

AGCL user guide

Version 0.1

Ernesto Posse

June 12, 2014

Preface

This document describes the Assume/Guarantee Contracts Language (AGCL) for describing behavioural contracts of component-based models or systems, specifically models described in the Architecture Analysis and Design Language (AADL) [19, 7]. It also describes the basic usage of the AGCL plug-in for OSATE, an Eclipse-based IDE for AADL.

Acknowledgements

This work was developed with support from Edgewater Computer Systems Inc., Ontario Centres for Excellence and Connect Canada.

Contents

1	Introduction	3
2	The AGCL plug-in for OSATE	6
2.1	Installation	6
2.1.1	Prerequisites	6
2.1.2	Installing ECLIPSE dependencies and OSATE sources . .	7
2.1.3	Installing NUSMV	10
2.1.4	Installing the AGCL plug-in	11
2.2	Initial setup	11
2.3	The user interface	12
2.3.1	The full environment	12
2.3.2	Setting preferences	12
2.3.3	Performing analysis	17
3	The AGCL language	18
3.1	Contracts in AGCL	18
3.2	Kinds of analysis	19
3.2.1	An example: client-mediator-server	19
3.2.2	Contracts for atomic components (threads)	21
3.2.3	Contracts for composite components (thread groups) . .	26
3.2.4	Conformance	29
3.3	Summary	31
	Bibliography	33

Chapter 1

Introduction

One of the most common approaches for modelling and designing systems is *component-based*, where the system's design (the model) consists of a hierarchically structured collection of components, *i.e.*, each component is either a *basic* or *atomic component*, or a *composite component* which is the composition of other components (atomic or composite). The precise meaning of the atomic components, the composition and the connection of components depends on the particular language used to describe the system. One such language is the Architecture Analysis and Design Language (AADL) [7]. In AADL, components are either software components, hardware components or systems with both software and hardware. We refer the reader to [8, 9], [7], [19] and [20] for information on AADL. The rest of this document assumes some familiarity with AADL.

In most cases we are interested in *dynamic systems*, *i.e.*, systems which have some *behaviour*. Each component may have some specification of behaviour associated with it, or it may be a purely structural component which groups together a set of sub-components.

The purpose of this document is to introduce the Assume/Guarantee Contracts Language (AGCL) for describing behavioural contracts of component-based models or systems and the basic use of AGCL plug-in for OSATE, an Eclipse-based IDE for AADL. AGCL provides the means to annotate components with specifications of behaviour and contracts which describe what behaviours components can assume from their environment and prescribe what behaviour they guarantee. The AGCL plug-in provides both basic syntactic support within the OSATE IDE, and more importantly, analysis tools to automatically determine whether components (both atomic and composite) satisfy their contracts.

Composite components, connections, environments, behaviours Components may interact with other components. In message-passing systems, such interaction takes place through *connectors* or *channels* linking specific components by connecting their *ports* (connection points). The set of ports of a com-

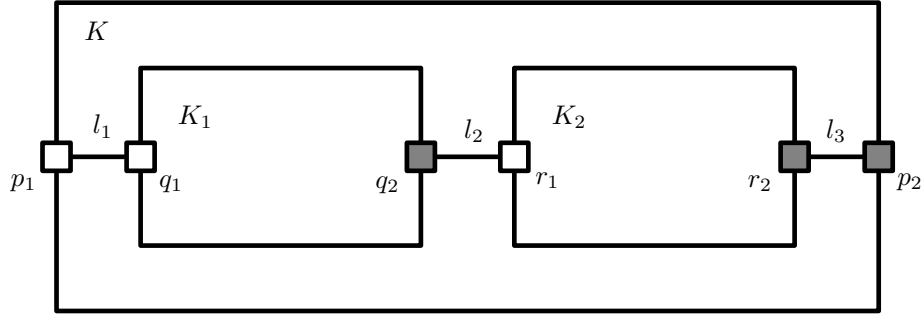


Figure 1.1: A composite component K with two subcomponents K_1 and K_2 . Each component has *ports* (p_1, p_2 for K , q_1, q_2 for K_1 and r_1, r_2 for K_2). A component is connected to another component via a *channel* or *connector* which links their respective ports. Connector l_2 links K_1 and K_2 by hooking up port q_2 of K_1 with port r_1 of K_2 .

ponent forms its *interface*. Such components can interact by sending messages (*output*) or receiving messages (*input*) through these channels. The *environment* of a component K is the set of all components which can interact with K , *i.e.*, those which are directly connected to K . It is often useful to see the environment of a component as a component itself: you can define a composite component E_K that contains all components connected with K . Figure 1.1 shows a composite component K with two sub-components K_1 and K_2 . K_1 's environment E_{K_1} consists of K_2 as well as any components connected through its port q_1 , via port p_1 of its containing component K .

The *observable behaviour* of a component K is the set of all possible *conversations* (sequences of interactions) between a component and its environment. These are sometimes called *traces*.

Contracts and behavioural specifications A *contract* C for some component K , is a pair (A, G) where A is an assumption specification of the behaviour of K 's environment (what K expects its environment to behave) and G is a guarantee specification of K 's behaviour under the assumption A . Note that it is not necessary for assumptions to be inputs and guarantees to be outputs: they could represent any kind of interaction between K and its environment E_K , including both inputs and outputs. In other words, A and G describe sets of possible conversations between K and its environment E_K . For example, a component may assume that if it sends a message to someone in its environment (output) it will eventually get a response (input). Such an assumption includes conversations with information flowing in both directions. Similarly, a component may guarantee that if it receives certain kind of message from someone, it will send out a particular message so someone else. Again, such guarantee includes conversations with information flowing both ways.

Specifications for component behaviours, assumptions and guarantees must

define the set of all possible (observable) behaviours or traces for components, assumptions and guarantees respectively. Such specifications may be described using some suitable formalism. There are many well-known formalisms to describe these, such as:

- regular expressions, and ω -regular expressions
- finite or infinite automata, Büchi automata or other form of labelled transition systems (LTSs),
- temporal logics such as LTL [17], CTL [2, 5], CTL* [6], the μ -calculus [12],
- process algebras or process-calculi such as CCS [14, 13], CSP [10], ACP [4, 3], the π -calculus [15, 16],
- programming or modelling languages based on the above.

The core of AGCL doesn't assume any particular formalism to describe specifications, but the AGCL plug-in for OSATE, described in Chapter 2, uses the Property Specification Language (PSL) [11], an IEEE industry standard. PSL can be seen as a combination of LTL with ω -regular expressions and optionally CTL. We use this language for the examples in Chapter 3.

Compositional analysis AGCL is intended to aid *compositional analysis*, this is, the analysis of (composite) components to determine whether they satisfy their contracts based on their internal structure. Compositional analysis is a *divide and conquer* technique: to determine whether a composite component satisfies its contract, we first analyze (recursively) the subcomponents to determine whether they satisfy their contracts, and then the results of these analyses are combined to give an answer for the composite.

The main advantage of compositional analysis over monolithic techniques is that it enables *incremental analysis*: if only one component changes, or only one contract changes, only the affected part needs to be re-analyzed and the analysis results are guaranteed to be preserved. This aids in tackling the so-called *state-space explosion problem*, whereby adding a new component multiplies the number of states in the system, thus exponentially increasing the total states (and therefore the verification time required).

