ComMA tutorial

version 3.1 – 09-03-2021

1 Introduction

The goal of this tutorial is to get experience with interface modeling in ComMA (Component Modelling and Analysis) and using generator facilities of the ComMA tooling. First in Section 1.1 a short overview of the ComMA approach is given. As a case study we use a small vending machine example as described in Section 1.2. Resources for support are listed in Section 1.3. The structure of the remainder of this tutorial is described in Section 1.4.

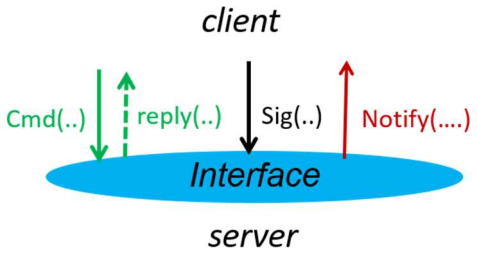
This tutorial assumes an Eclipse with ComMA plugin, including Graphviz, and requires a zip file *VendingMachine.zip* which contains a ComMA project which is the starting point for the exercises.

1.1 ComMA overview

The lack of precise and explicit specifications of component interfaces often leads to problems during the integration of components. Also updates of components might lead to system issues, e.g., caused by changes in the interaction protocol or timing behavior. At system level, it is usually difficult to detect the source of such issues. The challenge is to develop a set of tools which allows the precise definition and analysis of client-server interfaces.

The ComMA (Component Modeling and Analysis) approach is based on a hierarchy of Domain Specific Languages (DSLs). Interface specifications in ComMA consist of three main ingredients:

- The interface signature, i.e., the set of commands, signals and notifications that a server offers to its clients.
  - Commands: synchronous function calls from client to server; the server sends a reply to the client
  - Signals: asynchronous function calls from client to server; the server does not send a reply
  - Notifications: asynchronous messages from server to client; the client does not send a reply



- State machine(s) that describe the interaction protocol between client and server, i.e., the allowed sequence of commands, signals and notifications.
- Data and timing constraints on the client-server interaction, such as lower and upper bounds on response times, periodicity requirements, and constraints on parameters of subsequent events.

For an interface, the Eclipse-based ComMA tooling generates a large number of artefacts such as:

- UML diagrams of the state machine(s) and constraints. Also a document according to a company template can be generated.
- A framework to monitor whether implementations of client and server conform to the specified interface.
- Stubs and simulation possibilities.

- Scenarios for test case generation.

An important analysis tool is the monitoring framework, which allows frequent checks on interface conformance. For instance, during nightly tests and after components updates. Monitoring is based on a trace of client-server interactions, e.g., obtained via logging or sniffing. When a trace does not conform to the state machine behavior, the monitoring tool generates an error and stops monitoring. When a data or time constraint is violated, a warning is generated and monitoring continues. Monitoring also provides statistics about the specified time constraints, e.g., a histogram of the observed response times. In this tutorial, monitoring will be triggered manually, but there command-line tooling is available such that it can be invoked automatically during the test process.

An overview of the approach is shown in Figure 1.

