ICT FET IP Project

Deliverable: D2.2 prototypes (part of)

IAAnalyzer: User and Developer guide

http://www.connect-forever.eu
Abstract

This document contains the user and developer guide for the IAAnalyzer tool. IAAnalyzer stands for Interface Automata Analyzer. It is a software tool supporting the modelling and behavioural analysis of mediators that are designed as compound terms of the connector algebra formalized within the work of the work package WP2 of the CONNECT project. The formalized algebra is described in [2].

Keyword List

Interface Automata, Functional Analysis, Software Connectors, Software Mediators, Software Components.
### Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Type of Change</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Document Review

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Reviewer</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table of Contents

LIST OF FIGURES ......................................................................................... 7

LIST OF TABLES.......................................................................................... 9

1 THE IAAnalyzer TOOL: USER AND DEVELOPER GUIDE ................. 11
  1.1 Overview of the tool ............................................................................. 11
  1.2 User guide .......................................................................................... 13
  1.3 Developer guide .................................................................................. 24
  1.4 Future work ......................................................................................... 33

BIBLIOGRAPHY............................................................................................ 35
List of Figures

Figure 1.1: The IAAnalyzer modelling and analysis process ................................................. 12
Figure 1.2: Creation of an empty IAAnalyzer model .......................................................... 14
Figure 1.3: Definition of the alphabet of actions ................................................................. 16
Figure 1.4: Labelling a transition ......................................................................................... 17
Figure 1.5: The CFring/XMPP example ............................................................................. 18
Figure 1.6: Definition of a composite binary connector ....................................................... 19
Figure 1.7: Definition of a plugging operator ...................................................................... 20
Figure 1.8: Algebraic specification of the connector for our CFring/XMPP example .......... 21
Figure 1.9: IA model of the connector for our CFring/XMPP example ............................... 21
Figure 1.10: Generation of the Ticc specification ............................................................... 22
Figure 1.11: Packaging of the generated Ticc specification .................................................. 23
Figure 1.12: Uncorrect connector (on the top), correct connector (on the bottom) .......... 23
Figure 1.13: The IAAnalyzer meta-model ......................................................................... 25
Figure 1.14: Ecore annotations required for the generation of the custom validation skeleton code ... 26
List of Tables
1 The IAAnalyzer tool: user and developer guide

1.1 Overview of the tool

This document contains the user and developer guide for the IAAnalyzer tool. IAAnalyzer stands for *Interface Automata Analyzer*. It is a software tool supporting the modelling and behavioural analysis of mediators that are designed as compound terms of the connector algebra formalized within the work of the work package WP2 of the CONNECT project. The formalized algebra is described in [2]. We strongly suggest to look at the description of the algebra formalization before reading this document. That is, this document requires as a prerequisite the full knowledge of the formalized algebra.

The IAAnalyzer source code and binaries can be freely downloaded from the training page at the CONNECT web-site:


IAAnalyzer is implemented as a stand-alone Eclipse product so as to prevent users to deal with possible tedious issues concerning missing dependencies. It has been developed as a set of Eclipse plug-ins exploiting both the Eclipse Modeling Framework (EMF) project [7, 14] and the Acceleo 3.0 Model-to-Text (M2T) transformation project [6].

From a model specification described in XMI (an ecore meta-model), EMF provides tools and runtime support to produce a set of Java classes for the model, (along with a set of adapter classes that enable viewing and command-based editing of the model) and a basic editor (tree-view based). The Acceleo Project is part of the Eclipse Modelling/M2T project. Its primary goal is to provide an implementation of the MOF M2T OMG standard [13]. From the specified model and M2T transformations, the Acceleo 3.0 run-time support first generates a M2T Eclipse plug-in. This plug-in represents the Java implementation of the transformation classes. In a second stage, an Acceleo UI Launcher Project is also generated by the run-time support. It is a popup-menu-extension of the former plug-in which implements a GUI to load and browse a model specification and perform the developed transformations.

The main reasons for which we decided to embark on this implementation of the algebra are: (i) to validate the formalized connector algebra; (ii) to have a modelling tool in order to design connectors as (compound) terms of our algebra, while having automated support for the generation of the connector IA-based semantics; and (iii) to support the automated analysis (e.g., simulation, refinement, model-checking, etc.) of the modelled connectors in terms of “correctness” checks with respect to the protocol of the Networked Systems (NSs) and the requirements specified for the whole connected system.

Concerning this last purpose, by exploiting Acceleo M2T transformation, IAAnalyzer automatically generates the Ticc [1, 3] specification of the modelled NSs and connector. Ticc (Tool for Interface Compatibility and Composition) is a tool for the prototyping and verification of distributed designs. Ticc provides the following features: modelling of components and their interaction in terms of Sociable Interfaces [11] (IA represent a particular case of them); composition and compatibility checking; simulation and verification of CTL properties [12]. Ticc is implemented as a set of functions that extend the capabilities of the OCaml command-line. The tool is released under the GPL and its code is freely available.

The current version of IAAnalyzer adheres to the current formalization of the algebra and, hence, it should be considered as a base implementation for the possible future extensions of the algebra. That is the more the algebra formalization will be extended, the more new features and functionalities, for IAAnalyzer, will be accordingly implemented (or integrated with), e.g., combination of quantitative and interface models, connector (code) synthesis, integration with Prism, etc.

Figure 1.1 represents the modelling and analysis process that the users follow while interacting with IAAnalyzer (and its integrated tools). Process activities are represented as ellipses, whereas process artefacts as boxes. Some activities are specifically supported by IAAnalyzer, others such as analysis can be carried on by externals tools, such as Ticc, which are integrated through M2T transformation. Dotted-lined boxes with numbers denote the order in which the various activities are carried on. In the following, we briefly describe the process activities shown in Figure 1.1.

---

1http://connect-forever.eu/
1. **Networked Systems (NSs) & Connector Algebraic Specification.** By using the IAAnalyzer editor, the user can model the interaction behaviour of the NSs under study and of the connector that make them able to interoperate. The former is expressed by using the IA syntax, the latter by combining terms of the underlying algebra (e.g., Merge, Order, Trans, etc.) through its operators (e.g., Plugging, Alternation, Conjunction, etc.). IAAnalyzer enables hierarchical connector specification, i.e., two or more simpler connectors can be combined, through the algebra operators, to model a more complex connector. IAAnalyzer drives the user in specifying a model that conforms to the underlying ecore meta-model. For instance, given the ecore meta-model, EMF generates validation code for checking whether the value of a required attribute is missing or not. Nevertheless, some domain-specific constraints can be violated but they can be still detected by means of custom model validation.