The theory and algorithms to perform compositional analysis for AADL and RTEdgeTM (a subset of AADL) are described in [18].

Chapter 2

The AGCL plug-in for OSATE

In this section we describe the installation and basic operation of a plug-in for OSATE which supports analysis of AGCL annotations to AADL components¹.

OSATE is an open-source tool platform to support the development and analysis of AADL models. See [21] for information on how to obtain, install and use OSATE.

2.1 Installation

The current version of the AGCL plug-in (version 0.1) can be obtained only from sources.² To install it follow these instructions:

2.1.1 Prerequisites

The prerequisites are:

- On ECLIPSE:
 - ECLIPSE MODELING TOOLS (MDT) (4.3)
 - XTEXT (2.6.0)
 - GOOGLE GUICE
 - Graphical Editing Framework Zest Visualization Toolkit (GEF-ZVT)
 - SLF4J
 - OSATE core (sources) (2.0.3)
- External:
 - NuSMV (2.5.4)

¹The current version of the plug-in supports analysis only for threads, thread groups and their implementations.

²A future version may be obtained as an ECLIPSE feature.

2.1.2 Installing ECLIPSE dependencies and OSATE sources

Install OSATE from sources. You can follow the instructions from the OSATE wiki website: https://wiki.sei.cmu.edu/aadl/index.php/Getting_Osate_2_sources but you might run into problems with XTEXT if you obtain it from the source provided there, hence the following is recommended instead:

1. Install ECLIPSE MODELING TOOLS (MDT) version 4.3 (KEPLER) or later. It can be obtained at <https://www.eclipse.org/downloads/>:
 - (a) Download the appropriate archive (the archive should be named something like `eclipse-modeling-kepler-SR2-platform.zip` or alternatively, `eclipse-modeling-kepler-SR2-platform.tar.gz`) where *platform* is the platform where you are going to install ECLIPSE, for example, for the standard 64 bit Linux distribution *platform* is `linux-gtx-x86_64`, for the standard MacOSX 64 bit distribution, it is `macosx-cocoa-x86_64`, and for the standard Windows distribution it is `win64`, etc.
 - (b) Extract this archive³ in some folder, *e.g.*, `/home/user/osate-dev`. Once extracted it should have created a folder called “`eclipse`” (`/home/user/osate-dev/eclipse`) with an executable called “`eclipse`” inside it.⁴
 - (c) Run the ECLIPSE executable. It will ask you for a workspace folder. For example, `/home/user/osate-dev/workspace`.
2. Install XTEXT:
 - (a) Click on “Help>Eclipse Marketplace...”
 - (b) In the “Find” search box type “Xtext” and hit “Enter” or click on “Go”.
 - (c) The first entry should show the latest version of XTEXT (2.6.0 at the time of this writing). Click on “Install”.
 - (d) When it asks you to confirm the selected features, make sure that “Xtext” and “Xtext SDK” are ticked. Click on “Confirm>”
 - (e) Select “I accept the terms of the licence agreement” and click “Finish”.
 - (f) It will pop up a security warning window, saying that you are installing software that contains unsigned content. Click on “OK”.
 - (g) It will pop up a window asking if you would like to restart now. Click on “Yes”.

3. Install GOOGLE GUICE:

³If you are on MacOSX, avoid using the built-in unarchiver facility because due to a bug, it will corrupt the executable when extracting it. You will have to use a third-party archive extractor instead.

⁴Called “`eclipse.exe`” under Windows.

- (a) Click on “Help>Install New Software...”
- (b) Click the “Add...” button.
- (c) In the Name field enter “Google Guice”.
- (d) In the Location field enter:

`http://guice-plugin.googlecode.com/svn/trunk/
eclipse-update-site/`

- (e) Click “OK”.
- (f) Under the table an entry “Guice” should appear with a checkbox. Tick the checkbox to select it.
- (g) Click “Next” and “Next” again.
- (h) Select “I accept the terms of the licence agreement” and click “Finish”.
- (i) It will pop up a security warning window, saying that you are installing software that contains unsigned content. Click on “OK”.
- (j) It will pop up a window asking if you would like to restart now. Click on “Yes”.

4. Install the Graphical Editing Framework Zest Visualization Toolkit (GEF-ZVT):

- (a) Click on “Help>Install New Software...”
- (b) Click the “Add...” button.
- (c) In the Name field enter “GEF”.
- (d) In the Location field enter

`http://download.eclipse.org/tools/gef/updates/releases/`

- (e) Click “OK”.
- (f) Under the table an entry “GEF (Graphical Editing Framework)” should appear with a checkbox and a right-pointing arrow. Click on the right-pointing arrow to unfold a list of sub-components.
- (g) Select (tick) the box for “Graphical Editing Framework Zest Visualization Toolkit”.
- (h) Click “Next” and “Next” again.
- (i) Select “I accept the terms of the licence agreement” and click “Finish”.
- (j) It will pop up a window asking if you would like to restart now. Click on “Yes”.

5. Install SLF4J:

- (a) Click on “Help>Install New Software...”

- (b) In the “Work with:” field, click the down-pointing arrow to unfold the list of update sites.
 - (c) Choose “Kepler - <http://download.eclipse.org/releases/kepler>”.
 - (d) Look for “slf4j” either by typing it in the text-box field below the “Work with:” field, or by unfolding “General Purpose Tools”. The feature’s full name is “m2e - slf4j over logback logging (Optional)”. Select (tick) it.
 - (e) Click “Next” and “Next” again.
 - (f) Select “I accept the terms of the licence agreement” and click “Finish”.
 - (g) It will pop up a window asking if you would like to restart now. Click on “Yes”.
6. Install the core OSATE plug-in sources:
- (a) Once you restart ECLIPSE, if there is a “Welcome” tab in the ECLIPSE environment, close it.
 - (b) Select “File▷Import...”
 - (c) Unfold the option “Git”
 - (d) Select “Projects from Git” and click “Next”.
 - (e) Select “Clone URI” and click “Next”.
 - (f) Under “Location”, in the “URI” field, enter
“<https://github.com/osate/osate2-core.git>” and click “Next”.
 - (g) Make sure that the branch “master” is selected⁵ and click “Next”.
 - (h) Under “Destination”, in the “Directory:” field enter a directory where you want to download and store your local copy of the OSATE source repositories. For example,

`/home/user/osate-dev/git/osate2-core`
 - (i) Click “Next”.⁶
 - (j) When asked for the wizard for project import, choose “Import existing projects” and click “Next”.
 - (k) In the following dialog box all projects should be selected. Click “Finish”. In the “Package Explorer” view of your ECLIPSE workspace all core OSATE plugins should appear.
 - (l) IMPORTANT: under the plugin `org.osate.aad12` there should be a file called

`“aad12-nouml.genmodel”`

⁵These procedure has been tested only with the master branch. If you want to reproduce this procedure, uncheck the annex and develop branches.

⁶Usually the download gets slow at around 92%, but it will eventually end.