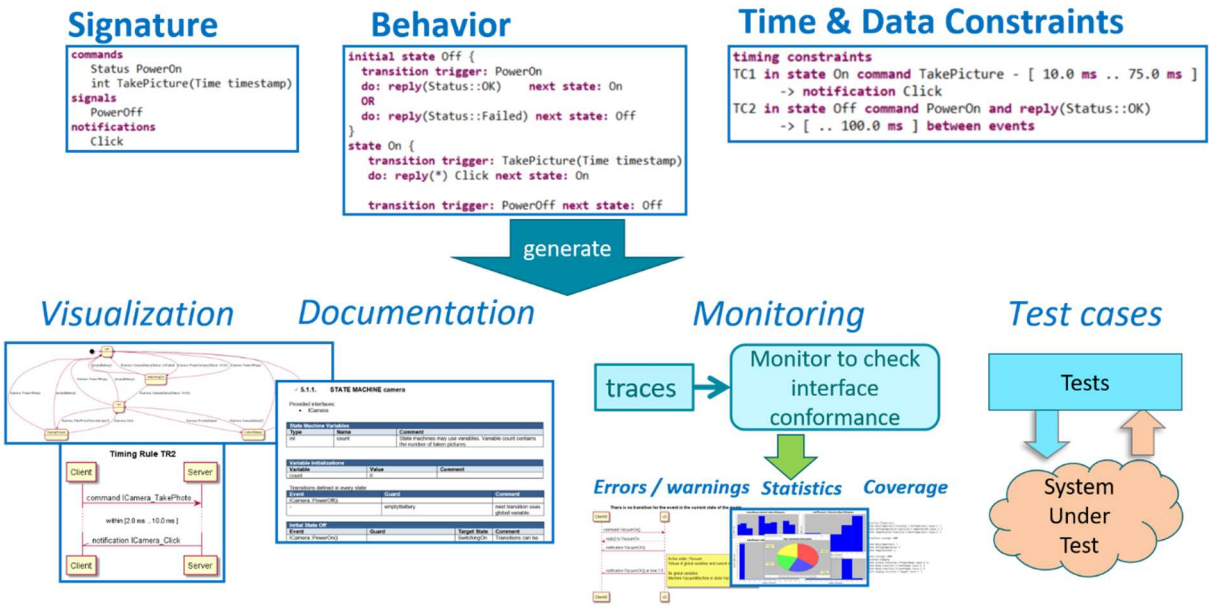


Figure 1 Overview of the ComMA approach

1.2 Case study

As a case study, we consider a simple *vending machine* implemented in Java. There is a GUI that can be used to interact with the vending machine. During the workshop, the interface *behavior* of the vending machine component will be specified using ComMA. The interaction between the GUI and the vending machine will be recorded in a trace file and using ComMA it can be checked if it conforms to the specified behavior. In addition, the tutorial addresses the generation of other artefacts such as UML diagrams and documentation.

The vending machine is a component with three interfaces, as shown in Figure 2:

- A provided interface for users to insert money, order products and get money back.
- A provided service interface to switch the machine on and off, and to load products.
- A required interface to check coins. The coin check component which provides this interface may raise an error, called CoinCheckerProblem, and it can be reset.

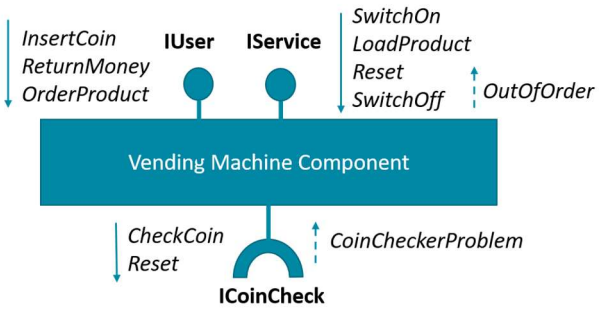


Figure 2 Interfaces of the Vending Machine component

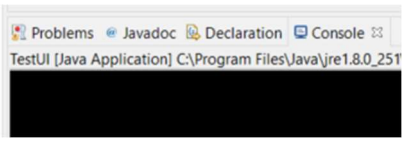
1.3 Resources for additional support

There are several resources for additional support:

- The last page of this appendix provides an overview of Eclipse commands and shortcuts. Most import is ***<CTRL>-<SPACE>*** for content assist. Also ***<CTRL>-<SHIFT>-F*** for automatic formatting might be useful.
- Together with the tutorial, a cheat sheet is distributed which provides an example of the grammar for state machines.
- The ComMA help in Eclipse: go to menu Help > Help Contents, section ComMA User Guide or use F1.
- Available example(s): File > New > Example ; see ComMA Examples.

Problem solving:

- Sometimes the Console view might become black, see picture on the right. Then right-click in the window, select Preferences. In Console view click on Background color, select white from Basic colors, OK, Apply and Close.



- In case of problems, e.g. because of caching, Project > Clean might help. Otherwise try a complete restart of Eclipse.
- Note: the Tasks view can be minimized.

1.4 Structure of the tutorial

The remainder of this tutorial is structured as follows. In Section 2 the modeling of the vending machine is prepared. Section 3 contains activities to define and check the three interfaces of the vending machine. The component and constraints on the relations between interfaces are defined in Section 4. Concluding remarks can be found in Section 5. Section 6 lists useful commands and shortcuts.

Note: this manual is suitable for the Standard version and the Philips version of ComMA. Activity 6 in Section 3.1 is different for the two versions and Activity 7 is only for the Philips version.

2 Preparation for the vending machine interfaces

This section contains a number of activities to prepare the definition of state machines for the interfaces in Section 3.