2. **Model Validation.** IAAnalyzer automatically validates the designed model with respect to (i) conformance to the underlying meta-model (validation code generated by EMF), and (ii) domain-specific constraints (hard-coded custom validation). For instance, custom validation is required in order to check whether the set of input actions of an IA model and the set of its output actions are disjoint. If the model suffers some constraint violation, the user is informed and the model specification can be fixed accordingly. Otherwise, the model is validated and its IA-based semantics can be generated.

3. **IA-based Semantics Generation.** In order to support automated reasoning, IAAnalyzer automatically generates the IA-based semantics of the specified NSs and connector models. NSs are specified by directly exploiting the IA syntax and, hence, their IA-based semantics generation is straightforward. The connector is specified by combining terms of the algebra by means of the algebra operators. Thus, the IA-based semantics of the connector is automatically generated by exploring its hierarchical structure and applying the operators’ semantics to the respective operands. After IA-based semantics generation, IAAnalyzer has produced an IA for each specified NS and connector.

4. **Ticc Specification Generation.** By means of M2T transformation, coded in Acceleo 3.0, IAAnalyzer automatically translates the generated IA-based semantics into a Ticc textual specification. Ticc deals with a particular case of IA (namely, sociable interfaces) and hence, to perform the transformation, no additional information is required beyond the generated IA-based semantics.

5. **Behavioural Analysis.** As mentioned above, once the Ticc specification is generated, the user can exploit the Ticc functionalities in order to, e.g., automatically check the correctness of the specified connector with respect to the interaction protocol of the specified NSs, i.e., check whether the connector does not introduce wrong interaction behaviours hence making the NSs able to interoperate.
Concerning the last two activities, it is worthwhile noticing that, being implemented by means of Model Driven Development (MDD) technologies such as EMF and Acceleo 3.0, IAAnalyzer is indeed independent from the specific analysis we wish to perform. In other words, by referring to the IAAnalyzer modules that implement the first four activities, IAAnalyzer can be easily integrated with other analysis tools beyond Ticc. In fact, integration with other analysis tools is just a matter of possibly extending the meta-model with new domain concepts (depending on the specific analysis tool) and code new M2T Acceleo transformations. Note that, due to the declarative nature of Acceleo, coding transformations is usually easier than writing imperative/procedural transformation code. Other advantages of our algebra implementation into the IAAnalyzer tool are listed as follows.

- IAAnalyzer has been found to be useful to validate the effectiveness and expressive power of the current version of our algebra, hence suggesting that new characteristics has to be added in the future to improve, e.g., usability of the algebra. For instance, the example described step-by-step in Section 1.2 shows that, in some cases, the interaction behaviour of a compound term of the algebra may be unpredictable for the user. Beyond this issue, that can be addressed by the behavioural analysis activity, the carried example suggested us that our set of operators should be refined to improve predictability of the modelled connectors.

- The.ecore meta-model underlying IAAnalyzer can be considered as a base meta-model for IA-based models and, hence, within CONNECT, it may be exploited also outside WP2.

- Being an EMF tool, IAAnalyzer can be integrated with other EMF tools that are becoming ever more popular in the practice of modelling tool development.

- By having exploited MDD technologies such as EMF, the IAAnalyzer’s code is engineered into a “Model-View-Controller” architecture [10] that supports easy extension of the tool with new features and functionalities.

As mentioned above, we release both the binaries and the source code of IAAnalyzer. Thus, the possible users of IAAnalyzer are both end-users (hereafter, simply called users) and developers. Users use IAAnalyzer for modelling and analysis purposes. Developers use it, e.g., for extending the tool with new functionalities or integrating it with other tools. Therefore, in Sections 1.2 and 1.3, we provide the documentation concerning IAAnalyzer for both users and developers, respectively. Indeed, Section 1.2 also concludes with a discussion about the usability degree of our algebra formalization.

### 1.2 User guide

A convenient way to start to learn how to interact with IAAnalyzer for modelling and analysis purposes is to follow the live tutorial available on the CONNECT web site at the following URL:


The live tutorial has been realized by using the Wink tool [5] and it shows IAAnalyzer at work on a simple but representative example. In this section, we discuss in more detail some steps of the live tutorial. Thus, this user guide should be considered as a detailed tutorial. The example that we show step-by-step, although very simple, suggested us interesting modifications to be done, as future work, on the current algebra formalization.

In order to download the binaries of IAAnalyzer as a stand-alone Eclipse product, the user has to go to the training page at the CONNECT web-site (see the URL at the beginning of Section 1.1) and click on the “IAAnalyzer tool: Eclipse boundle” link. The user gets a “.zip” archive, extracts it and launches the “IAAnalyzer.exe” application (by choosing a workspace directory).

- **Create an empty IAAnalyzer model.**

  Right-click on the left-hand side tab titled “Package Explorer” and, on the displayed contextual menu, choose “New → Project...”. On the displayed dialogue window, choose “General → Project”. Specify a project name (e.g., “UserGuide”) and click on the “Finish” button. Now, on the left-hand side tab, the main
Figure 1.2: Creation of an empty IAAnalyzer model
directory for the project is displayed. Right-click on the project directory and choose “New → Other...”. A dialogue window is displayed as shown in Figure 1.2. Select “Example EMF Model Creation Wizards → IAAnalyzer Model” and click on the “Next” button. Specify the name of the model (e.g., “UserGuide.iaa”), click on “Next”, select “Analyzer Specification” as Model Object, and click on “Finish”. It is worth noticing that, on the Model Object list-box, all the meta-classes of the underlying meta-model are shown. The “Analyzer Specification” meta-class is the root element of the ecore meta-model defined for IAAnalyzer (see Section 1.3 for more details on the IAAnalyzer meta-model). Now, in order to set the attributes of a model element, the user has to enable the “Properties” view. To do this, select the menu “Window → Show View → Other...”. On the dialogue window, choose “General → Properties” and click on “OK”. The IAAnalyzer model is displayed as a tree in the editor tab whose title is equal to the name of the model, e.g., “UserGuide.iaa”. Each node of the tree is a typed element of the model whose type is a meta-class defined in the underlying ecore meta-model, e.g., “Analyzer Specification”. Expand the “platform” node and then click on the “Analyzer Specification” node. When clicking on a node, the “Properties” tab on the bottom-side of the main window shows the attributes of the corresponding model element as defined by the ecore meta-model. The “Analyzer Specification” node has a required attribute that is called “Name”. An element’s attribute is required when it must be set, otherwise a meta-model conformance violation will be detected while performing model validation (menu “IAAnalyzer Editor” → “Validate”). For the sake of brevity, since the error messages displayed by the model validator are intuitive, in this guide, we avoid to report the list of validation problems and their explanation. The interested user can refer to Section 1.3, where the custom validation code is discussed.