This file should be inside the “model” folder. This file is essential to be able to create XTEXT languages that have access to **AADL** meta-model elements. Unfortunately, in the latest version of OSATE at the time of this writing (June 14, 2014), this file was removed. You will have to ask the OSATE developers to either put the file back, or provide you with an alternative. However, since at the time of this writing there is no alternative, the rest of these notes assume that you have a copy of this file and when you create a plugin called **sampleplugin** you create a “prerequisites” folder “**prereqs**” inside the plugin, and a “models” folder inside **prereqs**, where you should store a copy of **aadl2-nouml.genmodel**, as well as copies of the files “**aadl2.ecore**” and “**aadl2.genmodel**” found under **org.osate.-aadl2/model**. Summarizing, you should have in your plugin, the following files:

- i. **sampleplugin/prereqs/models/aadl2.ecore**
- ii. **sampleplugin/prereqs/models/aadl2.genmodel**
- iii. **sampleplugin/prereqs/models/aadl2-nouml.genmodel**

7. Setup a Run configuration:

- (a) On the ECLIPSE menu select “Run▷Run Configurations...”.
- (b) Select “Eclipse Application”.
- (c) Right-click on it and select “New”.
- (d) In the field “Name:” enter “Osate2”
- (e) In the field “Location:” choose a location for the runtime OSATE workspace. For example **/home/user/osate-dev/runtime-osate-workspace**.
- (f) In the field “Run a product:” click the down-pointing arrow to unfold and select the option “**org.osate.branding.osate2**”.
- (g) Click on the “Arguments” tab.
- (h) Under “VM Arguments” add “**-XX:MaxPermSize=256m -Xmx1200m**”
- (i) Click “Apply”.
- (j) Click “Run”. OSATE starts, and when you close it, the run configuration will appear under the list of Run Configurations as well as the list that appears when you click on the down-pointing arrow next to the “Run” button on the ECLIPSE toolbar.

2.1.3 Installing NuSMV

You can obtain NuSMV from

<http://nusmv.fbk.eu/>

You can install from either sources or binaries. Follow the instructions for your platform. If installing from sources, you’ll have to compile it.

You may be asked to register, but you can click the “Do not register” button near the bottom.

Make sure to take note the full path where the NuSMV executable is installed. For example, on most Unix-like systems (Linux and MacOS X) the default location is `/usr/local/bin/NuSMV`. You will need that to tell the AGCL plug-in where NuSMV is. This is specified in Section 2.2.

2.1.4 Installing the AGCL plug-in

1. Restart ECLIPSE, if there is a “Welcome” tab in the ECLIPSE environment, close it.
2. Select “File>Import...”
3. Unfold the option “Git”
4. Select “Projects from Git” and click “Next”.
5. Select “Clone URI” and click “Next”.
6. Under “Location”, in the “URI” field, enter
“<https://github.com/eposse/osate2-agcl.git>” and click “Next”.
7. Make sure that the branch “master” is selected and click “Next”.
8. Under “Destination”, in the “Directory:” field enter a directory where you want to download and store your local copy of the OSATE source repositories. For example,

`/home/user/osate-dev/git/agclrepo`

9. Click “Next”.
10. When asked for the wizard for project import, choose “Import existing projects” and click “Next”.
11. In the following dialog box all projects should be selected. Click “Finish”. In the “Package Explorer” view of your ECLIPSE workspace the AGCL plug-ins should appear.

2.2 Initial setup

In order to use the plug-in you must tell it where NuSMV is installed. This is done in the preferences dialog:

1. Run OSATE: in the ECLIPSE menu select “Run>Run Configurations...”, select “Ostate2” and click “Run”. If there is a dialog window warning you of errors in the project, just click on “Proceed”.

2. Open the preferences dialog: on Windows or Linux go to “Window▷ Preferences”, on MacOSX go to “OSATE2▷ Preferences...”.
3. Select “AGCL Analysis”.
4. In the field “Model-checker Executable” enter the full path of the NuSMV executable (Subsection 2.1.3). You can click on the “Browse” button to navigate through the file system to the executable.
5. Click on “Apply” and then “OK”.

2.3 The user interface

2.3.1 The full environment

Figure 2.1 shows the OSATE development environment including the AGCL plug-in. We refer the reader to the OSATE documentation for a description of the IDE’s user interface and its use.

The main components are:

- On the top, the main Eclipse menus, including two menus called “OSATE” and “Analyses” which are specific to OSATE.
- Also on the top, a toolbar, showing standard Eclipse icons, and icons for OSATE plug-ins, including the icons for AGCL, described in Subsection 2.3.3
- On the left, the AADL navigator which shows the folders with projects, models and analysis results.
- In the center, the model editors, and other editors.
- On the right, an outline view of the currently selected model.

2.3.2 Setting preferences

Figure 2.2 shows the dialog where you can set the preferences of the analysis plug-in. These are as follows:

- Model-checker Name (currently unused)
- Model-checker Executable: the full path to the binary executable of the model-checker. You must set this according to Section 2.2.
- Model-checker Flags: command-line flags to pass to the model-checker. This line can have the following two meta-variables:
 - `${script}`: the name of the script passed to the model-checker, if any.

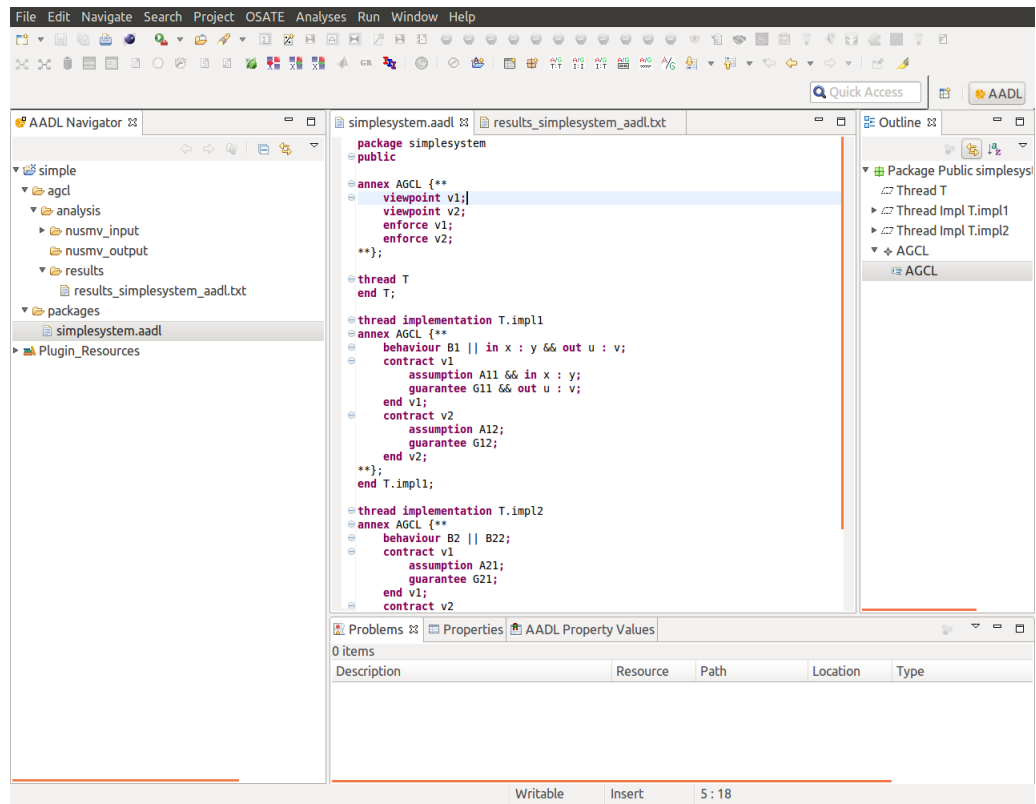


Figure 2.1: The OSATE IDE with the AGCL plug-in.