2.1 Create workspace

Create a folder "*ComMAWorkshop*" at a high level in your folder structure – a deeply nested folder leads to avoid long path names which might create problems. Start Eclipse with the created folder *ComMAWorkshop* as workspace. Import the vending machine project as follows:

- Go to menu File/Import..., open folder General and select "Existing Projects into Workspace". Hit the Next button.
- Select the "Select archive file" option and browse to select the *VendingMachine.zip* file. The project should be selected (otherwise select it manually) and click Finish

The imported project contains a *src* folder with the Java implementation and a GUI of the vending machine.

2.2 Definition of types

ComMA contains a number of basic types such as bool, int, real, and string. Additional types may be defined in files with extension *".types"*.

In the *Vending Machine* project create a file *VendingMachine.types* using File > New > File, choose the *VendingMachine* project as parent. If asked, convert the project to Xtext project.
Insert the following text (you may copy the text below):

```
/*
 * Enum with the available products
 */
enum ProductName {
	WATER
	COLA
	JUICE
}
/*
 * Result of inserting a coin
 */
enum CoinResult {
	ACCEPTED		// Coin is accepted
	NOT_ACCEPTED	// Coin is rejected
	NOT_OPERATIONAL	// Machine is not operational
}
/*
 * Type for the result of requesting a product
 */
enum OrderResult {
	DELIVERED			// Product is successfully delivered
	NOT_ENOUGH_MONEY		// The amount of inserted money is less than the
product price
	NO_PRODUCTS_AVAILABLE	// The requested product is not available
	NOT_OPERATIONAL		// The machine is not operational
}
/*
 * The result of a command
 */
enum Result {
	OK		// Command is executed successfully
	FAIL		// Command failed
	NOT_ALLOWED	// Command is not allowed
}
```

```
/*
 * Type to map products to prices
 */
pricesMap = map<ProductName,int>
```

Save the file.

2.3 Service interface

In this section the service interface is specified.

Create a folder *IService*, by right-clicking on the *VendingMachine* project, select New > Folder.

2.3.1 Service signature

Interface signatures are defined in files with extension *".signature"*. Such a file may import .types files, where the names of the types file must be different from the signature file name (to avoid potential name clashes in generated code). In folder *IService*, create a file *IService.signature* with the following contents:

```
import "../VendingMachine.types"

signature IService

commands
/* Switch the machine on
 * \return indicates if the command succeeds, fails or is not allowed
 */
Result SwitchOn
/* Load a number of products into the machine
 * \param product indicate the products to be loaded
 */
void LoadProduct(ProductName product)
/* Reset the machine after the occurrence of an error
 * \return indicates if the command succeeds, fails or is not allowed
 */
Result Reset

signals
/* Switch the machine off */
SwitchOff

notifications
/* An error occurred */
OutOfOrder
```

The doxygen-style tags in the comments will be used for document generation, as explained later.

2.3.2 Service interface state machine

A ComMA interface is specified in a file with extension *".interface"* by means of one or more state machines and additional constraints. The state machine(s) specifies the contract between client and server, i.e., the allowed sequences of interactions. A state machine describes the interface protocol from the viewpoint of server, i.e. which commands and signals can it receive in any state and which notifications it is allowed to send. Messages that are not specified are not allowed.
For the service interface specification, create a file *IService.interface* with the following contents:

```
import "IService.signature"
```

```
interface IService version "0.1"

machine serviceMachine {

	initial state Dummy {

		// dummy transition
		transition trigger: SwitchOn
			do:
			reply(Result::OK)
			next state: Dummy
	}
}
```

Note that there are warnings about unused messages from the signature. By means of the activities below, the state machine will gradually be completed for these messages.

2.4 Project file

The generator tasks are specified in the project workflow file with extension *".prj"*. Files with this extension should always be placed in the main project folder. For the vending machine, create a file *VendingMachine.prj* in the project directory with the contents:

```
import "IService/IService.interface"

Project VendingMachineProject {

}
```

Place the cursor between the two brackets and use Ctrl-space to inspect the possible generators; select *"Generate UML Block"* by double clicking.

3 Model and check the interfaces of the vending machine

The section contains activities to specify the three interfaces of the vending machine such that they conform to the implementation. Conformance is checked by the generated monitoring. Moreover a few other artifacts are generated.

3.1 Specify the service interface

The behavioral specification in the file *IService.interface* is extended by a number of activities.