- Define the alphabet of actions.
Right-click on the “Analyzer Specification” node. A contextual menu is displayed as shown in Figure 1.3. It shows all child elements of “Analyzer Specification” as defined in the ecore meta-model. Thus, the user does not have to always account for meta-model conformance since IAAnalyzer drives him/her in specifying a conforming model. Among all possible child elements, the “Action” element allows the user to add an action to the alphabet of actions on top of which the NSs and connector algebraic specification will be built.

By clicking on “New Child” → “Action”, a new “Action” element is created as a child of “Analyzer Specification”. It has a required attribute, “Label”, used to specify the action label.

- Define the IA-based specification of the NSs.
By referring to Figure 1.3, click on “Networked System” and a new “Networked System” element is added as a child of the model root node. It has a required attribute “Name”. By right-clicking on “Networked System” and choosing “New Child” → “Interface Automaton”, a new IA specification for the NS is created. “Interface Automaton” is the only child of “Networked System” that hence seems to be a useless model element. Indeed “Networked System” has been kept in the meta-model for future purposes. For instance, the IA element represents the interface model of the NS element. If, in the future, we want to, e.g., combine quantitative and interface models, probably, the definition of the NS meta-class will change leading a NS element to have other child elements.

By clicking on the IA node and by using the associated “Properties” tab, input and output actions for the IA can be specified. They will be selected from the defined alphabet of actions. As it is for all the other elements, “Name” is a required attribute of “Interface Automaton”. By right-clicking on “Interface Automaton” and choosing “New Child” → “State” or “New Child” → “Transition”, the user can define respectively a state or a transition for the IA.

A state has two attributes, “Id” and “Initial”. The former is an integer used to uniquely label a state (i.e., the user must set it in a way that two different states have different values for their “Id” attributes). The latter is a boolean used to distinguish the initial state (“Initial” equal to “true”) from non-initial states (“Initial” equal to “false”). An IA must have exactly one initial state.

A transition has three attributes, “Action”, “Source”, and “Target”. The value of “Action” is chosen among the actions specified in the defined alphabet (see above). The values of “Source” and “Target” are chosen among all the specified states. By referring to Figure 1.4, the value for each of these three attributes is set by means of a list-box that shows all the possible values that can be assigned depending on the type of the attribute.
In the right-hand side of Figure 1.5, we show the IA models of the two NSs, “CFring” and “XMPP”, which we consider in this user guide as a simple explanatory example. In the left-hand side of Figure 1.5, we show the corresponding IAAnalyzer model.

By exploiting the explanations that we provided so far, the user can easily reproduce what is shown in Figure 1.5.

Define the algebraic specification of the connector, generate its IA-based semantics and its Ticc specification.

In this step, we want to specify the connector for our example by using the algebra operators and primitives. By referring to Figure 1.5, we recall that CFring, when sending a message, expects to send the message’s content first (action “message”) and, then, the sender identifier (action “id”). Instead XMPP, when receiving a message, expects to receive the sender identifier first (action “who”) and, then, the message’s content (action “msgin”). Thus, by using IAAnalyzer, we want to model, as a compound term of the implemented algebra, a connector that reorders the sequence (“message” “id”) into the sequence (“id” “message”) and, then, translates “id” (resp., “message”) into “who” (resp., “msgin”). In other words, by means of IAAnalyzer, we want to show how to model the connector that is expressed below, by using the algebra syntax.

\[
\text{Order([message, id],(2,1),[message1, id1])} \odot \text{Trans(id1,who)} \odot \text{Trans(message1,msgin)}
\]

By referring to the contextual menu shown in Figure 1.3, beyond the “Networked System” and “Action” elements, all the possible connector elements that can be defined are shown. According to the formalization of our algebra, they concern the definition of two types of compound connectors (“Composite Binary Connector” and “Composite Unary Connector”) and seven types of primitive connectors (“No Op”, “Cons”, “Prod”, “Trans”, “Split”, “Merge”, and “Order”). By right-clicking on “Analyzer Specification” and choosing “New Child” ➔ “Composite Binary Connector” the user can define a connector built as a compound term of the algebra which is based on the application of some binary operator (e.g., plugging). A “Composite Binary Connector” node appears, it has three attributes, “Name” (required), “Comments” (optional), and
Figure 1.4: Labelling a transition
"Semantics". The last attribute is a derived attribute meaning that it is not set by the user, i.e., its value is automatically computed as a function of other model elements. Each connector element in the model has this attribute, it does not matter if the connector element is binary, unary, primitive, or compound. Whenever all the attributes and parameters of a connector element have been instantiated\(^2\), IAAnalyzer automatically generates an "Interface Automaton" element for it and initializes the value of its "Semantics" attribute with a reference to the generated IA element. Connector specification is a task that can be carried on hierarchically. That is, we can define, e.g., a composite binary connector as part of another composite binary connector and so on and so forth. Leaves of the hierarchy will be primitive connectors of the algebra. For all the elements of the hierarchy, the "Interface Automaton" element referenced by "Semantics" is stored into the IAAnalyzer.ecore model, clearly as part of a new "Networked System". For the leaves and intermediary elements, it is stored as a hidden element, i.e., it is not shown in the model editor. For the root element (e.g., a "Composite Binary Connector" element at the root of the hierarchy), it is stored as a displayable element. That is, whenever the root element specification is finalized, by simply clicking on it, its corresponding IA will be displayed as an additional "Interface Automaton" that is child of an additional "Networked System" (see the NS specification "Plug1" in Figure 1.8).

