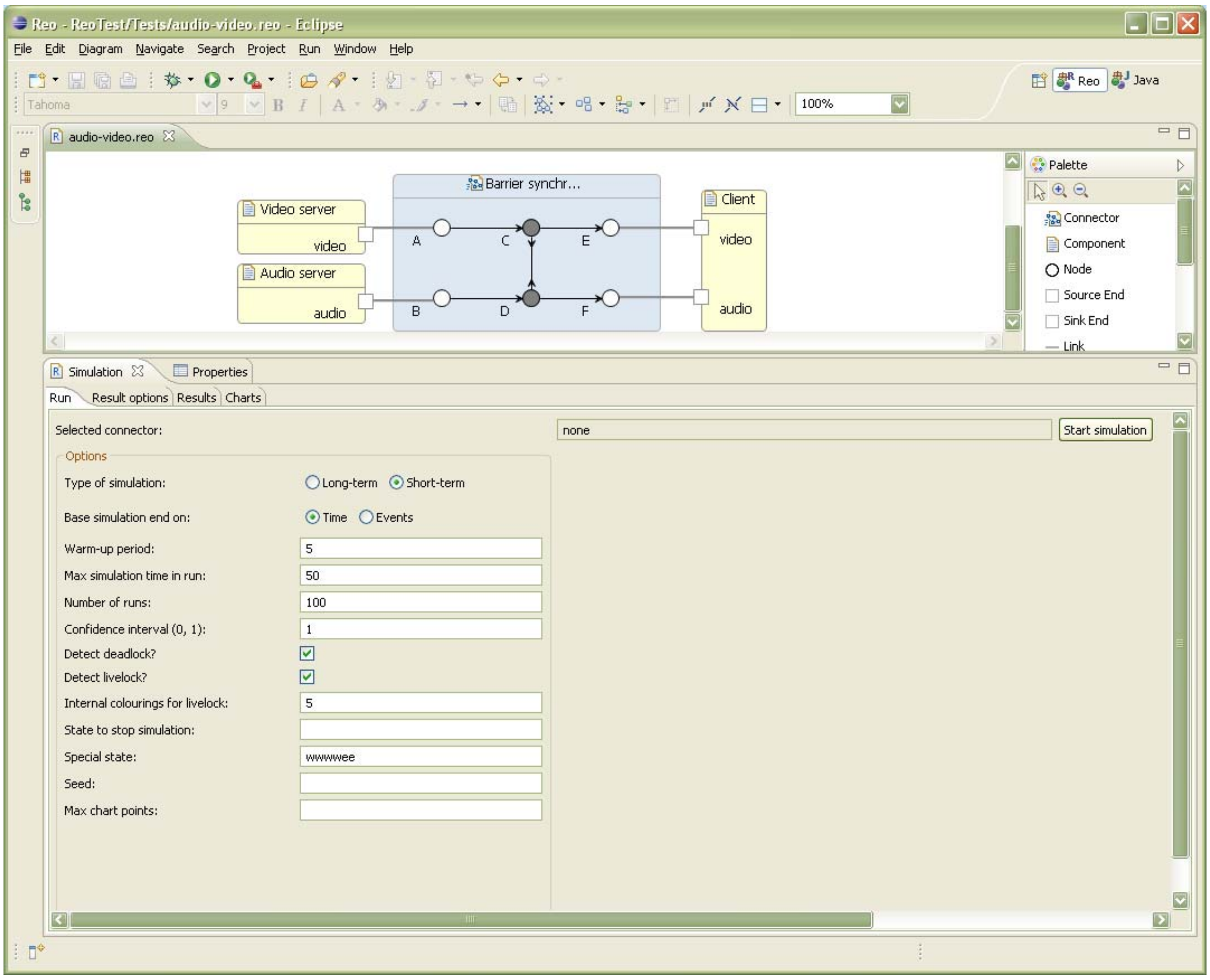
These tools are part of the ECT. For the installation instructions see the Reo project web site or Deliverable D3.2 [D3.2].

5.1. Reo simulation engine

The Reo simulation engine can perform QoS analysis of Reo process models given stochastic distributions of input and output supplied or accepted by coordinated services. It also can be used to detect logical flaws in the model control flow such as deadlocks or livelocks.

We assume that data processing by each channel yields a delay and, thus, each channel can process requests with a certain rate. The distributions of channel delays can be set by clicking on a channel and going to the **Delay** tab in the **Properties** view of the ECT. Depending on the channel type, one can specify one or two distributions. In the context of workflow modeling, channel delays can refer either to communication delays (e.g., delays in synchronous channels) or task processing delays (modeled as FIFO channels or external components/services). In the latter case, two distributions typically will be used – the first distribution describes stochastic time delays of task execution while the second distribution models the delay for disposal of the processed requests to the subsequent party.

The parameters of process simulation can be specified in the **Simulation** view available in **Reo perspective** of the ECT (see Figure 5). The tool supports *Beta*, *Binomial*, *Constant*, *Exponential*, *Gamma*, *Lognormal*, *Poisson*, *Triangular*, *Uniform* and *Weibull* distributions. For example, to specify a uniform distribution between 0 and 1, one should specify *uniform(0,1)* or *unif(0,1)* in the **Delay** channel property field. The inter-arrival rates on the boundary nodes can be specified by clicking on a boundary node and going to the **Arrivals** tab in the **Properties** view. The same distributions as for channels are supported.



Selected connector: none

Options

Type of simulation:	Long-term [✓ Short-term]
Base simulation end on:	[✓ Time] Events
Warm-up period:	5
Max simulation time in run:	50
Number of runs:	100
Confidence interval (0, 1):	1
Detect deadlock?	✓
Detect livelock?	✓
Internal colourings for livelock:	5
State to stop simulation:	
Special state:	wwwwee
Seed:	
Max chart points:	

Figure 5 Reo simulation engine

In discrete event simulation, the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system. In addition to the representation of system state variables and the logic of what happens when system events occur, discrete-event simulations normally include the following components:

- **Clock**: The simulation must keep track of the current simulation time. In discrete-event simulations, as opposed to real-time simulations, time hops because events are instantaneous the clock skips to the next event start time as the simulation proceeds.
- **Event list:** The simulation maintains at least one list of simulation events. This list sometimes is called the *pending event set* because it shows events that are pending as a result of previously simulated event but have yet to be simulated themselves. An event is described by the time at which it occurs and a type, indicating the code that will be used to simulate that event. It is common for the event code to be parameterized, in which case, the event description also contains parameters to the event code.
- **Random-number generators**: The simulation needs to generate random variables of various kinds, depending on the system model. This is accomplished by one or more pseudorandom number generators. The use of pseudorandom numbers as opposed to true random numbers is preferable because rerunning a simulation produces exactly the same behavior.