Activity 1: Get familiar with the Vending Machine

- Navigate to folder *src/vendingMachine.* It contains file *TestUI.java*. Run this file as a Java application (right click and then *Run as/Java Application*).
- This leads to a GUI where buttons are commands that the machine can execute. Some commands require the selection of the result, indicated in red.
- Use the *Start* button and experiment with the machine, with a focus on the service interface.
- After pushing the *Save* button, the sequence of the executed commands and the responses is stored as an execution trace in the file *VendingMachine.events*. This file will be used later to check if the trace conforms to a behavioral specification. If the file *VendingMachine.events* is not visible in the project, right-click on the project and choose "Refresh" or F5.

Activity 2: Make an initial specification of the service interface

The goal of this activity is to create an initial version of a state machine model for the service interface. Open file *IService.interface* and modify the dummy state machine such that it matches the following description:

- Initially the vending machine only accepts the *SwitchOn* command, which leads to result OK and the machine becomes operational.
- When the machine is operational, it accepts command *LoadProduct*. Moreover it can be switched off by the *SwitchOff* signal.

Note: Ctrl-space provides templates for various types of transitions.

When ready with the state machine, right-click on file *VendingMachine.prj* and choose "Execute ComMA Workflow".

- In the lower right corner there is an indication in green while the workflow is being executed.
- When finished, the result can be inspected in folder *src-gen/uml*.
- Since the project file contains a task to generate UML, the ComMA tool generates UML state machine diagrams in the form of
  - *.png* files; note that there are reduced version of the state machine which is convenient for large state machines
  - *.plantUML* files; to show the diagram,
    - open the file (in case a pop-up occurs, select "Associate '*.plantuml' files with the current editor …")
    - open the PlantUML view; go to menu Window/Show View/Other… , expand the *PlantUML* folder and choose *PlantUML*.
      PlantUML files are displayed using Graphviz. If this does not work correctly, first restart Eclipse and try again; if this is not sufficient, add the path to the *bin* folder of Graphviz to the *path* environment variable.
  - *.graphml*; these files can be edited with, for instance, yEd. This is outside the scope of this tutorial; guidelines can be found on http://comma.esi.nl/.

Activity 3: Check an Execution Trace against the State Machine

Once the initial state machine model is defined, an execution trace like *VendingMachine.events* can be checked for conformance against this model. First create in the main project directory a folder *VendingMachineEventsFiles*. Next extend *VendingMachine.prj* with a task to generate the monitoring:

```
Generate Monitors {
	monitoringIService for interface IService {
		trace directories "VendingMachineEventsFiles"
	}
}
```

Next perform the following steps:

1. Execute again the steps from Activity 1 by just switching on and off the machine a couple of times and performing *LoadProducts*. Do not perform any other actions! These actions are recorded in file *VendingMachine.events*.

2. Rename *VendingMachine.events* to *VendingMachine1.events* (use right-click on the files > Refactor) and move it to folder *VendingMachineEventsFiles*.
3. Construct a few more .event files, including a few that do not conform to the specification, and move them also to folder *VendingMachineEventsFiles* after renaming.
4. Right-click on file *VendingMachine.prj* and choose "Run As > ComMA Generation and Monitoring". The check is executed and results are produced.
    - In order to view the results of monitoring, see folder *comma-gen/monitoringIService.* If there are no errors, coverage info can be found in file *Coverage.info*. In each state the transitions are numbered in the order of occurrence and the file describes for each transition how often it was used during monitoring. If a transition contains multiple clauses (by mean of the "OR" construct) then also the clauses are numbered. The file also report the percentage of covered transitions and states.
    - In case of errors, an overview is giving in file *ErrorSummary.txt* with an indication where to find the errors.
    - Folder *Statistics* is related to time constraints and will be explained later.
5. In case of errors, a sequence diagram of the error trace is written in a file with extension *.plantuml*. Show the content using the PlantUML view as described in Activity 2. Note that if you constructed .event files that do not conform to the specification then there is no need to adapt the model such that all errors are resolved.