Now, by following the discussion above, let us give a name to the added "Composite Binary Connector", e.g., "Plug1", and by right-clicking on it, let us choose "New Child" → "Plugging" as shown in Figure 1.6. A new "Plugging" element is added, it has only a "Comments" attribute that is optional. According to our algebra formalization, Figure 1.6 shows also the other possible binary operators that can be used to specify a composite binary connector.

With the above connector algebraic specification in mind, starting from the "Plugging" element, the user can define the connector hierarchically. That is, the left child of "Plugging" is a "Composite Binary

\(^2\)For instance, the left and right operands of a "Plugging", or the input and output ports of a "Trans".
Figure 1.6: Definition of a composite binary connector
Connector” and the right child is a “Trans”. Then, the lowermost “Composite Binary Connector” is defined by means of another “Plugging” element whose left and right childs are a “Order” and a “Trans”, respectively. In order to specify the childs of the uppermost “Plugging” element, right-click on it and choose “New Child” → “Left Op Composite Binary Connector” and, subsequently, “New Child” → “Right Op Trans” as shown in Figure 1.7.

Figure 1.7: Definition of a plugging operator

By analogously proceeding for all the other connector elements to be defined, the user can model the connector for our CFring/XMPP example, as it is shown in Figure 1.8. In order to avoid to overwhelm the reader, for the definition of the used primitives (“Order” and “Trans”) and the instantiation of their attributes, we refer to the available live tutorial (see the URL above).

If the user simply clicks on the composite binary connector “Plug1”, IAAnalzyer automatically generates its IA-based semantics and shows it as an additional “Networked System” element whose name is “Plug1”, as it is shown in Figure 1.9.

Once the user has specified the NSs and the connector, by exploiting the M2T transformation facilities implemented into IAAnalzyer, the user can automatically build the equivalent Ticc textual specification in order to check, e.g., whether the specified connector achieve interoperability or not. This can be done by right-clicking on the file “UserGuide.iaa” shown in the “Package Explorer” tab, and choosing the menu “Acceleo Model to Text” → “Generate iaa2ticc”, as it is shown in Figure 1.10.

In the IAAnalzyer project workspace, the performed M2T transformation generates a directory, “Ticc-Spec”, plus some files as shown in Figure 1.11. There is a “.si” file for each “Networked System” element. Furthermore, there is a “.in” file that, by using the textual notation of Ticc, specifies the composition of the IA models defined by the “.si” files. For the sake of brevity, for details concerning both the content of the generated files and the way to check composability by using Ticc, we refer to both the live tutorial and the work described in [1, 3, 11].

As it is shown by the live tutorial, although very simple, the example discussed in this user guide highlights, on one hand, the importance of having IAAnalzyer integrated with analysis tools and, on the
Figure 1.8: Algebraic specification of the connector for our CFring/XMPP example

Figure 1.9: IA model of the connector for our CFring/XMPP example
Figure 1.10: Generation of the Ticc specification
other hand, that our preliminary algebra formalization has to be improved in terms of its usability. In fact, although IAAnalyzer implements an high-level connector algebra that has been conceived with the aim of easing connector modelling, due to the current formalization of the plugging operator which is mainly based on the IA parallel composition operator, the user might not always be able to predict the result of its modelling. The composability check, that the user can perform by having IAAnalyzer integrated with Ticc, reveals that the connector shown in Figure 1.8 may not let “CFring” and “XMPP” able to interoperate, although intuitively it might seem to model a connector that always achieves interoperability.

We recall that when composing two IA in parallel, first the product between the two IA is performed. It lets the two IA synchronize on common input/output actions and independently progress on non-common actions. If there is no error state\(^3\) in the product, the result of the parallel composition is the same as the one of the product. Otherwise all the traces that always lead to error states are pruned, hence producing the result of the parallel composition. Since the environment can prevent only input actions of a component\(^4\), while pruning these traces, from an error state \(err\), backwards error propagation is performed until encountering the first state from which both \(err\) is reachable only by performing an input action and there exist at least an action that does not lead to an error state.

\[\text{Order([message, id], (2, 1), [message1, id1])} \ Join \ Trans(id1, who) \ Join \ Trans(message1, msgin)\]

\[\text{Order([message, id], (2, 1), [message1, id1])} \ Join \ Order([id1, message1], (1, 2), [who, msgin]) \ Join \ Order([who, msgin])\]

![Figure 1.12: Uncorrect connector (on the top), correct connector (on the bottom)](image-url)

Coming back to our example, the top-side of Figure 1.12 shows the IA of the modelled connector\(^5\) plus its algebraic specification. When building the product between the IA of the connector, of “CFring”, and of “XMPP”, there is a sequence of interactions that would let the two NSs able to interoperate and there is one that would not achieve interoperability, hence detecting an error state. The former would let the connector exchange with the two NSs (and the environment) the messages in the sequence “message?” “id?” “id1!” “who!” “message1!” “msgin!”, hence making “CFring” and “XMPP” always able to interoperate. The latter would let the connector reach the state ‘5’ that is an error state. Since error states can be

---

\(^3\)An error state is a sink state or a state from which only error states can be reached.

\(^4\)Within the IA setting, output actions are always uncontrollable since, realistically, a component plays an active role with respect to them, whereas it is a passive entity on inputs.

\(^5\)As usual, the suffix '?' (resp., '!') denotes input (resp., output) actions.
prevented by acting only on input actions, the result of the parallel composition is “empty” as also revealed by the message printed as output by the Ticc composability check: "The initial condition of module CFrинг*XMPP*Plug1 is empty: this is a sign of incompatibility!" (see the live tutorial).

The bottom-side of Figure 1.12 shows the IA of the correct connector plus its algebraic specification. It is the connector that lets “CFrинг” and “XMPP” able to interoperate. Although correct, its algebraic specification reveals that for a user it can be unintuitive to write it down since he/she should use the “Order” primitive on the rightmost side of the expression to translate messages rather than to re-order them. Thus, due to the current formalization of the algebra, specifying the correct connector would need to make an inappropriate use of the “Order” primitive. Nevertheless, this conclusion should be considered as a positive one with respect to the overall work done so far about the connector algebra and its related implementation. In fact, IAAnalyzer allowed us to raise, at an early stage of our formalization, an expressiveness issue of the current version of the algebra that, without the validation carried on, would not have been trivial to discover or would have been discovered late. Possible ideas to be developed as future work in order to solve the raised issue are discussed in Section 1.4.