- **Statistics**: The simulation typically keeps track of the system's statistics, which quantify the aspects of interest.
- **Ending condition**: Because events are bootstrapped, theoretically a discrete event simulation could run forever and this is a task of a simulation designer to decide when to stop the simulation. Typical choices are (i) at time ***t,*** (ii) after processing ***n*** events or, more generally, (iii) when some statistical measure reaches the certain value.

The following list indicates the options for the Reo simulation plug-in:

- *Type of simulation*: long- or short-term simulation.
- *The base simulation ends on*: events or time.
- *Warm-up period*: time or number of events till the simulation starts gathering statistics.
- *Simulation length*: time or number of events till the simulation stops.
- *Number of batches*: the simulation length will be split into multiple batches to be able to give a confidence interval for the statistics. The number of batches is normally chosen between 25 and 50.
- *Confidence interval*: the accuracy of the confidence interval defined as double value between 0 and 1.
- *Detect deadlock*: if enabled, the simulation will stop whenever we are in a deadlock. If disabled, the simulation will go on so the user can see what happens with the statistics after the deadlock.
- *Detect livelock*: ability to specify if a livelock should be detected.
- *Internal colourings for livelock*: specify how many data transfers in a row without any involved boundary nodes should be chosen to indicate a livelock in a process model. The name of the option refers to the colouring semantics for Reo: each data transfer through the channel is simulated by changing its colour in the same way it is done in the **Animation** plug-in [D3.2].
- *State to stop simulation* (optional): possibility to define a certain system state in which the simulation should stop.
- *Special state* (optional): a system state to get statistics from.
- *Seed*: define a seed if you want to produce the same results in every consecutive simulation with the same parameters.
- *Max chart points*: the maximum number of chart points for the charts.

The following list indicates the possible outputs for the simulation. The tab **Results options** in the **Simulation** view allows the user to disable certain output statistics.

- *Actual loss ratio*: the ratio of requests lost in a lossy synchronous channel with respect to the total number of requests arrived to the channel.
- *Average conditional waiting time*: average waiting time of all requests which will not be blocked.
- *Average waiting time*: average waiting time over all requests.
- *Buffer utilization*: ratio of time the FIFO buffer is full.
- *Channel locked*: ratio of time a channel is locked because a colouring involving this channel is active.
- *Channel utilization*: ratio of time a channel is actually in use.
- *Colourings*: ratio of time a colouring is active.
- *End-to-end delay*: average total delay between two points.

- *Inter-arrival times*: average time between two arrivals at an ending point from a certain starting point.
- *Merger directions*: ratio of requests from a certain sink end of a node compared to the total number of requests into the node.
- *Node state*: the ratio of time a node is in a certain state.
- *Request observations*: the ratio of requests which observes the node in a certain state compared to the total number of requests arriving at the node.
- *System state*: ratio of time the system is in a certain state.
- *Special state*: ratio of time the system is in the specified special state.

To run the simulation, select a Reo network extended with the information about delay distributions and press **Start simulation**. The results of the simulation can be observed on Tabs **Results**, which shows different statistics in text format, and **Charts**, which shows graphical charts based on the collected statistical results. For example, Figure 6 shows the end-to-end delay for the synchronization in a didactic audio/video synchronization case study where the delays on boundary ports of the circuit are defined by exponential distributions with rates dA=6, dB=3, dE=5 and dF= 4, while the synchronization drain operates with the rate dC=dD=100000.