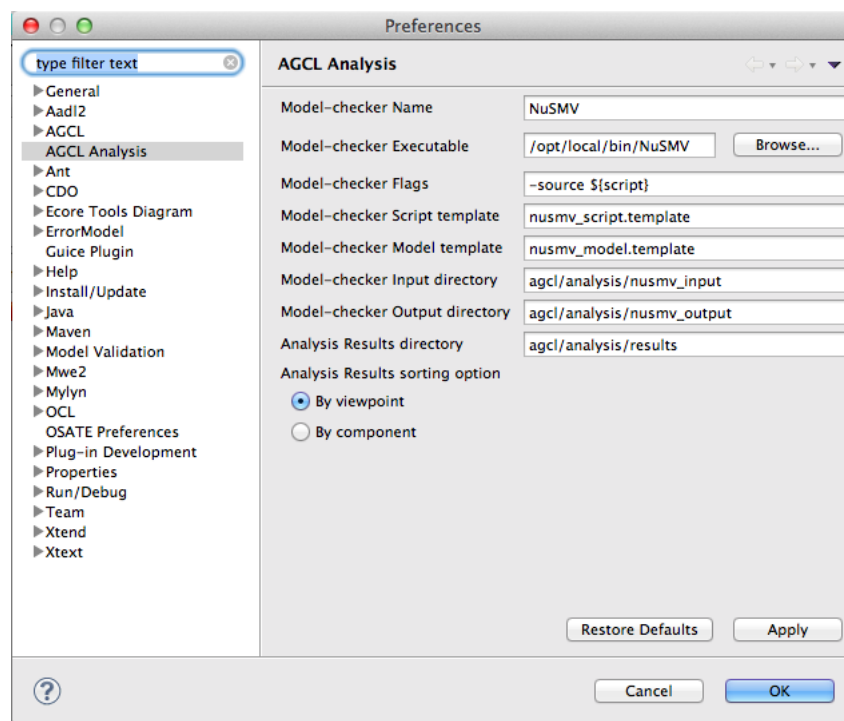


Figure 2.2: AGCL Preference dialog

- `${inputmodel}`: the name of the source file containing the model in the input language of the model-checker.

The values of these variables are computed for each particular invocation of the model-checker based on the name of the AADL model and the kind of analysis. The default “`-source ${script}`” is used for NuSMV without specifying an input model because this is given inside the script passed to NuSMV.

- Model-checker Script Template: the name of the template from which the script passed to the model-checker will be built. This template is located in the “`resources/templates`” folder of the plug-in’s state location. This location is specific to your installation. See the documentation for `org.eclipse.core.runtime.Plugin.getStateLocation()`.

For the NuSMV model-checker, this template accepts the following meta-variables:

- `${inputmodel}`: the name of the source file containing the model in the input language of the model-checker.
- `${counterexamples}`: the name of the file generated by the model-checker with counter-examples, if any.
- `${stdout}`: the name of the file where standard output from the model-checker is to be redirected.
- `${stderr}`: the name of the file where standard errors from the model-checker is to be redirected.

- Model-checker Model Template: the template for source files passed to the model-checker. This template is located in the “`resources/templates`” folder of the plug-in’s state location. This location is specific to your installation. See the documentation for `org.eclipse.core.runtime.Plugin.getStateLocation()`

For the NuSMV model-checker, this template accepts the following meta-variables (See the NuSMV documentation for details):

- `${variables}`: the placeholder for boolean variables representing atomic propositions in a formula.
- `${init}`: the formula defining the set of initial states.
- `${trans}`: the formula(s) defining the set of transitions in the automata.
- `${logic}`: the temporal logic used: LTL, CTL or PSL.
- `${spec}`: the specification to verify.

- Model-checker Input Directory: the directory within the AADL project where files generated by the plug-in to pass to the model-checker is located.

This folder contains the source input models and scripts created from templates. The default is `agcl/analysis/nusmv_input`.

The general form of input model files generated for NUSMV is as follows:

`<aadl_model>_<analysis>_<viewpoint>_<aadl_component>.smv`

The general form of script files generated for NUSMV is as follows:

`<aadl_model>_<analysis>_<viewpoint>_<aadl_component>_session_script.smv`

- **Model-checker Output Directory:** the directory where the model-checker must generate its output. For NUSMV, this includes, standard output, standard error and counter-example files.

The default is `agcl/analysis/nusmv_output`.

The general form of standard output files generated by NUSMV is as follows:

`<aadl_model>_<analysis>_<viewpoint>_<aadl_component>.stdout`

The general form of standard error files generated by NUSMV is as follows:

`<aadl_model>_<analysis>_<viewpoint>_<aadl_component>.stderr`

The general form of counter-example files generated by NUSMV is as follows:

`<aadl_model>_<analysis>_<viewpoint>_<aadl_component>_counterexample.xml`

- **Analysis Results directory:** the directory where the result tables will be generated. The default is `agcl/analysis/results`. Each analysis produces two files named

- `result_<aadl_model>.txt`: plain text table with the results
- `result_<aadl_model>.xml`: XML representation (an ECORE resource).

- **Analysis Results sorting option:** the criterion used to display the results table. There are two options






- By viewpoint
- By component

2.3.3 Performing analysis

To perform analysis, the user must do the following:

1. select a particular AADL model, by clicking on the “.aadl” file in the AADL Navigator view, and then
2. either select an option from the menu **Analyses**▷**Contract-based Analyses**, or click on the appropriate analysis button in the OSATE toolbar.

The AGCL buttons in the toolbar are the following:

Icon	Analysis
	Perform type to type analysis
	Perform implementation to implementation analysis
	Perform implementation to type analysis
	Perform atomic analysis
	Perform compositional analysis

See Section 3.2 for a detailed description of these kinds of analyses.

When executing any of these analyses, the results are stored in a folder within root directory of the project to which the AADL model belongs. The name of the folder for results is defined in the preferences (see Subsection 2.3.2).

IMPORTANT: Due to a bug in the current version of OSATE (2.0.3), if you select a model, perform some analysis, edit the source AADL model after performing one of the analyses, and then try to perform the analysis again on the same model, OSATE will not update correctly the resources and pass the old version of the model to the plug-in, so the results will be the same as in the first analysis. Currently there are only two known workarounds: either restart OSATE, or create a new model, copy the contents of the old one into the new one, edit it and perform the analysis in the new copy.