Activity 4: Extend specification of service interface

1. Extend the specification in *IService.interface* with the remaining part of the signature according to the following informal description:
    - At any point in time, the vending machine may send an *OutOfOrder* notification (indicating some internal problem), going into error mode. Note that such an autonomous action of the server can be modelled as a transition without trigger and a notification in the "do"-part.
    - In error mode, the *SwitchOn* command fails and the commands to switch off and load product have no effect.
    - In error mode, the *Reset* command may bring the vending machine to the initial state where it is off. There is also a possibility that the *Reset* command fails. The *Reset* command is not allowed outside error mode.
2. Use monitoring to check the correctness of the IService specification with respect to the implementation:
    - To generate an *OutOfOrder* notification, click the *CoinChecker* Problem button of the coin checker interface.
    - After a *Reset*, select the result.

Activity 5: Add a time constraint

Interface specifications can be extended with constraints on time and data.

1. As example, add the following time constraint after the state machine in *IService.interface*:

```
timing constraints
loadProduct_reply command LoadProduct - [ 1000.0 ms .. 1100.0 ms] -> reply to
command LoadProduct
```

Note that a real is denoted in ComMA with a dot, e.g. 7.0, or exponential notation, e.g. 3.4e-5.

2. Use the GUI to create a .event file where the command *LoadProduct* is called a number of times, e.g. interleaved with other events.
3. Run the monitoring and inspect the results; note that violations to time constraints lead to warnings.
4. When a trace is checked against time constraints, statistical information is collected. The information is located in folder *comma-gen/<taskname>/statistics* in a file with extension .statistics. You can view different charts generated from files with prefix *statisticsTime* by right-clicking on it and choosing menu *Show Statistics Charts*. You can find more information about statistics in ComMA Help, section *Generation of Statistics.*

Activity 6: Documentation Generation *[for the Philips version]*

To obtain documentation, first obtain the template as follows.

- Right-click on *VendingMachine.prj* and select "Import Philips Template"; this adds a file *Template.docx*.

Next extend *VendingMachine.prj* with the following task for document generation:

```
Generate Documentations {
	documentationTask for interface IService {
		template = "Template.docx"
		DHF = 123456
		author = "John Smith"
		role = "R&D: SW Designer"

	}
}
```

Execute the ComMA workflow and next navigate to folder *src-gen/doc/* and open the generated *.docx* file. It contains information from the comments in the *.signature* and *.interface* files. In addition, the simple state machine is presented in a table form.

Activity 6: Documentation Generation *[for the Standard version]*

To obtain documentation, first obtain the template as follows.

- Right-click on *VendingMachine.prj* and select "Import Documentation Template"; this adds a file *Template.docx*.

Next extend *VendingMachine.prj* with the following task for document generation:

```
Generate Documentations {
	documentationTask for interface IService {
		template = "Template.docx"
		targetFile = "Documentation.docx"
		author = "John Smith"
		role = "R&D: SW Designer"
	}
}
```

Execute the ComMA workflow and next navigate to folder *src-gen/doc/* and open the generated *.docx* file. It contains information from the comments in the *.signature* and *.interface* files. In addition, the simple state machine is presented in a table form.

Activity 7: SSCF Generation ***[only for the Philips version of ComMA]***

Extend *VendingMachine.prj* with the following task for SSCF generation:

```
Generate CPP {
	cppTask for interface IService
}
```

- Execute the ComMA workflow and navigate to folder *src-gen/cpp11*. It contains the generated C++ proxy code following the SSCF conventions. Code for C++ 98 is generated in folder *src-gen/cpp98*.

Also CLI code can be generated. When the specification includes user defined primitive types that are not based on a ComMA primitive type, or when ComMA primitive types such as *real*, *string* or *integer* need to be mapped to custom implementation types, then a type mapping task is needed. See the ComMA help for more information.

3.2 Specify the coin checker interface

In this section the required ICoinCheck interface is specified. To avoid long execution times, the *VendingMachine.prj* file might be simplified such that it only contains the monitoring task for the IService interface.

Create a folder *ICoinCheck* in the project directory.

3.2.1 Signature coin check interface

In folder *ICoinCheck* create a file *ICoinCheck.signature* with the following contents:

```
import "../VendingMachine.types"

signature ICoinCheck

commands
CoinResult CheckCoin
Result     ResetCoinChecker

notifications
CoinCheckerProblem
```

For brevity's sake we will often omit comments.