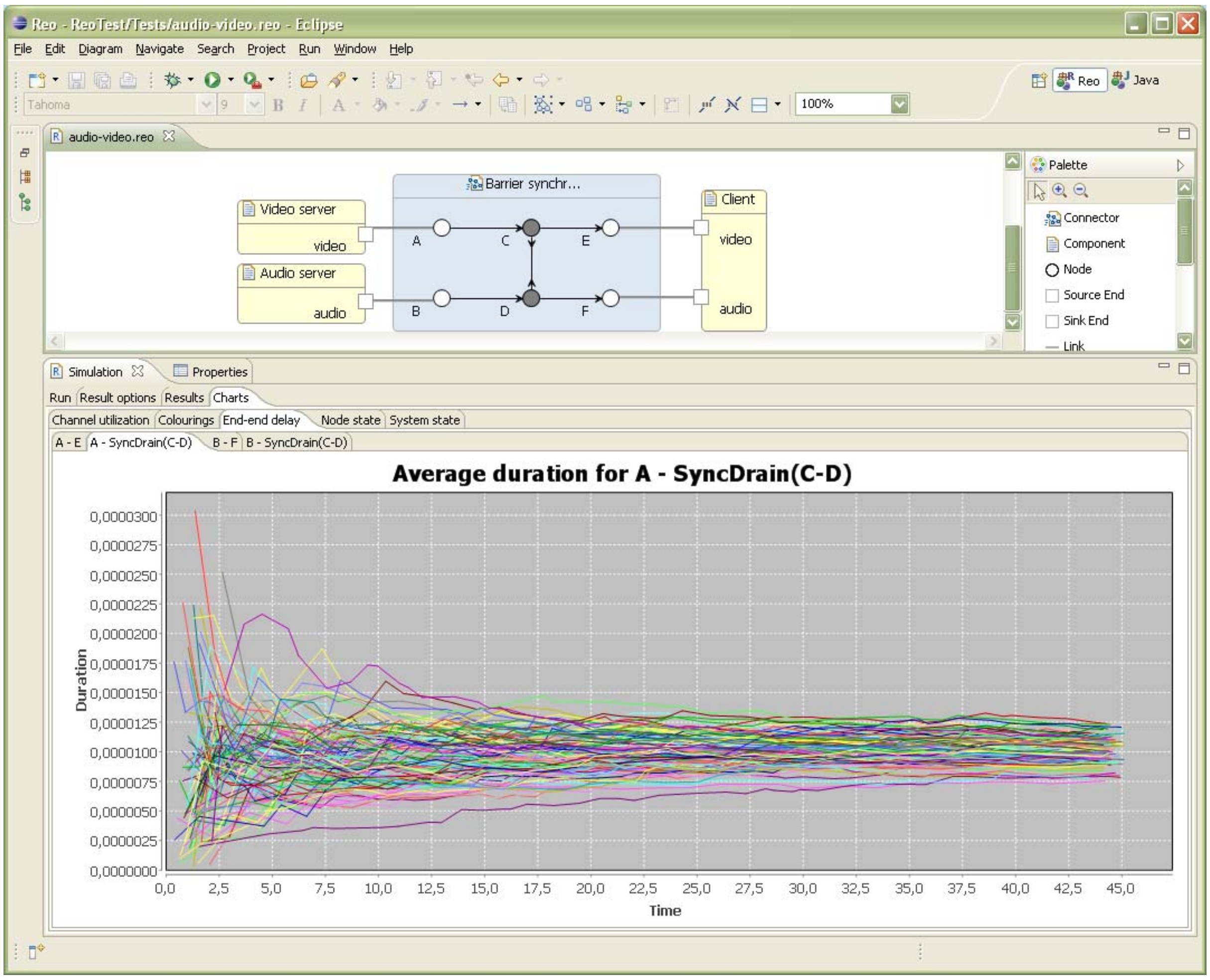


Figure 6 Process simulation statistics

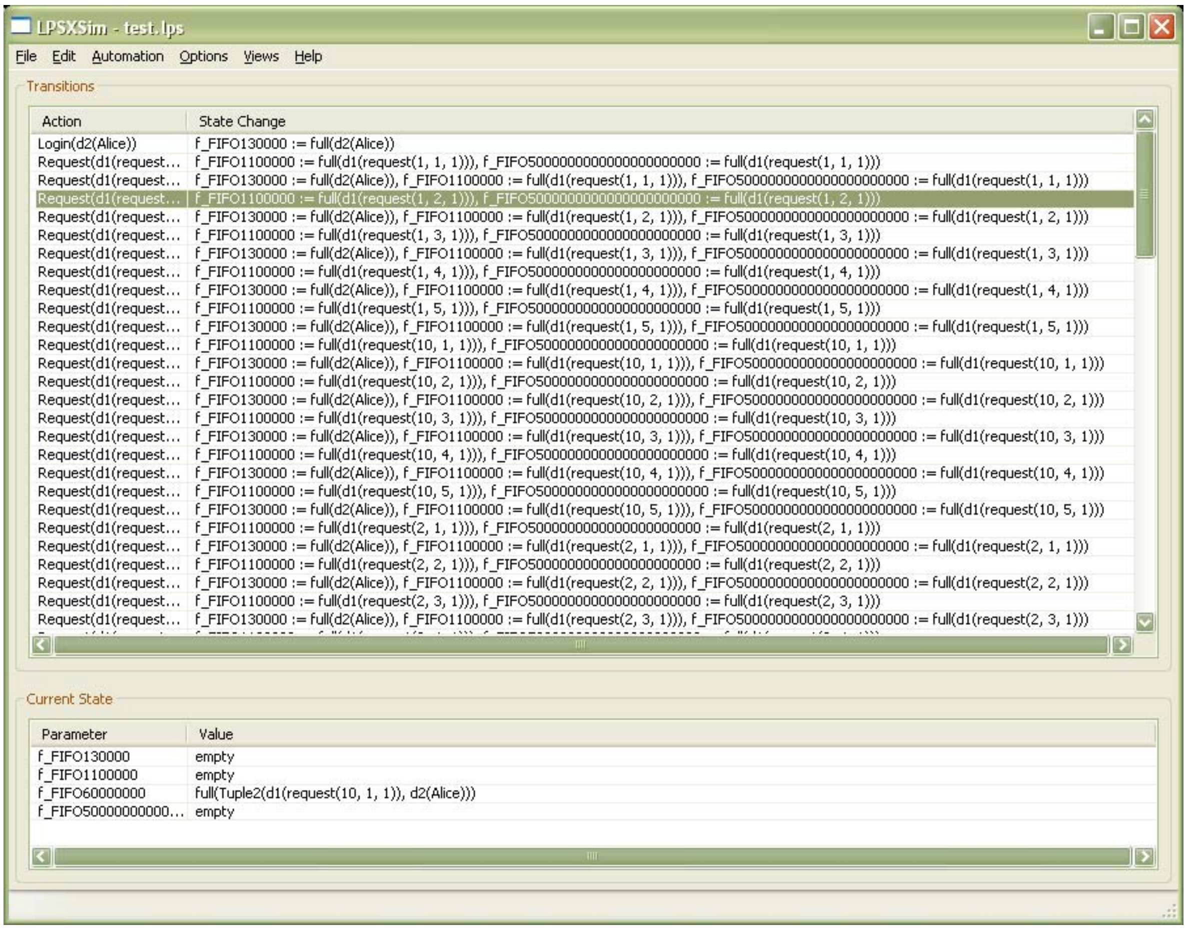
More details on the Reo simulation engine implementation and examples of its application to Reo network analysis can be found in [Kan10].

5.2. mCRL2 simulation tools

Since Reo simulation tool is based on coloring semantics (see Deliverable D3.2 [D3.2]), it is not able to properly deal with data constraints. For example, while simulating process workflow with conditional choice gateways, the tool considers all alternative braches of the process graph fairly reachable. In practice, however, some branches may be executed less frequently. Therefore, it can be important to run simulations on data-aware workflow models for more precise estimation of the process end-to-end QoS.

In the ECT we can simulate dataflow models using **lpssim** and **lpsxsim** tools from the mCRL2 toolset. The first tool provides a command-line simulation while the second tool provides graphical simulation of linear process specifications (LPS) generated by the **mcrl22lps** tool from the mCRL2 code for a graphical Reo network including those annotated with data constraints. Figure 7 shows a graphical simulation tool that shows traces for the Loan Request scenario from Figure 4. In this example, to keep the scenario state space finite and relatively small, we limited the input data domain for the loan requests using the expression