Chapter 3

The AGCL language

3.1 Contracts in AGCL

The following is a simple specification for a component annotated with a behaviour specification B_1 and a contract $C_1 = (A_1, G_1)$ with assumption A_1 and guarantee G_1 :

```
1 behaviour B1;  
2 contract C1  
3   assumption A1;  
4   guarantee G1;  
5 end C1;
```

Such specification is intended to be associated with a component in a model. For example, suppose we have a component in AADL which is a thread implementation K_1 of some thread type T_1 as follows:

```
1 thread implementation T1.K1  
2 -- implementation details  
3 -- ...  
4 end T1.K1;
```

Then, this component can be annotated with an AGCL *annex* as follows:

```
1 thread implementation T1.K1  
2 -- implementation details  
3 -- ...  
4 annex AGCL {**  
  
5   behaviour B1;  
6   contract C1  
7     assumption A1;  
8     guarantee G1;  
9   end C1;
```

```

10 **}
11 end T1.K1;

```

The meaning of this annotation is that component $T_1.K_1$ behaves as specified by B_1 and should satisfy the contract C_1 , this is, the behaviour B_1 and the assumptions defined by A_1 must imply the guarantees given by G_1 . Whether the component does satisfy the implementation is determined by one of the analyses described below.

3.2 Kinds of analysis

There are several types of analysis that can be done with contracts. The main types of analysis in which we are interested are the following:

- Atomic analysis: verify that the behaviour of thread implementations satisfy their contract.
- Compositional analysis: verify that the composition of sub-component contracts satisfy the corresponding contract of a thread group (composite component).
- Conformance analysis:
 - Type to type analysis: verify that the contract of a (thread or component) type satisfies the corresponding contract of its parent type.
 - Implementation to implementation analysis: verify that the contract of a (thread/component) implementation satisfies the corresponding contract of its parent implementation.
 - Implementation to type analysis: verify that the contract of a component (thread) implementation satisfies the corresponding contract of its type.

We describe these analyses through an example. We assume the reader is familiar with PSL or with LTL. If the reader is unfamiliar with either of these, we suggest starting by reading about LTL, which is a proper subset of PSL. There are many sources for LTL such as textbooks on model-checking, e.g. [1].

3.2.1 An example: client-mediator-server

Consider the model shown in Figure 3.1. This model depicts a system consisting of a client and a server which itself consists of a front-end or mediator, and a back-end. In this common pattern, the client may issue requests via a channel `req` and expects an answer on channel `ans`. The front-end of the server receives these requests, may perform some preprocessing, and delegates the requests to the back-end server via the `internal_req` channel. When the back-end

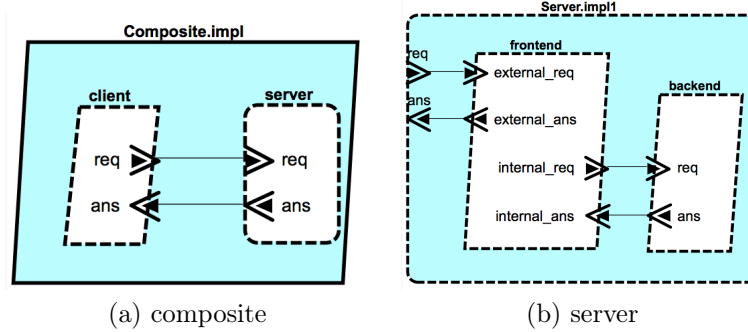


Figure 3.1: A simple client-server architecture with a mediator process.

server responds on the `internal_ans` channel, the front-end may do some post-processing, and deliver the final answer to the client.

We will now look at the textual representation of this model, annotated with AGCL annex subclauses containing the component's contracts. We assume that the model is saved in an AADL file called `client_server_mediator.aadl`.

To annotate components with contracts we need to declare viewpoints, which is done at the package-level annex library as shown in the excerpt below (using the `viewpoint` keyword). The `enforce` keyword is used to inform the tool which viewpoints should be analyzed. A *viewpoint* is a family of contracts representing a particular requirement. The name of a contract determines the viewpoint to which it belongs. Thus, all contracts with the same name across components are meant to represent a particular requirement for the whole system, with the contract for each component representing the contribution of that component to the whole. In the example below we show two viewpoints, but in the remainder of the example we consider only the first one. The developer is free to introduce new viewpoints and the corresponding contracts at any point during development.

```

1 package client_server_mediator
2 public
3 annex AGCL {**
4     viewpoint normal_operation;
5     viewpoint alternative_operation;
6     enforce normal_operation;
7 **};
8 -- etc.
9 end client_server_mediator;
```

The composite process is shown in Figure 3.2¹ where the client thread type

¹To keep the presentation of our example simple we show only the ports, connections and annex for each classifier.

```

1 process Composite
2 end Composite;

3 process implementation Composite.impl1
4 subcomponents
5   client : thread Client.impl1;
6   server : thread group Server.impl1;
7 connections
8   req : port client.req -> server.req;
9   ans : port server.ans -> client.ans;
10 end Composite.impl1;

```

Figure 3.2: The composite process containing the client and server.

```

1 thread Client
2 features
3   req : out event data port;
4   ans : in event data port;
5 end Client;

```

Figure 3.3: Client thread type.

is shown in Figure 3.3. The server is described in Subsection 3.2.3 below.

3.2.2 Contracts for atomic components (threads)

Figure 3.4 shows the backend server. Its annex has a **behaviour** clause describing the behaviour of the actual implementation, and a **contract** clause defining a contract for this component within the **normal_operation** viewpoint as indicated by the contract's name. At any point you can add more viewpoints and more contracts on any component².

The behaviour, and the assumption and guarantee of contracts are written as PSL expressions.³ Atomic propositions on these formulas must have one of the following forms:

- **in** *port:signal*
- **out** *port:signal*
- *connection*

The meaning of these forms is slightly different on a **behaviour** clause, on an **assumption** and on a **guarantee**. This is explained below.

Note: you cannot mix the first two styles with the third. Either all atomic propositions are marked **in** or **out**, or they are connections, but not both. The

²The current version of the plugin supports contracts only on **thread** and **thread implementation** classifiers.

³The current version of the plug-in supports only the LTL subset of PSL.

```

1 thread BackendServer
2 features
3   req : in event data port;
4   ans : out event data port;
5 end BackendServer;

6 thread implementation BackendServer.impl1
7 annex AGCL {**
8   behaviour always (in req:s1 -> next out ans:s2);
9   contract normal_operation
10    assumption TRUE;
11    guarantee always (in req:s1 -> eventually out ans:s2);
12  end normal_operation;
13 **};
14 end BackendServer.impl1;

```

Figure 3.4: Backend server.

same is true for assumption and guarantee clauses. In fact, the same style must be used in the three kinds of clauses: if you use atomic propositions marked **in** or **out** on a behaviour clause, you must also use these in the contracts for the component, and if you use the connections, the contract clauses must use connection names as well.

The behaviour clause of the backend server in Figure 3.4 states that whenever the server receives a request (an **in** event on the **req** port with some signal **s1**), then it will produce an output on the **ans** port in the next state or cycle. The contract in this case has no assumptions and therefore it is simply **true**. The guarantee is that whenever the backend receives a request, it will eventually produce an answer. In this case, it should be fairly trivial that the behaviour satisfies the contract.