1.3 Developer guide

The IAAnalyzer’s code is released as an open source software. In order to make developers able to either extend it with new functionalities or integrate it with new features, in this section, we provide information about the technologies used to develop IAAnalyzer and how its implementation code has been structured. The discussion is organized with respect to the six Eclipse projects that we have developed to built IAAnalyzer.

To develop IAAnalyzer, we have used Eclipse Ganymede, version number 3.4.2 and build id M20090211-1700. By means of the Ganymede Update Site, we have installed the EMF and Graphical Modelling Framework (GMF) development and run-time support. Instead, the development and run-time support for the Acceleo 3.0 M2T transformations, can be installed by following the instructions available on: http://www.eclipse.org/acceleo/download/.

In order to download the sources of IAAnalyzer, as a set of six Eclipse projects, the developer has to go to the training page at the CONNECT web-site (see the URL at the beginning of Section 1.1) and click on the “IAAnalyzer tool: Source code” link. The developer gets a “.zip” archive. By referring to the requirements mentioned above, once the developer has configured a suitable Eclipse distribution, he/she can import the projects contained in the downloaded archive into his/her workspace. This can be done by right-clicking on the “Package Explorer” tab (left-hand side of the Eclipse application window) and choosing “Import”. On the displayed dialogue window, the developer has to select “General” → “Existing Projects into Workspace” and click on “Next”. Choice “Select archive file” and browse the file system in order to select the downloaded “.zip” archive. Make sure that all shown projects are selected and that the check-box “Copy projects into workspace” is selected. Click on “Finish” and the six Eclipse projects constituting the IAAnalyzer sources have been imported in the developer’s workspace. In the following, we discuss each of these projects.

- IAAnalyzer Eclipse Project.

The IAAnalyzer project is an “EMF Project” that defines the IAAnalyzer.ecore meta-model. The meta-model is stored into the “IAAnalyzer.ecore” XMI file under the directory “model”. The content of “IAAnalyzer.ecore” can be conveniently browsed or modified by using the provided EMF editor (tree-view based). EMF provides also generation facilities to generate an ecore diagram out of “IAAnalyzer.ecore”, see “IAAnalyzer.ecorediag” under the directory “diagrams”. As shown in Figure 1.13, “IAAnalyzer.ecorediag” encodes a diagrammatic representation of “IAAnalyzer.ecore”, which is given in terms of meta-classes and relationships among them, that is suitable for presentation purposes.

The EClass “AnalyzerSpecification” is the root element of the meta-model. As shown in Figure 1.13, it contains a list of reference elements to EClass “NetworkedSystem”, a list of reference elements to EClass “Action”, and a reference element to EClass “ConnectorSpecification”. “NetworkedSystem”, in turn, contains a reference element to EClass “InterfaceAutomaton” that is a container for elements of type

---

This can be done by selecting the items “Models and Model Development” and “Graphical Editors and Framework” from the project-tree shown for the Ganymede Update Site.
Figure 1.13: The IAAnalyzer meta-model
“State” and “Transition”, hence implementing a data structure for IA. Furthermore, “InterfaceAutomaton” contains two lists of reference elements to “Action” so as to implement the lists of input and output actions of the IA. The “getInitial()” operation retrieves an instance of “State” that represents the designated initial state of the IA.

The most interesting part of the IAAnalyzer meta-model is represented by the EClass “ConnectorSpecification” that is the container for all the EClasses that implement our algebra formalization. As it can be seen, the meta-model for a connector specification is defined recursively, hence leading to a hierarchical specification model. Beyond being specialized by EClasses that serve to model the primitives of the algebra, “ConnectorSpecification” is specialized also by “CompositeUnaryConnector” and “CompositeBinaryConnector”. Let us discuss only the latter since the discussion is analogous for the former. “CompositeBinaryConnector” is a container for “BinaryOperator” that, in turn, is specialized by EClasses that allow the modelling of the binary operators of the algebra. “BinaryOperator” contains two reference elements, “leftOp” and “rightOp”, that model the two operands of the binary operator. Their type is, again, “ConnectorSpecification” hence allowing the definition of a hierarchical connector model. The operation “getSemantics()” serves to compute the value of the “semantics” derived attribute (see Section 1.2), i.e., the “InterfaceAutomaton” element that implements the IA corresponding to the semantics of the modelled connector. Note that, at the level of the meta-model, the “semantics” attribute is defined as an association between “ConnectorSpecification” and “InterfaceAutomaton”.

The most interesting part of the IAAnalyzer meta-model is represented by the EClass “ConnectorSpecification” that is the container for all the EClasses that implement our algebra formalization. As it can be seen, the meta-model for a connector specification is defined recursively, hence leading to a hierarchical specification model. Beyond being specialized by EClasses that serve to model the primitives of the algebra, “ConnectorSpecification” is specialized also by “CompositeUnaryConnector” and “CompositeBinaryConnector”. Let us discuss only the latter since the discussion is analogous for the former. “CompositeBinaryConnector” is a container for “BinaryOperator” that, in turn, is specialized by EClasses that allow the modelling of the binary operators of the algebra. “BinaryOperator” contains two reference elements, “leftOp” and “rightOp”, that model the two operands of the binary operator. Their type is, again, “ConnectorSpecification” hence allowing the definition of a hierarchical connector model. The operation “getSemantics()” serves to compute the value of the “semantics” derived attribute (see Section 1.2), i.e., the “InterfaceAutomaton” element that implements the IA corresponding to the semantics of the modelled connector. Note that, at the level of the meta-model, the “semantics” attribute is defined as an association between “ConnectorSpecification” and “InterfaceAutomaton”.