3.2.2 Coin check interface state machine

In folder *ICoinCheck,* create a file *ICoinCheck.interface* with the following contents:

```
import "ICoinCheck.signature"

interface ICoinCheck version "0.1"

machine CoinCheckMachine {


}
```

Activity 8: State machine coin checker

Complete the state machine based on the following information:

- In the initial state only command *CheckCoin* can be accepted, which results in a reply indicating that the coin is accepted or not accepted.
- At any moment, the coin checker may generate a *CoinCheckerProblem* notification after which the coin checker is in error mode.
- The *ResetCoinChecker* command is only allowed in the error mode; it may be successful, leading to the initial state or fail (then the coin checker stays in error mode).

Activity 9: Check the specified interfaces by monitoring

Extend the project file *VendingMachine.prj* with an import of *ICoinCheck.interface.* Add a monitoring task as follows:

```
Generate Monitors {
	monitoringIService for interface IService {
		trace directories "VendingMachineEventsFiles"
	}
	monitoringICoinCheck for interface ICoinCheck {
		trace directories "VendingMachineEventsFiles"
	}
}
```

- Use the GUI to generate a few .event files using commands from the service interface and the possibility to generate an error by the coin checker interface. Observe that the Reset operation requires the selection of a result.
- Apply monitoring (right-click on *VendingMachine.prj* and choose "Run As > ComMA Generation and Monitoring") to check all *.event* files. Inspect the result in folder *comma-gen*. Improve the state machine when needed.

3.3 Specify the user interface

Next the user interface is specified. Create a folder *IUser* in the project directory.

3.3.1 Signature user interface

In folder *IUser*, create a file *IUser.signature* with the following contents:

```
import "../VendingMachine.types"

signature IUser

commands
CoinResult   InsertCoin(out int credit)
int          ReturnMoney
OrderResult  OrderProduct(ProductName prodName)
```

3.3.2 User interface state machine

In folder *IUser*, create a file *IUser.interface* which imports *IUser.signature*. Next use content assist (Ctrl-space) to obtain a skeleton of an interface definition of IUser.

Activity 10: State machine user interface

To specify the behavior of the interface, first define and initialize two variables:

- a variable of type *integer* to represent the credit of the user, initialize it to 0;

- a variable of type *pricesMap* (see *VendingMachine.types*) to represent the prices of the product, where cola costs 3, juice costs 2, and water costs 1. See the Help > Help Contents > ComMA User Guide > ComMA Languages > ComMA Statements and Expressions.

Next define the state machine of the IUser interface according to the following description:

- In the initial state, when there is no credit, only command *InsertCoin* is allowed. This command has an output parameter representing the credit when the command has been completed. This is specified as the first parameter in the reply, where the second one is of type *CoinResult*. See the ComMA User Guide in Eclipse > ComMA Languages > Interfaces > Transitions.
  InsertCoin may have three possible results:
  - The coin is accepted and the credit is increased by one.
  - The coin is not accepted and the credit is not changed.
  - The machine is not operational; in this case the value of the out parameter is not specified, that is, any value is allowed.
- When the credit is positive, *InsertCoin* is also possible, as specified above. In addition, *ReturnMoney* is allowed, returning the total amount of credit. Finally, the command *OrderProduct* is allowed, which may have a number of possible responses:
  - NOT_OPERATIONAL to indicate the machine is not operational
  - NOT_ENOUGH_MONEY to indicate that the credit is not sufficient
  - NO_PRODUCTS_AVAILABLE to indicate that the credit is sufficient, but no products are available
  - DELIVERED to indicate that the credit is sufficient and the product is delivered

Activity 11: Check the specified interfaces by monitoring

- Extend the project file *VendingMachine.prj* with an import of *IUser.interface*. Add a monitoring task for the IUser interface.
- Use the GUI to generate .event files using commands from all interfaces.
- Apply monitoring to check all .event files. Inspect the coverage, the monitoring results and improve models when needed.

Activity 12: Define time and data constraints

A few constraints on time and data are added to interface IUser.

1. Write two time constraint for the following requirements:
   a. the reply of a command to return money occurs between 1000.0 ms and 5000.0 ms
   b. when an *OrderProduct* leads to a delivered product, the reply occurs within 3.0 ms
2. Use the GUI to create a .event file where a coin is inserted many times and money is returned frequently. Similarly, construct a *.events* file where products are delivered several times.
3. Run the monitoring and inspect the results; note that violations to time constraints lead to warnings.
4. When a trace is checked against time constraints, statistical information is collected. The information is located in folder *comma-gen/<taskname>/statistics* in a file with extension *.statistics*. You can view different charts generated from files with prefix *statisticsTime* by right-clicking on it and choosing menu *Show Statistics Charts*. You can find more information about statistics in ComMA Help, section *Generation of Statistics.*