The idea behind having both a behaviour clause and a contract is that the behaviour clause describes the behaviour as it is (or should be) implemented by the component, whereas the contract is an abstraction that describes the (expected) obligations on the environment and the component itself. Hence, the behaviour will be in general more detailed than the contract. In fact, the behaviour specification may be inferred from the actual implementation⁴. For example, if the component is implemented as a state machine, it may be possible to infer the PSL behaviour specification from the state machine. The contract, on the other hand, is intended to be a *declarative* specification.

Note that the guarantee can talk about both inputs and outputs. The same is true for assumptions. This is an important distinction with several existing approaches to assume/guarantee reasoning where typically assumptions are only on inputs and guarantees are only on outputs. In our framework, assumptions and guarantees are specifications of sets of possible *observable behaviours*, where an observable behaviour is a “conversation” or sequence of interactions between a component and its *environment* (the set of components to which it is con-

⁴The current version of the plug-in doesn’t do this sort of inference.

nected), and in these conversations, information can flow in both directions. A guarantee represents an obligation on the component, whereas an assumption represents an obligation on its environment. Hence, when a guarantee states an atomic proposition labeled **in**, it is stating the component's obligation to accept or receive an input. When a **in** atomic proposition appears in an assumption, the input direction is stated from the point of view of the component but it actually represents an output obligation from the component's environment to the component. Similarly, an **out** in a guarantee is an obligation for the component to produce output, whereas an **out** in an assumption, while stated from the point of view of the component, actually represents an obligation on the environment to accept or receive input coming from the component.

So, as we have seen, the meaning of the forms of atomic propositions is slightly different for **behaviour** clauses, **assumption** clauses and **guarantee** clauses. Summarizing, on a **behaviour** clause the meanings are:

- **in port: signal**: the component's behaviour at that point in the execution sequence is the reception of the given signal on the given input port,
- **out port: signal**: the component's behaviour at that point in the execution sequence is the sending of the given signal on the given output port,
- **connection**: the component's behaviour at that point in the execution sequence is an interaction (input or output) over the given connection.

On an **assumption** clause the meanings are:

- **in port: signal**: the component, at that point in the execution sequence, expects the reception of the given signal on the given input port, from its environment,
- **out port: signal**: the component, at that point in the execution sequence, expects its environment to receive a signal on the given output port, from the component,
- **connection**: the component, at that point in the execution sequence, expects an interaction (input or output) over the given connection.

On a **guarantee** clause the meanings are:

- **in port: signal**: the component, at that point in the execution sequence, will accept the reception of the given signal on the given input port, from its environment,
- **out port: signal**: the component, at that point in the execution sequence, will send the signal on the given output port,
- **connection**: the component, at that point in the execution sequence, expects an interaction (input or output) over the given connection.

```

1 thread Frontend
2 features
3   external_req : in event data port;
4   external_ans : out event data port;
5   internal_req : out event data port;
6   internal_ans : in event data port;
7 end Frontend;

8 thread implementation Frontend.impl1
9 annex AGCL {**
10   behaviour always (in external_req:s1
11     -> eventually (out internal_req:s1
12       & always (in internal_ans:s2
13         -> eventually out external_ans:s2)));
14   contract normal_operation
15     assumption always (out internal_req:s1
16       -> eventually in internal_ans:s2);
17     guarantee always (in external_req:s1
18       -> eventually out external_ans:s2);
19   end normal_operation;
20 **};
21 end Frontend.impl1;


```

Figure 3.5: Server frontend (mediator).

These distinctions might be subtle, but are significant: the behaviour clause describes what the component does. The assumption clause expresses an obligation on the component’s environment. The guarantee clause expresses an obligation on the component, if the assumption holds.

Figure 3.5 shows the frontend. Its behaviour clause specifies that whenever an external request arrives (from the client), eventually it will reach a state where it will send a request to the backend (through the `internal_req` port) and from that point onwards, whenever it receives an answer from the backend, it will eventually forward the answer to the client on the `external_ans` port. The contract clause specifies as assumption that whenever it sends a request to the backend server, it will get an answer from it eventually. The guarantee states that whenever it receives an external request from the client, it will eventually send an answer back to the client. In this case it is less trivial that the behaviour satisfies the contract, but this follows from the formal semantics of PSL.

In OSATE we can perform this analysis by first selecting the AADL model in the AADL Navigator view (by default on the left side of the main window), and then choosing the menu option “Analyses▷Contract-based Analyses▷

A/G atomic analysis” or by clicking on the toolbar button . This will cause the plug-in to go through all atomic components (thread implementations in the current version) and try to establish if their behaviour, as specified by the `behaviour` clause in the annex, satisfies each of the contracts for that component that belong to an enforced viewpoint (a viewpoint which has been declared and for which there is an `enforce` clause in the annex library at

Viewpoint	Component	Result
normal_operation	BackendServer.impl1	satisfied
normal_operation	Frontend.impl1	satisfied

Table 3.1: Results for the atomic analysis of the BackendServer and Frontend threads.

the package level, as shown in Subsection 3.2.1). Then the results will appear in the project's results folder specified in the preferences (see Subsection 2.3.2), which by default is `agcl/analysis/results`, and the file name will be `results_client_server_mediator.txt`. Table 3.1 shows the contents of the plain-text results file. The file name includes the model name and the table lists each of the components analyzed and each of the viewpoints (see Subsection 2.3.2 to select the sorting options). Each row in the table corresponds to the analysis results for a particular component and a particular contract viewpoint.⁵ The last column states whether the contract was satisfied or not.

If a contract is not satisfied, a NUSMV counterexample file will be generated under the model-checker's output folder located in the project's folder (by default `agcl/analysis/nusmv_output`). This counter-example consists of a trace of states, where each state is a particular truth-value assignment to each atomic proposition. For example, suppose that we add a new implementation of the back-end server called `BackendServer.impl2` as shown in Figure 3.6. In this case, the backend server's behaviour is that whenever it receives both signals `s1` and `s2` on port `req`, then it will respond on port `ans`, two cycles later. The guarantee is the same as before. In this case, the results are shown in Table 3.2 and the counter-example file, named `client_server_mediator_aadl_atomic_normal_operation_BackendServer_impl2_counterexample.xml` is shown in Figure 3.7. In the trace we can see the problem. In the first state, both `in req:s1` and `in req:s2` hold, and two states later, `out ans:s2` holds, as prescribed by the `behaviour` clause. However, in the same behaviour, we end in a state where `in req:s1` holds alone (state 4), but `out ans:s2` never holds from then onwards, because the trace ends in a state where `out ans:s2` is false and there is a loop to itself (state 5). Hence there is no guarantee that whenever `in req:s1` holds, eventually `out ans:s2` will hold. This was to be expected, as the behaviour says that it will produce an answer but only when both signals arrive on the port, so the component cannot guarantee that it will give an answer if only signal `s1` arrives.

In general, for threads, a behaviour B satisfies a contract $C = (A, G)$ with assumption A and guarantee G , if the formula $B \wedge A \Rightarrow G$ is valid. Intuitively,

⁵In this example we have only one enforced viewpoint, but if there is more than one, all of them will be analyzed and shown. The table is sorted first by viewpoint and then by component, but this can be changed in the preferences. See Subsection 2.3.2.

```

1 thread implementation BackendServer.impl2
2 annex AGCL {**
3   behaviour always (in req:s1 & in req:s2
4                     -> next next out ans:s2);
5   contract normal_operation
6     assumption TRUE;
7     guarantee always (in req:s1 -> eventually out ans:s2);
8   end normal_operation;
9 **};
10 end BackendServer.impl2;

```

Figure 3.6: New implementation of the back-end server.