$$(amount \leq 10) \wedge (salary \leq 5) \wedge (period = 1).$$

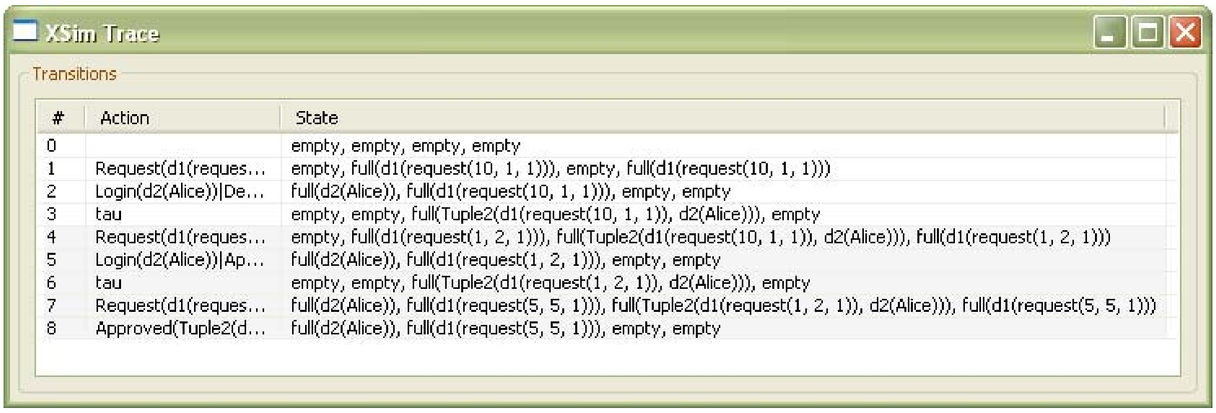


Action	State Change
Login(d2(Alice))	f_FIFO130000 := full(d2(Alice))
Request(d1(request…	f_FIFO1100000 := full(d1(request(1, 1, 1))), f_FIFO5000000000000000000000 := full(d1(request(1, 1, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(1, 1, 1))), f_FIFO5000000000000000000000 := full(d1(request(1, 1, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(1, 2, 1))), f_FIFO5000000000000000000000 := full(d1(request(1, 2, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(1, 2, 1))), f_FIFO5000000000000000000000 := full(d1(request(1, 2, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(1, 3, 1))), f_FIFO5000000000000000000000 := full(d1(request(1, 3, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(1, 3, 1))), f_FIFO5000000000000000000000 := full(d1(request(1, 3, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(1, 4, 1))), f_FIFO5000000000000000000000 := full(d1(request(1, 4, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(1, 4, 1))), f_FIFO5000000000000000000000 := full(d1(request(1, 4, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(1, 5, 1))), f_FIFO5000000000000000000000 := full(d1(request(1, 5, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(1, 5, 1))), f_FIFO5000000000000000000000 := full(d1(request(1, 5, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(10, 1, 1))), f_FIFO5000000000000000000000 := full(d1(request(10, 1, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(10, 1, 1))), f_FIFO5000000000000000000000 := full(d1(request(10, 1, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(10, 2, 1))), f_FIFO5000000000000000000000 := full(d1(request(10, 2, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(10, 2, 1))), f_FIFO5000000000000000000000 := full(d1(request(10, 2, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(10, 3, 1))), f_FIFO5000000000000000000000 := full(d1(request(10, 3, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(10, 3, 1))), f_FIFO5000000000000000000000 := full(d1(request(10, 3, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(10, 4, 1))), f_FIFO5000000000000000000000 := full(d1(request(10, 4, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(10, 4, 1))), f_FIFO5000000000000000000000 := full(d1(request(10, 4, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(10, 5, 1))), f_FIFO5000000000000000000000 := full(d1(request(10, 5, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(10, 5, 1))), f_FIFO5000000000000000000000 := full(d1(request(10, 5, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(2, 1, 1))), f_FIFO5000000000000000000000 := full(d1(request(2, 1, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(2, 1, 1))), f_FIFO5000000000000000000000 := full(d1(request(2, 1, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(2, 2, 1))), f_FIFO5000000000000000000000 := full(d1(request(2, 2, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(2, 2, 1))), f_FIFO5000000000000000000000 := full(d1(request(2, 2, 1)))
Request(d1(request…	f_FIFO1100000 := full(d1(request(2, 3, 1))), f_FIFO5000000000000000000000 := full(d1(request(2, 3, 1)))
Request(d1(request…	f_FIFO130000 := full(d2(Alice)), f_FIFO1100000 := full(d1(request(2, 3, 1))), f_FIFO5000000000000000000000 := full(d1(request(2, 3, 1)))

Parameter	Value
f_FIFO130000	empty
f_FIFO1100000	empty
f_FIFO60000000	full(Tuple2(d1(request(10, 1, 1)), d2(Alice)))
f_FIFO50000000000…	empty

Figure 7 mCRL2 simulation tool

To run the simulation, press **Automation → Play Trace** or **Random Trace** (**from Initial State** or **from Current State).** The tool will show the current state and the corresponding data parameters stored in each channel buffer. Data values on each observable state and the corresponding states of the system for a given trace can be seen by choosing menu **Views → Trace** (see Figure 8). The set of generated traces can be used to check temporal or data-related compliance requirements or evaluate the process execution time by associating time delay with each process activity.



#	Action	State
0		empty, empty, empty, empty
1	Request(d1(reques...	empty, full(d1(request(10, 1, 1))), empty, full(d1(request(10, 1, 1)))
2	Login(d2(Alice))|De...	full(d2(Alice)), full(d1(request(10, 1, 1))), empty, empty
3	tau	empty, empty, full(Tuple2(d1(request(10, 1, 1)), d2(Alice))), empty
4	Request(d1(reques...	empty, full(d1(request(1, 2, 1))), full(Tuple2(d1(request(10, 1, 1)), d2(Alice))), full(d1(request(1, 2, 1)))
5	Login(d2(Alice))|Ap...	full(d2(Alice)), full(d1(request(1, 2, 1))), empty, empty
6	tau	empty, empty, full(Tuple2(d1(request(1, 2, 1)), d2(Alice))), empty
7	Request(d1(reques...	full(d2(Alice)), full(d1(request(5, 5, 1))), full(Tuple2(d1(request(1, 2, 1)), d2(Alice))), full(d1(request(5, 5, 1)))
8	Approved(Tuple2(d...	full(d2(Alice)), full(d1(request(5, 5, 1))), empty, empty