A further key aspect of the defined meta-model concerns its support to model validation. As mentioned in Section 1.2, starting from the meta-model, EMF generates validation code for the defined structural constraints, e.g., checking whether required attributes have a value different from “null”. Beyond these structural checks, the developer may wish to write validation code also for domain-specific checks, e.g., the two sets of input and output actions of an IA must be disjoint. Clearly, the developer wish to do that by exploiting the APIs of the EMF validator. As shown in Figure 1.14, this can be done by defining ecore annotations on the EClasses over which custom validation should be performed. For instance, by referring to Figure 1.14, the ecore annotation “DisjointInAndOutActionSets”, defined on “InterfaceAutomaton”, allows EMF to generate the skeleton code of a Java class which exports a public method that implements the constraint expressed by that annotation. This public method is invoked when performing model validation as discussed in Section 1.2. First, EMF standard validation is performed by means of the methods
inherited by “EObjectValidator” (it implements the EMF API for performing model validation). Then, the
domain-specific validation methods are invoked hence performing custom validation. The advantage for
the developer is that he/she has to write only the custom validation code within the body of the generated
skeleton code without worrying about details on how model validation is performed by EMF. Thus, he/she
can be concentrated only on the domain-specific semantics of the validation check. Just to give an intu-
ition, the listing below shows the Java code for the above mentioned ecore annotation. All the code has
been automatically generated by EMF, directly out of the meta-model, except for the code lines from 19 to
28, which is the domain-specific validation code that the developer has hard-coded.

```java
package it.univaq.modeling.IAAnalyzer.util;
import org.eclipse.emf.ecore.util.EObjectValidator;
...
/**
 * <!-- begin - user - doc -->
 * The <b>Validator</b> for the model.
 * <!-- end - user - doc -->
 * @see it.univaq.modeling.IAAnalyzer.IAAnalyzerPackage
 * @generated
 */
public class IAAnalyzerValidator extends EObjectValidator {
...
public boolean validateInterfaceAutomaton_DisjointInAndOutActionSets(InterfaceAutomaton
interfaceAutomaton, DiagnosticChain diagnostics, Map<Object, Object> context) {
  boolean violation = false;
  Iterator<Action> itIn = interfaceAutomaton.getInactions().iterator();
  while ((itIn.hasNext()) && (! violation)) {
    Iterator<Action> itOut = interfaceAutomaton.getOutactions().iterator();
    Action inAct = itIn.next();
    while ((itOut.hasNext()) && (! violation)) {
      Action outAct = itOut.next();
      if (inAct.getLabel().equals(outAct.getLabel()))
        violation = true;
    }
  }
  if (violation) {
    if (diagnostics != null) {
      diagnostics.add
        (createDiagnostic(Diagnostic.ERROR,
         Diagnostic.ERROR,
         "_UI_GenericConstraint_diagnostic",
         new Object[]{ "DisjointInAndOutActionSets",
         interfaceAutomaton, context });
    }
    return false;
  } return true;
}
...
```

Listing 1.1: An example of custom validation

The above code is part of a generated (by EMF) Eclipse project different from IAAnalyzer, whose
name is it.univaq.modeling.iaanalyzer. This project and all the other generated ones are discussed
below.

- it.univaq.modeling.{iaanalyzer, iaanalyzer.edit, iaanalyzer.editor} Eclipse Projects.
These three projects realize the Model-View-Controller architecture of the IAAnalzyer tool. They are automatically generated by EMF directly out of the meta-model defined by the IAAnalyzer Eclipse Project. It.univaq.modeling.iaanalyzer represents the Java implementation of the meta-model. It defines Java classes for the programmatic definition of a model conforming to the defined meta-model and for the model run-time management (e.g., change notification, model serialization and reflection) and validation. This project allows also for the integration with other EMF-based tools. Roughly speaking it implements the “Model” part of the Model-View-Controller architecture of IAAnalzyer. It.univaq.modeling.iaanalyzer.edit implements the “Controller” part of the architecture. It defines a set of adapter Java classes that enable viewing and command-based editing of the model. Its main advantage is in the fact that it is presentation independent, i.e., it does not matter how the model layout is implemented (by the model editor), the viewing and editing logic is encapsulated into this project. Thus, a developer can change the model layout without impacting on the code generated or written for it.univaq.modeling.iaanalyzer.edit (and also for it.univaq.modeling.iaanalyzer). It.univaq.modeling.iaanalyzer.editor is the “View” part of the IAAnalzyer architecture. As it is generated by EMF, it implements the tree-view based editor whose user interaction has been discussed in Section 1.2. Essentially, it defines the Java classes for implementing the model presentation code and it is based on SWT/JFace user interaction.

In order to implement the current version of IAAnalzyer, no modification has been done over the code generated by EMF for it.univaq.modeling.iaanalyzer.editor, i.e., IAAnalzyer completely relies on the tree-view based editor generated by EMF. We have just written custom presentation code by applying few modifications to some Java classes in it.univaq.modeling.iaanalyzer.edit. In particular, we customized the code of the method “getText” for both classes “it.univaq.modeling.IAAnalzyer.provider.StateItemProvider.java” and “it.univaq.modeling.IAAnalzyer.provider.TransitionItemProvider.java”. The method “getText” is invoked by the editor when showing the textual layout of a meta-model element, e.g., a state or a transition of an IA model. Since, the code generated by EMF provides a very simple textual representation that, e.g., does not allow the user to distinguish between states with the same label that belong to two different IA, we decided to slightly modify the generated presentation code. However, most of the code hard-coded by us concern the it.univaq.modeling.iaanalyzer project. Thus, hereafter, we concentrate on discussing how this project is implemented.

It.univaq.modeling.iaanalyzer is structured into three packages: It.univaq.modeling.IAAnalzyer, IAAnalzyer.impl, IAAnalzyer.util. For each EClass in the meta-model, it.univaq.modeling.IAAnalzyer defines a Java interface that extends the EMF interface EObject. This Java interface is, essentially, the Java translation of the corresponding EClass in the meta-model. For instance, the EClass “InterfaceAutomaton” shown in Figure 1.13 is translated into the “InterfaceAutomaton” Java interface that is partially shown below.

```java
public interface InterfaceAutomaton extends EObject {

  /**
   * @generated
   */
  String getName();

  /**
   * @generated
   */
  void setName(String value);

  /**
   * @generated
   */
  EList<State> getStates();

  ...

```
* @generated
*/
EList<Transition> getTransitions();

/**
 * @generated
*/
EList<Action> getInactions();

/**
 * @generated
*/
EList<Action> getOutactions();

/**
 * @generated
*/
State getInitial();

/**
 * @generated NOT
*/
public void determinise(...);

Listing 1.2: The “InterfaceAutomaton” interface

As it can be seen, for each attribute of the corresponding EClass, putter and setter method declaration is generated (e.g., “getName” and “setName”). Furthermore, a method declaration is generated also for each operation of the EClass (e.g., “getInitial”) and for each relationship with other EClasses (e.g., “getInactions”). The annotation “@generated” serves to indicate that it is a method declaration generated by EMF. For those methods hard-coded by the developer, the annotation “@generated NOT” is used; see the method declaration for “determinise” that has been added by the developer to determinise a non-deterministic IA (useful when performing alternation).