Viewpoint	Component	Result
normal_operation	BackendServer.impl1	satisfied
normal_operation	BackendServer.impl2	not satisfied
normal_operation	Frontend.impl1	satisfied

Table 3.2: Results for the atomic analysis of the two implementations of the BackendServer and the Frontend threads.

the behaviour and the assumptions must be enough to imply the guarantee. A (linear) temporal logic formula (including PSL) is *valid* if it holds in all possible paths for every possible model. In our case, the premise of this implication captures the model: the guarantee will be required to be true only on those models with behaviour B , if the assumption A is true as well. The validity of PSL formulas can be established with a model-checker.

As we mentioned before, an AGCL annex can contain multiple contracts, which can be verified independently. This allows the developer to add contracts as the design progresses, and define contracts which focus only on particular aspects of interest.

3.2.3 Contracts for composite components (thread groups)

Figure 3.8 shows the server combining frontend and backend. In this case, the thread group does not have a behaviour specification, but only a contract. It's contract doesn't make any assumptions, but it states the guarantee that whenever an external request comes from the client, eventually it will answer it.

The problem in this case is the following: if we already know that the sub-components (`BackendServer.impl1` and `Frontend.impl1`) satisfy their respective contracts, how do we establish if the composition (the `Server.impl1`) satisfies its contract?

In OSATE we can perform this analysis by first selecting the AADL model in the AADL Navigator view (by default on the left side of the main window), and then choosing the menu option “Analyses▷Contract-based Analyses▷A/G

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <counter-example type="0" desc="PSL Counterexample" >
3   <node>
4     <state id="1">
5       <value variable="in_req_s2">TRUE</value>
6       <value variable="in_req_s1">TRUE</value>
7       <value variable="out_ans_s2">FALSE</value>
8     </state>
9   </node>
10  <node>
11    <state id="2">
12      <value variable="in_req_s2">TRUE</value>
13      <value variable="in_req_s1">FALSE</value>
14      <value variable="out_ans_s2">FALSE</value>
15    </state>
16  </node>
17  <node>
18    <state id="3">
19      <value variable="in_req_s2">TRUE</value>
20      <value variable="in_req_s1">FALSE</value>
21      <value variable="out_ans_s2">TRUE</value>
22    </state>
23  </node>
24  <node>
25    <state id="4">
26      <value variable="in_req_s2">FALSE</value>
27      <value variable="in_req_s1">TRUE</value>
28      <value variable="out_ans_s2">FALSE</value>
29    </state>
30  </node>
31  <node>
32    <state id="5">
33      <value variable="in_req_s2">FALSE</value>
34      <value variable="in_req_s1">FALSE</value>
35      <value variable="out_ans_s2">FALSE</value>
36    </state>
37  </node>
38  <node>
39    <state id="6">
40      <value variable="in_req_s2">FALSE</value>
41      <value variable="in_req_s1">FALSE</value>
42      <value variable="out_ans_s2">FALSE</value>
43    </state>
44  </node>
45  <loops> 5 </loops>
46 </counter-example>

```

Figure 3.7: Counter-example for the normal-operation contract of BackendServer.impl2.


```

1 thread group Server
2 features
3   req : in event data port;
4   ans : out event data port;
5 end Server;

6 thread group implementation Server.impl1
7 subcomponents
8   backend : thread BackendServer.impl1;
9   frontend : thread Frontend.impl1;
10 connections
11   client_req : port req -> frontend.external_req;
12   client_ans : port frontend.external_ans -> ans;
13   server_req : port frontend.internal_req -> backend.req;
14   server_ans : port backend.ans -> frontend.internal_ans;
15 annex AGCL {**
16   contract normal_operation
17     assumption TRUE;
18     guarantee always (in req:s1
19       -> eventually out ans:s2);
20   end normal_operation;
21 **};
22 end Server.impl1;

```

Figure 3.8: The server.

composite analysis” or by clicking on the toolbar button . As with atomic analysis, the results will appear in the project’s results folder specified in the preferences (see Subsection 2.3.2), which by default is `agcl/analysis/results`. In this example, the contract is indeed satisfied by the composition of the subcomponents.

How is the satisfaction of the contract for a composite component determined based on the contracts of its subcomponents? This can be established as follows: let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be contracts for the two subcomponents K_1 and K_2 of a composite component K with contract $C = (A, G)$. Assuming that K_1 satisfies C_1 , and K_2 satisfies C_2 , then K satisfies C if the following two PSL formulas are valid:

1. $G' \Rightarrow G$ where $G' \stackrel{\text{def}}{=} G_1 \wedge G_2$, and
2. $A \Rightarrow A'$ where $A' \stackrel{\text{def}}{=} (G_2 \Rightarrow A_1) \wedge (G_1 \Rightarrow A_2)$

Intuitively the first one states that the guarantees of the subcomponents together must imply the guarantee of the composite. The second one states that the assumption of the composite must be enough to ensure that 1) the guarantee of the second must imply the assumption of the first, and 2) the guarantee of the first component implies the assumption of the second. This is because the subcomponents may be connected and information may flow both ways between them, and they are part of each other’s environments: the behaviour of K_1 ’s

environment is given by K_2 's guarantees G_2 together with K 's environment given by A . Hence, A and G_2 must imply A_1 . Similarly for K_2 .

To be precise, there is a little processing that needs to be done on the formulas G_i and A_i , namely we need to replace port references occurring in atomic propositions by connector references so that they refer to the same entity, and we need to flip the direction (in/out) of those atomic propositions in assumptions for the same reason.

For composite components with n subcomponents, the formulas are generalized to $G' \stackrel{\text{def}}{=} G_1 \wedge G_2 \wedge \dots \wedge G_n$ and $A' \stackrel{\text{def}}{=} \bigwedge_{i=1}^n ((\bigwedge_{j \neq i} G_j) \Rightarrow A_i)$ respectively. In other words, the guarantees of all subcomponents must imply the guarantee of the composition, and the assumption of each subcomponent must be implied by the guarantees of all other subcomponents. This later requirement can be relaxed in that it is only needed that the assumption of each subcomponent must be implied by the guarantees of only those subcomponents connected to it.

In our example, K_1 and K_2 are `Backend.impl1` and `Frontend.impl1`, and K is `Server.impl1`. As before, we establish the validity of the formulas above with a model-checker and in this case they happen to be true.

Incremental analysis is supported in the following way: if one component changes its behaviour, for example the frontend, we only need to check whether this behaviour satisfies its contract(s). If the result of this analysis is positive, then there is no need to check other components, or the validity of the composite formulas, as the contract has not changed and therefore the validity of formulas 1 and 2 is preserved. If the result of this analysis fails, then the developer needs to either modify the behaviour or the contract for the component in question. If the contract for a component changes then one must re-analyze that component (recursively if it is a composite component) and then re-evaluate the implications $G' \Rightarrow G$ and $A \Rightarrow A'$ as above, but there is no need to re-analyze components which have not changed or whose contract has not changed, as they would not change the validity of these formulas.

3.2.4 Conformance

Contracts can annotate not only implementations but also types. This opens a set of closely related problems that need be addressed. The first one is this: if we have a component implementation K of type T and K has a contract $C_K = (A_K, G_K)$ and T is annotated with contract $C_T = (A_T, G_T)$, how do we know that C_K conforms to C_T ?

Suppose that we modified the server type to include an additional port on as shown in Figure 3.9. The `Server` thread type is annotated with a contract whose assumption states that in every state, if there is a request, the on signal is true in the same state, and if the server provides an answer in that state, the on signal is also true. The guarantee is that every client request will eventually be answered. The implementation `Server.impl2` is annotated with a contract whose assumption states that if there is a request, the on signal is true in the