Figure 8 mCRL2 simulation tool: trace parameters

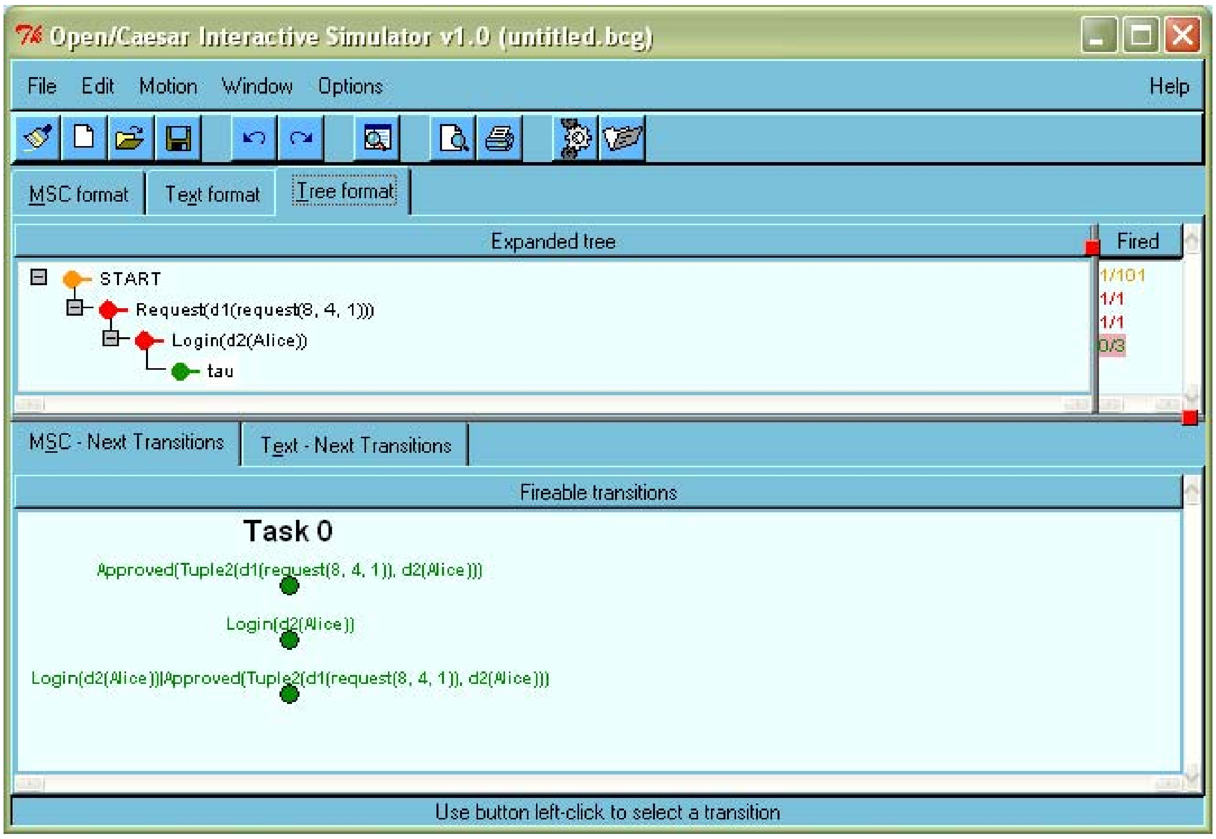
We plan to develop a tool for analyzing QoS of the process models using the mCRL2 data-aware simulator as a basis. However, in contrast to the coloring semantics, the CA semantics for Reo does not preserve the information about data flow within synchronous regions and, thus, is not suitable for accurate QoS evaluation. Therefore, we developed a model that combines properties of the coloring semantics (i.e., ability to model dataflow in synchronous regions) and CA (compositionality and data constraints). For more details on this issue see Section 3.6.

5.3. CADP simulation tools

CADP provides several interactive simulation tools. The most powerful of these tools is OCIS. OCIS stands for *Open/Caesar Interactive Simulator* and enables visualization and error detection during the design phase of systems containing parallelism and asynchronous communication. The main features of the OCIS tool include:

- *Visualization of simulation scenarios,* which are trees describing the execution paths followed by the user. Scenarios can be visualized under three different forms: execution traces, trees, and message sequence charts that reflect the communications between parallel processes, e.g., emission/reception of messages, rendezvous, diffusion, etc.
- *Manipulation of simulation scenarios,* which can be edited, saved as BCG graphs, and loaded again during another simulation session.
- *Manual and automatic navigation* in the system under simulation: when several tasks are involved, the navigation is done on the communicated automata describing the whole system. Automatic navigation is done in two ways. The first way uses the Open/Caesar tool EXHIBITOR in order to find one or more execution sequences corresponding to regular expressions searched by the user. The second way consists in re-playing some parts of sequences obtained in the former simulation sessions.

- *Source-level debugging:* The user can follow the execution of the source code and the evolution of the state variables during simulation. The user can also focus on a particular parallel task, and on the evolution of a subset of the state variables.
- *Possibility to modify the source code* and to re-compile it without quitting the current simulation session.



```
START                                   1/101
  Request(d1(request(8, 4, 1)))         1/1
    Login(d2(Alice))                    1/1
      tau                               0/3
```

```
Task 0
Approved(Tuple2(d1(request(8, 4, 1)), d2(Alice)))
Login(d2(Alice))
Login(d2(Alice))|Approved(Tuple2(d1(request(8, 4, 1)), d2(Alice)))
```

Use button left-click to select a transition

Figure 9 Simulating Loan Request scenario with the OCIS tool

To simulate a Reo network with OCIS, convert it to the BCG format as explained in Section 4.2 and execute the following command:

bcg_open *fileName.bcg* **ocis**.

Figure 9 shows the visualization of simulation scenarios with OCIS in a tree format for the Loan Request case study. One can observe that in the shown scenario the request with the *amount=8*, *salary=4* and *period=1* was submitted and Alice logged into the system. Among the possible actions at the next step are the following: (i) Alice approves the loan request, (ii) Alice logs into the system again and (iii) Alice performs both these actions simultaneously.