For each Java interface in it.univaq.modeling.IAAnalyzer, the package it.univaq.modeling.IAAnalyzer.impl defines a Java class implementing that interface. Again, the implementation of the methods annotated by “@generated” has been completely generated by EMF, e.g., the implementation of getter and putter methods. The implementation of the methods annotated by “@generated NOT” is hard-coded by the developer, e.g., the implementation of “getSemantics” for each “ConnectorSpecification” type and its specializations, or the implementation of all the operations that can be performed on an “InterfaceAutomaton” type, such as the inversion of an IA. The following listing shows a fragment of the implementation code for the class “NoOpImpl”, in particular for its “getSemantics” method. This class implements the interface “NoOp” and it is a specialization of “ConnectorSpecificationImpl” (that, in turn, implements the interface “ConnectorSpecification”).

```java
public class NoOpImpl extends ConnectorSpecificationImpl implements NoOp {
...
/*
  * @generated NOT
*/
public InterfaceAutomaton getSemantics() {
  // let's see if all the parameters are initialized so that we can generate the semantics of the primitive (NoOp has no parameters)
  InterfaceAutomaton res = IAAnalyzerFactory.eINSTANCE.createInterfaceAutomaton();
  res.setName(name);
}
```
State initialState = IAAnalyzerFactory.eINSTANCE createState();
initialState.setId(0);
initialState.setInitial(true);
res.getStates().add(initialState);

return res;
}

Listing 1.3: Implementation of the IA model construction for a NoOp connector

The following listing shows the implementation of the method "inversion" of the class "InterfaceAutomatonImpl". This class implements the interface "InterfaceAutomaton" and the "inversion" method can be invoked to built the inverted IA.

public class InterfaceAutomatonImpl extends EObjectImpl implements InterfaceAutomaton {
...
/**
... * The semantics of the inverted automaton is equal to the semantics of "this" where
* the input (resp., output) actions of the inverted are the output (resp., input)
* actions of "this"
... */
public InterfaceAutomaton inversion(String name) {
    InterfaceAutomaton inversion = IAAnalyzerFactory.eINSTANCE.createInterfaceAutomaton();
    inversion.setName(name);
inversion.getStates().addAll(getStates());
inversion.getTransitions().addAll(getTransitions());
inversion.getInactions().addAll(getOutactions());
inversion.getOutactions().addAll(getInactions());

    return inversion;
}
...

Listing 1.4: Implementation of the IA inversion operation

Again what is worth noticing is that, by relying on the EMF code generation and on the way it structures the code, the developer can concentrate on writing the domain-specific Java code without knowing too much details about EMF and its run-time support.

The package it.univaq.modeling.IAAnalyzer.util contains the Java implementation of the custom validation code, which has been already discussed above (see Listing 1.1), and typically all the other Java classes required to accomplish further auxiliary tasks beyond custom validation. In the remainder of this section we discuss the two Eclipse Projects that, by exploiting Acceleo M2T transformation, allow to automatically translate an IAAnalyzer model specification into a Ticc textual specification.

- it.univaq.modeling.{iaanalyzer.ia2ticc, iaanalyzer.ui} Eclipse Projects.
  it.univaq.modeling.iaanalyzer.ia2ticc is an Acceleo 3.0 project ("New Project" → "Acceleo Model to Text" → "Acceleo Project"). Typically, an Acceleo project has a package (it.univaq.modeling.iaanalyzer.ia2ticc.files in our case) designated to contain the M2T template files and the .java files of the so called transformation classes. A M2T template is hard-coded by the developer by using the Acceleo language. A transformation class is automatically generated by Acceleo out of the meta-model and the M2T templates. Essentially, a transformation class is a Java class that implements the transformation defined by a M2T template. Thus, given the IAAnalyzer meta-model, the only thing to do, in order to build the M2T transformation towards the Ticc notation, is to code the required M2T template file. Listing 1.5 shows our main M2T template (i.e., "iaa2in"); it is analogous to the main program written in some programming language (see the annotation @main on line 5). This template uses another M2T template (see the code [import iaa2si/] on line 1) that can be invoked as a function in

8Their extension is ".mtl".
a functional programming language (see the code [iaa2si(ns.protocol)/] on line 20). Acceleo is a tag-based language. Examples of tag are import, template, or for. All the text within the brackets (i.e., within `[` and `]`) is Acceleo code that must be interpreted in order to perform the defined transformation. It is expressed, e.g., by using tags or by invoking some template. As in any other tag-based language, by using a familiar terminology, tags must be “opened” and “closed”; see, for instance, [import iaa2si/] (opened and closed on one single line) or [template ...] ... [/template] (opened on line 3 and closed on line 42). All the Acceleo code within the (opened and closed) tag is the body of the statement expressed by the tag. Instead, analogously to scripting languages (e.g., Javascript), all the text outside the brackets is interpreted as plain text and, hence, it is reproduced as output as it is. Thus, the Acceleo language eases the task of writing M2T transformations. In fact, a relevant part of the transformation text is simply the fixed part of the output text. Concerning the parametric part of the output text, it is expressed by using a declarative notation very similar to first-order logic that, e.g., eases the way the content of a model element is browsed (see the code at lines 23 to 25).