```

1 thread group Server
2 features
3   req : in event data port;
4   ans : out event data port;
5   on : in event port;
6 annex AGCL {**
7   contract normal_operation
8     assumption always (in req:s1 -> in on:s1)
9       & always (out ans:s2 -> in on:s1);
10    guarantee always (in req:s1
11      -> eventually out ans:s2);
12  end normal_operation;
13 **};
14 end Server;


15 thread group implementation Server.impl2
16 subcomponents
17   backend : thread BackendServer.impl1;
18   frontend : thread Frontend.impl1;
19 connections
20   client_req : port req -> frontend.external_req;
21   client_ans : port frontend.external_ans -> ans;
22   server_req : port frontend.internal_req -> backend.req;
23   server_ans : port backend.ans -> frontend.internal_ans;
24 annex AGCL {**
25   contract normal_operation
26     assumption always (in req:s1 -> in on:s1);
27     guarantee always (in req:s1
28       -> eventually (out ans:s2
29         & in on:s1));
30   end normal_operation;
31 **};
32 end Server.impl2;

```

Figure 3.9: The server.

same state, and its guarantee is that every client request is answered and when the answer is provided, the on signal is true.

As before, in OSATE we can perform this analysis by first selecting the AADL model in the AADL Navigator view (by default on the left side of the main window), and then choosing the menu option “Analyses▷Contract-based Analyses▷A/G implementation/type conformance analysis” or by clicking

on the toolbar button . This will cause the plug-in to go through all component implementations (thread implementations and thread group implementations in the current version) and try to establish if, for each enforced viewpoint their contracts of that viewpoint conform to their type’s contract of the same viewpoint. And as before, the results will appear in the project’s results folder specified in the preferences (see Subsection 2.3.2), which by default is `agcl/analysis/results`, and the file name will be `results_client_server_mediator.txt`.

We can establish that C_K conforms to C_T by checking two implications: $G_K \Rightarrow G_T$ and $A_T \Rightarrow A_K$. Note that the implication is covariant on guarantees and contra-variant on assumptions. For guarantees, this is because the guarantee of the type must be a guarantee of any of its implementations: the set of possible observable behaviours described in G_K must be a subset if the set of behaviours defined by G_T , otherwise there would be at least one behaviour guaranteed by the implementation which does not conform to what the type prescribes. For assumptions the direction is contra-variant because the set of behaviours specified by A_T must be a subset of the set of behaviours specified by A_K . If this wasn't required, there would be at least one environment behaviour acceptable by A_T but not by A_K which would entail that component K would not be able to be placed in some composite components expecting type T .

The result for our example is that indeed the implementation satisfies the contract of its type, as the guarantee of the implementation implies that of its type, and the type's assumption implies the implementation's assumption.

The other related problems occur when an implementation extends another implementation or a type extends a type and both have contracts in the same viewpoint. These cases can be handled as the above: if K' (or T') has contract $C' = (A', G')$ and it extends K (resp. T) with contract $C = (A, G)$, then conformance can be established by checking the validity of $G' \Rightarrow G$ and $A \Rightarrow A'$.

3.3 Summary

- AADL components can be annotated with multiple contracts with an assumption and a guarantee specification (in PSL).
- The contract's name is the *viewpoint*.
- A viewpoint is a collection of contracts across multiple components specifying some requirement for the whole system.
- Assumptions and guarantees can talk about inputs and outputs on the component's ports.
- Assumptions represent obligations on the component's environment.
- Guarantees represent obligations on the component when the assumptions hold.
- Atomic components (*i.e.* components without sub-components) are also annotated with a *behaviour* clause, specifying the implementation.⁶
- There are three kinds of analysis:
 - Atomic analysis: whether the behaviour of an atomic component satisfies its contracts.

⁶A future version may infer this from the actual implementation.

- Composite analysis: whether the composition of the contracts of the subcomponents of a composite component satisfy the contract of the composite.
- Conformance analysis:
 - * whether the contracts of an implementation component satisfy the corresponding contract(s) of its type, or
 - * whether the contract of an implementation component satisfy the corresponding contracts(s) of its parent implementation, or
 - * whether the contract of a type component satisfy the corresponding contract(s) of its parent type.
- These analysis are performed automatically by the AGCL plugin by choosing the corresponding action in the Analyses menu or the toolbar.
- The results of analysis are stored in a folder within the project containing the model being analysed.

Bibliography

- [1] C. Baier and J-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
- [2] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
- [3] J. A. Bergstra and J. W. Klop. Fixed point semantics in process algebra. Tech. Rep. IW 208, Mathematical Centre, Amsterdam, 1982.
- [4] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1–3):109–137, 1984.
- [5] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Proc. of the Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.
- [6] E. A. Emerson and J. Y. Halpern. “Sometimes” and “not never” revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.
- [7] P. H. Feiler. The SAE Architecture Analysis & Design Language (AADL). SAE International Document AS-5506B., September 2012.
- [8] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language: An Introduction. Tech. Rep. CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie-Mellon University, 2006.
- [9] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language*. SEI series in software engineering. Addison-Wesley, 2012.
- [10] C. A. R. Hoare. Communicating Sequential Processes. *Comm. of the ACM*, 21(8):666–677, August 1978.
- [11] IEEE Computer Society. IEEE Standard for Property Specification Language (PSL). IEEE Standard 1850TM-2010, June 2012.
- [12] D. Kozen. Results on the propositional μ -calculus. In *Proc. of ICALP’82*, 1982.

- [13] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.
- [14] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [15] R. Milner. *Communicating and mobile systems: the π -calculus*. Cambridge University Press, 1999.
- [16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Reports ECS-LFCS-89-85 and ECS-LFCS-89-86 86, Computer Science Dept., University of Edinburgh, March 1989.
- [17] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE Symp. on the Foundations of Comp. Sci. (FOCS '77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE, IEEE Computer Society Press.
- [18] E. Posse. Contract-based compositional analysis for reactive systems in RTEdgeTM, an AADL-based language. Tech. Rep. 2013-607, School of Computing – Queen’s University, August 2013. <http://research.cs.queensu.ca/TechReports/Reports/2013-607.pdf>.
- [19] SAE International. Architecture Analysis & Design Language (AADL). SAE Standard AS5506b, 10 September 2012.
- [20] SAE International. AADL website. <http://www.aadl.info/>, 6 June 2014.
- [21] SEI CMU. Osate 2 wiki. https://wiki.sei.cmu.edu/aadl/index.php/Osate_2, 6 June 2014.