Activity 13: Define a Data Constraint

1. Add the following data constraint which expresses that when the return of money is requested, the delivered amount cannot be negative:

```
data constraints
variables
int val
returnReg command ReturnMoney;reply(val) where val >= 0
```

2. Run the monitoring and inspect the results.

4 Add component constraints

Constraints on relations between events of different interfaces can be expressed in a ComMA component specifications with extension .component. As an illustration we consider a vending machine component with interfaces as depicted in Figure 2. Create in the main project directory a file *VendingMachine.component* with the following contents:

```
import "IUser/IUser.interface"
import "IService/IService.interface"
import "ICoinCheck/ICoinCheck.interface"

component VendingMachine

provided port IUser vmUserPort
provided port IService vmServicePort
required port ICoinCheck vmCoinPort

functional constraints

OperationalConstraintInsertCoin {
	/* InsertCoin Command of IUser returns NOT_OPERATIONAL
	 * when IService is not in state Operational
	 * Otherwise it returns ACCEPTED or NOT_ACCEPTED
	 */
use events
vmUserPort::reply to command InsertCoin

initial state CoinReply {
	vmUserPort::reply (*,CoinResult::NOT_OPERATIONAL) to command InsertCoin
		where NOT vmServicePort in Operational
	      next state: CoinReply

	vmUserPort::reply (*,CoinResult::ACCEPTED) to command InsertCoin
	      where vmServicePort in Operational
	next state: CoinReply

	vmUserPort::reply (*,CoinResult::NOT_ACCEPTED) to command InsertCoin
	      where vmServicePort in Operational
	next state: CoinReply
	}
}
```

This specification assumes that interface IService has a state called *Operational* to represent that the machine is on and not in an error. Replace it with the state name of your model.

To monitor the component constraints, change the file *VendingMachine.prj* as follows:

- Import *VendingMachine.component*
- Remove the current monitoring tasks
- Add the following monitoring task:

```
Generate Monitors {
	monitoring for component VendingMachine {
		trace directories "VendingMachineEventsFiles"
	}
}
```

Component monitoring checks the components constraints and all interfaces of the component. Run monitoring and inspect the result in folder *comma-gen/monitoring*. Note that the folder contains and a summary of all errors and warnings. Moreover, the coverage file provides an overview of all component interfaces.

Activity 14: Define component constraints

- Define a functional constraint for the vending machine component which expresses that each *Reset* on the service port is followed by a *ResetCoinChecker* on the coin check port.
- Define a time constraint (after heading **`timing constraints`**) which expresses that an *CoinCheckerProblem* notification on the coin check port is followed by an *OutOfOrder* notification on the service port within 2 ms.
- Run monitoring and inspect the results.

5 Concluding remarks

As mentioned in the introduction, the ComMA tooling includes a number of additional features, such as simulation and the generation of stubs and tests. Monitoring is supported by command-line tooling that can be integrated. For instance, to perform monitoring after smoke tests automatically.

In the case study we started from an existing implementation and gradually modeled its interfaces, checking conformance by manually constructing traces and applying the generated monitoring. An alternative is to generate test cases or a test client based on the ComMA model. ComMA is also frequently applied for the definition of new interfaces, for instance of third party components. In these case, model simulation can be useful to validate the model. Stubs might support the independent development of client and server. Monitoring or test generation can be exploited to check if the implementations conform to the interface models.

6 Commands and shortcuts

6.1 Useful basic Eclipse shortcuts

- Switch between tabs:
  - <CTRL>-<Page Up>
  - <CTRL>-<Page Down>
- Maximize current editor (tab):
  - <CTRL>-<M>
  - Double click on tab
- Jump to position of last edit
  - <CTRL>-<Q>
- Save file

    - <CTRL>-S
- Split screen: open two tabs, click on one tab and drag it to the text area; then a vertical or horizontal screen outline will become visible; release the mouse for the desired lay-out.

6.2 Editing

- Content assist:
    - <CTRL>-<SPACE>
- Automatic formatting: (pretty printing)
    - <CTRL>-<SHIFT>-F

6.3 Fault detection and correction

- Revalidate all files:
    - Project -> Clean...
- Validation results:
    - error: icon: cross in a red box
    - warning: icon: exclamation mark in a yellow triangle