A M2T transformation in Acceleo deals with elements of a model that is conforming to a given metamodel. It takes as input a model element and, typically, as output, produces a text that is parametric in the value of the considered model element. By referring to Listing 1.5, the main template takes as input an element of type “AnalyzerSpecification” (i.e., the root of the model) and browses its list of NS models and the defined alphabet of actions in order to produce a “.in” Ticc file. Its textual content will result in the specification of the parallel composition of the Ticc interface models that correspond to the IA models of the considered NS elements. The tag file serves to create the “.in” output file, and all the code between [file ...] and [/file] produces its textual content.

```acceleo
[import iaa2si/]
[template public iaa2in (as : AnalyzerSpecification)]
[comment @main /
[file (as.name.concat('.in'), false, 'Cp1252')]
open Ticc;;

(*
NOTE: If you do not have included the Ticc directory in the
definition of your PATH environment variable (see the hidden
file .profile in your home directory), you should replace the
following paths with the absolute paths of the *.si files. Otherwise,
do not change anything and type "ticc [as.name.concat('.in')]"
in the same directory where [as.name.concat('.in')] and the
referenced *.si files are.
*)

[file /
[for (ns : NetworkedSystem | as.nslist)]
parse "'[ns.protocol.name.concat('.si')]'"; [iaa2si(ns.protocol)]
[/for]

[for (ns : NetworkedSystem | as.nslist)]
let [ns.protocol.name.toLowerCase()] = mk_sym "'[ns.protocol.name]'";;
[/for]

[if (as.nslist->size() = 0 or as.nslist->size() = 1)]
(* nothing to compose *)
[else]
let composedsystem = compose [as.nslist->at(1).protocol.name.toLowerCase()] [as.nslist->at(2).protocol.name.toLowerCase()];;
[for (ns : NetworkedSystem | as.nslist->at(2).followingSiblings(NetworkedSystem))
let composedsystem = compose composedsystem [ns.protocol.name.toLowerCase()];;
[/for]
[/if]

[for (act : Action | actions)]
print_input_restriction composedsystem "'[act.label]'";;
[/for]
[/file]
[/template]
```

Listing 1.5: M2T transformation from an IAAnalyzer model to a Ticc specification (Acceleo 3.0 code)
The transformation from an IA model to a Ticc interface model (i.e., iaa2si) is defined by Listing 1.6. Ticc requires that each interface model is stored into a .si file. Each .si file is referenced by the .in file that specifies the IA parallel composition (as discussed above). Thus, the M2T transformation expressed by Listing 1.6 produce a .si file for each model element of type “InterfaceAutomaton”. The content of a .si file is a textual representation a sociable interface; IA are a particular case of sociable interfaces.

```
[comment encoding = Cp1252 ]
[module iaa2si('it.univaq.modeling.iaanalyzer')]
[template public iaa2si(ia : InterfaceAutomaton)]
[file (ia.name.concat('.si'), false, 'Cp1252')]
module [ia.name]:
  var s: [''/0..[ia.states->size()-1]`]`
initial: s = 0

  [for (inact : Action | ia.inactions)]
  input [inact.label]/: {
    [let trList : Sequence(Transition) = ia.transitions->select(trans : Transition |
      trans.action.label.equalsIgnoreCase(inact.label)) asSequence()]
    [if (trList->size() > 0)]
      local: s = [trList->at(1).source.id] => s' := [trList->at(1).target.id]/
      [for (tr : Transition | trList->at(1).followingSiblings(Transition))]
        [if (tr.action.label.equalsIgnoreCase(inact.label))]
          else s = [tr.source.id] => s' := [tr.target.id]/
        [/if]
    [/for]
  [/let]
  [/for]

  [for (outact : Action | ia.outactions)]
  output [outact.label]/: {
    [let trList : Sequence(Transition) = ia.transitions->select(trans : Transition |
      trans.action.label.equalsIgnoreCase(outact.label)) asSequence()]
    [if (trList->size() > 0)]
      s = [trList->at(1).source.id] => s' = [trList->at(1).target.id]/
      [for (tr : Transition | trList->at(1).followingSiblings(Transition))]
        [if (tr.action.label.equalsIgnoreCase(outact.label))]
          ; s = [tr.source.id] => s' = [tr.target.id]/
        [/if]
    [/for]
  [/let]
  [/for]
endmodule
[/file]
[/template]
```

Listing 1.6: M2T transformation from IAAnalzyer interface models to Ticc interface models (Acceleo 3.0 code)

It.univaq.modeling.iaanalyzer.ui is an Acceleo UI Launcher Project (“New Project” → “Acceleo Model to Text” → “Acceleo UI Launcher Project”). By means of a project wizard, it can be completely generated by Acceleo directly out of it.univaq.modeling.iaanalyzer.ia2ticc, i.e., the Acceleo Project that implements the M2T transformations to be performed. It.univaq.modeling.iaanalyzer.ui is generated as an Eclipse plug-in with all necessary code for right-clicking on a .iaa file (i.e., an IAAnalzyer model file), shown in the “Package Explorer” tab, and launching, through the menu “Acceleo Model to Text” → “Generate iaa2ticc” (shown in Figure 1.10), the transformation expressed by Listing 1.5. In other words, it implements a popup-menu-extension for the it.univaq.modeling.iaanalyzer.ia2ticc Eclipse plug-in that, in turn, depends on the it.univaq.modeling.iaanalyzer plug-in.

We conclude this developer guide by summarizing the dependencies among the discussed Eclipse plug-ins as follows:
• `it.univaq.modeling.iaanalyzer.edit` depends on `it.univaq.modeling.iaanalyzer`  
• `it.univaq.modeling.iaanalyzer.editor` depends on both `it.univaq.modeling.iaanalyzer` and `it.univaq.modeling.iaanalyzer.edit`  
• `it.univaq.modeling.iaanalyzer.iaa2ticc` depends on `it.univaq.modeling.iaanalyzer`  
• `it.univaq.modeling.iaanalyzer.ui` depends on `it.univaq.modeling.iaanalyzer.iaa2ticc`  

### 1.4 Future work

It seems that the high-level algebra that is implemented into the IAANalyzer tool allows a designer to easily and intuitively specify complex connectors, although further justification would be required for this claim based on further case studies. For instance, the usability issue raised by the example described in Section 1.2 suggests us that our set of operators should be refined so as to either include an operator that behaves analogously to *sequencing* in regular expressions or conceive a notion of *priority* to suitably filter the interactions exhibited by the plugging operator, in a way similar to what is done in BIP [4].

As further future work on the IAANalyzer tool, we will extend its front-end by developing (i) a graphical editor that is more effective, for modelling purposes, than the standard tree-view-based editor generated by EMF; and (ii) a textual editor that is unavoidably needed for the modelling of real-scale examples. The former can be developed by exploiting the GMF Tooling Project [9], which provides a model-driven approach to generating graphical editors in Eclipse. In particular, EuGENia GMF [8] is a convenient tool to automatically generates a GMF editor from an annotated EMF meta-model. Through the CONNECT life-time, we will also keep the current IAANalyzer implementation up-to-date according to each future evolution of our algebra formalization, e.g., combination of quantitative and interface models, connector (code) synthesis, integration with Prism, etc. Last but not least, we have to extensively validate the tool with more complex examples.
Bibliography