We believe that CADP tools can be useful for Reo network analysis in many ways. In particular, end-to-end performance evaluation of Reo process models annotated with the information about channel and node delays can be performed using CADP performance evaluation tools such as BCG_STEADY, which performs steady-state analysis, and BCG_TRANSIENT, which performs transient analysis of Continuous Time Markov Chains (CTMC) encoded in the BCG format. These tools may become a good alternative to the data-agnostic Reo simulation engine presented in Section 5.1 as they can be applied to compute throughput of data-aware process models. Currently we are working on the generation of

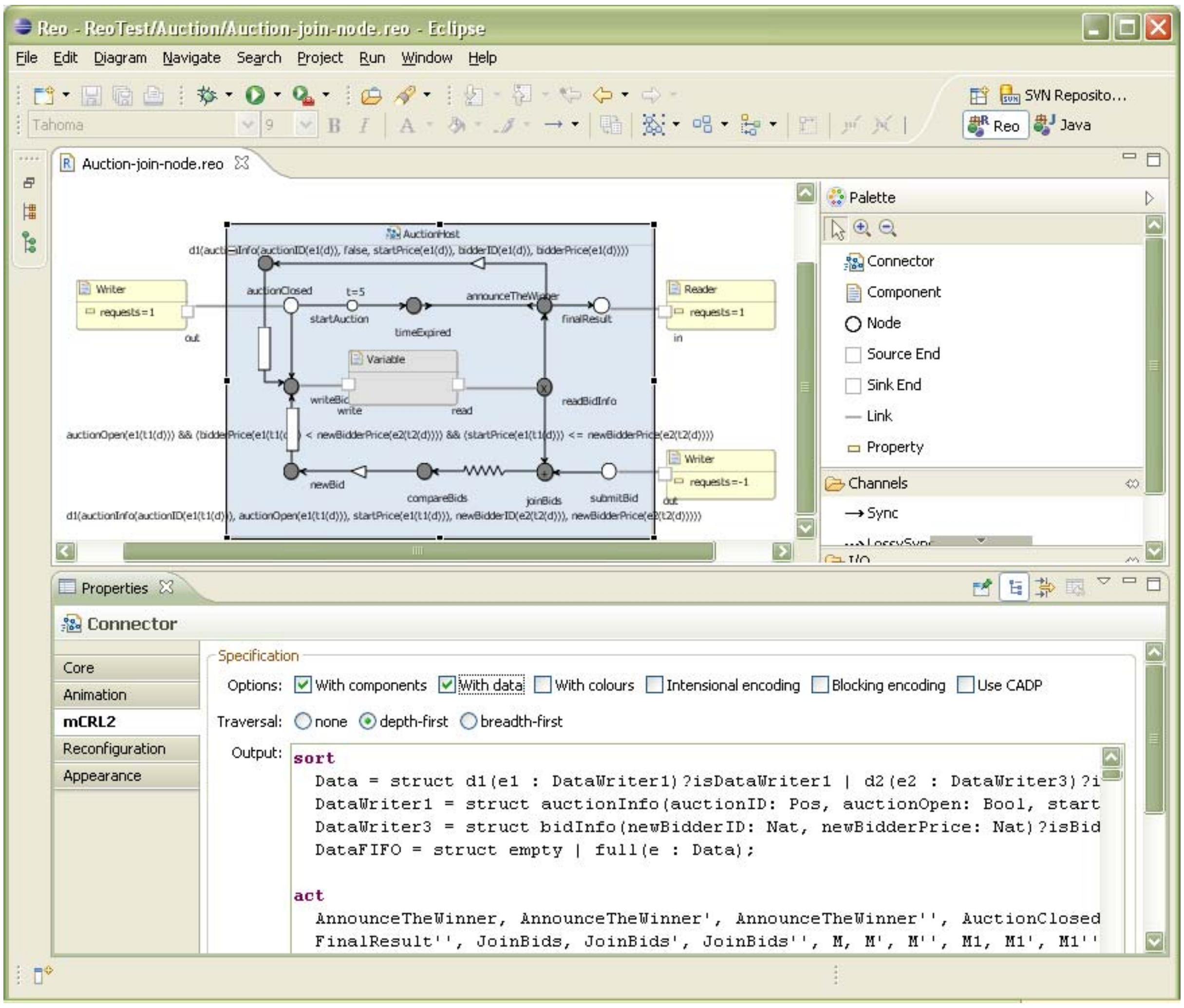
CTMC in the BCG format for QoS-annotated Reo networks. Initial work on the generation of CTMC for Reo can be found in [ACM+09].

6. Business Process Verification

In this section, we describe new functionalities of *Process verification tools* developed according to the goals set in Deliverable D3.2 or motivated by the work on the COMPAS case studies.

6.1. Model-checking process timed dataflow models

During the past period, we completed our work on modeling and verification of timed dataflow models. In particular, we included timed channels in a set of basic Reo channels and extended a conversion tool for generating mCRL2 specifications from Reo with the support of timed CA. Figure 10 shows a model for a simple auction process. In this process, a seller opens an auction that runs for a predefined period of time. During this time, an auction participant can submit bids, and if the new bid is higher than the starting price and any bid submitted previously, it is accepted as such and stored in the system. After the auction period expires, the participant who submitted the highest bid is the winner and made known to the seller of the item. Observe that a timer channel with port names *startAuction* and *timeExpired* is used to model the auction timer set to 5 time units. The aforementioned model also demonstrates the use of abstraction mechanism in Reo: we use a component *variable* to store data about the current state of the auction. This component was obtained from a Reo connector by hiding dataflow in its internal ports. We also developed an optimization for the mCRL2 code generation tool that can recognize most commonly used modeling primitives and generate optimized mCRL2 code expressing semantics of such components without processing its internal structure. More technical details about this model and timed dataflow analysis can be found in our recent papers [KKV10a][KKV10b].



Specification

Options: [✓ With components] [✓ With data] With colours Intensional encoding Blocking encoding Use CADP

Traversal: none [✓ depth-first] breadth-first

Output:

```
sort
  Data = struct d1(e1 : DataWriter1)?isDataWriter1 | d2(e2 : DataWriter3)?i
  DataWriter1 = struct auctionInfo(auctionID: Pos, auctionOpen: Bool, start
  DataWriter3 = struct bidInfo(newBidderID: Nat, newBidderPrice: Nat)?isBid
  DataFIFO = struct empty | full(e : Data);

act
  AnnounceTheWinner, AnnounceTheWinner', AnnounceTheWinner'', AuctionClosed
  FinalResult'', JoinBids, JoinBids', JoinBids'', M, M', M'', M1, M1', M1''
```

Figure 10 Simple auction process model: model-checking timed dataflow

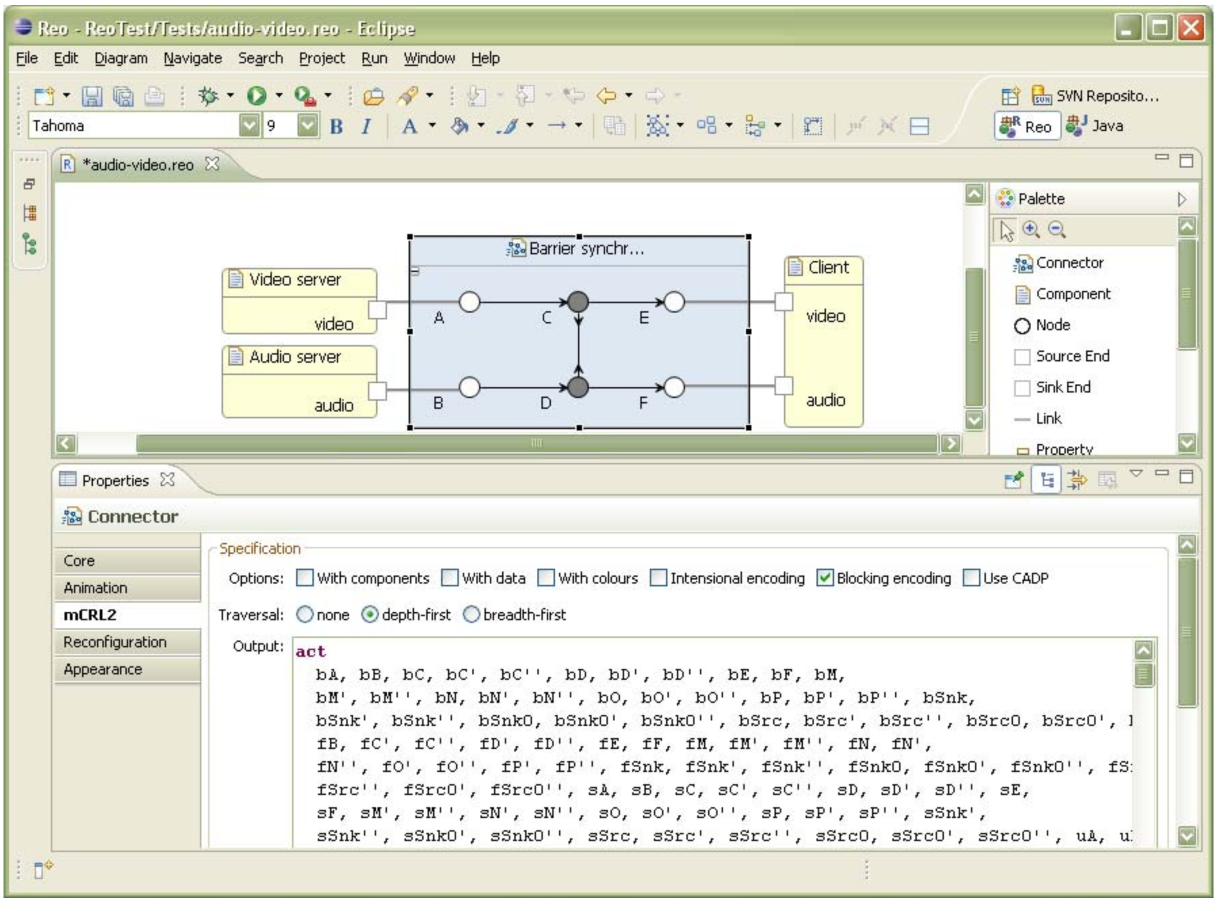
6.2. Performance evaluation: TARC-PL case study challenge

One of the possible concerns of service provider in the Watch-Me case study is the compliance of the delivered end-to-end QoS with the SLA defined by the end user. In particular, service provider would like to ensure that its application for mobile devices that synchronizes video and audio streams coming from various sources meets necessary performance requirements. We discovered that existing models for service behavior specification lack means for reasoning about real-time performance characteristics of interacting services. We started to work on a model that would eliminate this gap and become a basis for performance evaluation, implementation, runtime service replication and service-based system reconfiguration tools.

In a very simplified form, the video-audio synchronization process can be modeled using the circuit shown in Figure 11. This Reo circuit represents a so-called barrier synchronization pattern: in order to communicate successfully, all four services, two *writers* and two *readers*, should be ready to provide input data (video/audio services) and consume output data (multimedia applications on the client mobile device). Assuming that video and audio services provide input data across the network, we assign delays to synchronous Reo channels to model the corresponding delays for data transfer in the network. Given relative rates for transmitting audio and video data packages that should be synchronized, the goal is to verify that the network has sufficient throughput capacity.

In a constraint automata model (presented in D3.1), it is assumed that the QoS characteristics do not affect the behavior of a circuit and QoS labels are assigned to the transitions of the basic automata models. In our recent work [KCA10], we argued that data transfer delays are important for coordination behavior and accommodating them properly requires an appropriate formal model. *Action constraint automata* (ACA) is a new semantic model for Reo, which, in contrast to CA in their classic form, distinguishes several kinds of actions triggered on channel ports to signal the state changes of the channel. This model helped us to solve the conceptual problem with the evaluation of end-to-end delay in service-based systems modeled by means of Reo networks.

We implemented a tool that allows us automatically obtain and visualize a state space for a given Reo process model in a form of LTS according to its ACA-based semantics and apply verification tools from mCRL2 and CADP toolsets for its analysis. This model explicitly shows sequential dataflow within synchronous transactions, thus, giving more information about the state of the circuit than the original CA. Channel delays and bandwidth can be represented as labels on ACA transitions. Given such labels, the end-to-end execution time of a process run can be computed as a sum of time delays for each transition, while the guaranteed bandwidth equals to the minimal bandwidth value among all transitions. More generally, we associate functions for computing end-to-end QoS parameters for the whole process model given QoS values for each channel with the help of Q-algebras [CK07].



Specification

Options: [ ] With components [ ] With data [ ] With colours [ ] Intensional encoding [✓ Blocking encoding] [ ] Use CADP

Traversal: ( ) none [✓ depth-first] ( ) breadth-first

Output:

```
act
  bA, bB, bC, bC', bC'', bD, bD', bD'', bE, bF, bM,
  bM', bM'', bN, bN', bN'', bO, bO', bO'', bP, bP', bP'', bSnk,
  bSnk', bSnk'', bSnk0, bSnk0', bSnk0'', bSrc, bSrc', bSrc'', bSrc0, bSrc0', 
  fB, fC', fC'', fD', fD'', fE, fF, fM, fM', fM'', fN, fN',
  fN'', fO', fO'', fP', fP'', fSnk, fSnk', fSnk'', fSnk0, fSnk0', fSnk0'', fS
  fSrc'', fSrc0', fSrc0'', sA, sB, sC, sC', sC'', sD, sD', sD'', sE,
  sF, sM', sM'', sN', sN'', sO, sO', sO'', sP, sP', sP'', sSnk',
  sSnk'', sSnk0', sSnk0'', sSrc, sSrc', sSrc'', sSrc0, sSrc0', sSrc0'', uA, u
```

Figure 11 A simplified model for video-audio synchronization process

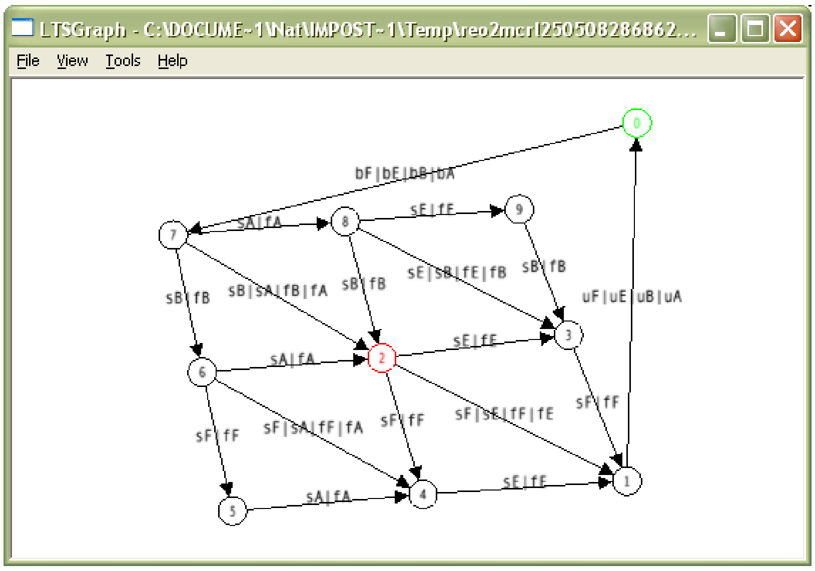


Figure 12 Semantics of the barrier synchronization circuit generated by mCRL2

In order to complete the QoS analysis of Reo process models, a timed version of ACA should be developed with the corresponding tool support. However, this work is out of the scope of the COMPAS DoW [DoW] and may form a basis for a separate project that would target at the application of Reo tools to the integration of performance-critical services.

7. Conclusions and Future Work

In this deliverable, we described a framework for the simulation of service behavioral models. The framework includes a novel tool for the simulation of graphical Reo networks annotated with performance information in the form of stochastic distributions of delays in the connector channels, and a plug-in for the integration of two powerful external tools for the simulation of distributed data-aware system specifications.

We plan to integrate ECT with other model-checking and simulation tools such as PRISM http://www.prismmodelchecker.org/, which is the leading tool for the analysis of probabilistic systems and UPPAAL http://www.it.uu.se/research/group/darts/uppaal/, which is the leading tool for the analysis of timed systems. This work may require the extension of Reo modeling environment with the support of new specialized channels (e.g., for modeling probabilistic systems) and conversion of corresponding extended CA models to the input format accepted by these tools. Using Reo as a frond-end for process specification allows us to benefit from its simplicity, graphical notation and compositionality as well as to avoid using multiple specifications of the same process for checking specific classes of system properties (timed, probabilistic, QoS) and corresponding compliance requirements.

State-of-the-art verification tools often are not able to cope with the analysis of real-world applications because of their size and complexity. We believe that our framework will help us to partially avoid this problem as we rely on compositional modeling techniques to deduce the properties of a whole system given the information about its individual components. We also plan to investigate automated abstraction mechanisms to reduce system state spaces without affecting their critical properties. This will require the extension of our models with compositional QoS calculus to evaluate the performance of a system given its abstracted behavioral models.

8. Reference documents

8.1. Internal documents

[DoW] "Description of Work" for COMPAS, 2008-02-01.

[D3.1] "Specification of a Behavioral Model for Services", 2009-01-31.

[D3.2] "Visual Environment for Service Description", 2009-07-31.

8.2. External documents

[ACM+09] Arbab, F., Chothia, T., van der Mei, R., Sun, M., Moon, Y.-J. and Verhoef, C.: "From Coordination to Stochastic Models of QoS". Proceedings of the International Conference on Coordination Models and Languages (Coordination'09). Vol. 5521 of LNCS, Springer, 2009, pages 268-287.

[CK07] Chothia, T. Kleijn, J.: "Q-Automata: Modelling the Resource Usage of Concurrent Components". Electronic Notes in Theoretical Computer Science (ENTCS), 175(2):153-167, 2007.

[CKA10] Changizi, B., Kokash, N., Arbab, F.: "A Unified Toolset for Business Process Model Formalization", International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA), ENTC, 2010, pages 147-156 (tool demonstration paper).

[Kan10] Kanters, O.: "QoS analysis by Simulation in Reo", Master Thesis, 2010, CWI.

[KCA10] Kokash, N., Changizi, B., and Arbab, F.: "A Semantic Model for Service Composition with Coordination Time Delays", International Conference on Formal Engineering Methods (ICFEM), Volume 6447 of LNCS, Springer, 2010, pages 106-121.

[KKV10a] Kokash, N., Krause C., de Vink, E.P.: "Data-Aware Design and Verification of Service Compositions with Reo and mCRL2", Proceedings of the ACM Symposium on Applied Computing, ACM Press, 2010, pages 2406-2413.

[KKV10b] Kokash, N., Krause, C., de Vink, E.P.: "Time and Data Aware Analysis of Graphical Service Models", IEEE International Conference on Software Engineering and Formal Methods (SEFM'10), IEEE Computer Society, 2010, pages 125-134.

[KKV10c] Kokash, N., Krause, C., de Vink, E.P.: "Verification of Context-Dependent Channel-Based Service Models", Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO'09), Springer, 2010.

[mCRL2-d] mCRL2 Language reference: Data types http://www.mcrl2.org/mcrl2/wiki/index.php/Language_reference/Data_types

[mCRL2-t] mCRL2 User manual: Tool manual pages http://www.mcrl2.org/mcrl2/wiki/index.php/Tool_manual_pages

[STK+10] Schumm, D., Turetken, O., Kokash, N., Elgammal, A., Leymann, F., Heuvel, W.-J.: "Business Process Compliance through Reusable Units of Compliant Processes", Current Trends in Web Engineering, Vol. 6385 of LNCS, Springer, 2010, pages 325-337.