Using Transformation Rules to Align Requirements and Architectural Models

Monique Soares, Carla Silva, Gabriela Guedes, Jaelson Castro, Cleice Souza, Tarcisio Pereira
Centro de Informática
Universidade Federal de Pernambuco – UFPE
Recife, Brasil
{mcs4, cttls, ggs, jbc, tcp}@cin.ufpe.br

Abstract— In previous works we have defined the STREAM strategy to align requirements and architectural models. It includes four activities and several transformations rules that could be used to support the systematic generation of a structural architectural model from goal oriented requirements models. The activities include the Preparation of Requirements Models, Generation of Architectural Solutions, Selection of Architectural Solution and Refinement of the Architecture. The first two activities are time consuming and rely on four horizontal and four vertical transformation rules which are current performed manually, requiring much attention from the analyst. For example, the first activity consists of the refactoring of the goal models, while the second one derives architectural models from the refactored i* (iStar) models. In this paper we automate seven out of the eight transformation rules of the two first activities of STREAM approach. The transformation language used to implement the rules was QVTO. We rely on a running example to illustrate the use of the automated rules. Hence, our approach has the potential to improve the process productivity and the quality of the models produced.

Keywords— Requirements Engineering, Software Architecture, Transformation Rules, Automation

I. INTRODUCTION

The STREAM (A Strategy for Transition between Requirements Models and Architectural Models) is a systematic approach to integrate requirements engineering and architectural design activities, based on model transformation, to generate architectural models from requirements models [1]. It generates structural architectural models, described in Acme [4] (the target language), from goal-oriented requirements models, expressed in i* (iStar) [3] (i.e. the source language). This approach has four activities, namely: Prepare Requirements Models, Generate Architectural Solutions, Select Architectural Solution and Refine Architecture.

The first two activities are time consuming and rely on horizontal and vertical transformation rules (HTRs and VTRs), respectively. Currently, these transformations rules are made manually, requiring much attention from the analyst. However, they seem likely to be automated, which could reduce not only the human effort required to generate the target models, but also minimize the number of errors produced during the process. Hence, our proposal is to use the QVT [2] transformation language to properly define the rules, and also to develop some tool support to execute them.

Therefore, two research questions are addressed by this paper: Is it possible to automate the transformation rules defined in the first two STREAM activities, namely: Prepare Requirements Models, Generate Architectural Solutions? And, if so, how could these rules be automated? Henceforth, the main objective of this paper is to automate the transformation rules defined by the first two phases of STREAM process1. To achieve this goal it is necessary to:

- describe the transformation rules using a suitable transformation language;
- make the vertical and horizontal transformation rules compatible with the modeling environment used to create the goal-oriented requirements models, i.e. the iStarTool [6];
- make the vertical transformation rules compatible with the modeling environment used to create the structural architectural models, i.e. the AcmeStudio [4].

In order to automate the HTRs and VTRs proposed by the STREAM process, it was necessary to choose a language that would properly describe the transformation rules and transform the models used in STREAM approach. We opted for the QVTO (Query / View / Transformation Operational) language [2], a transformation language that is integrated with Eclipse environment [16] and that is better supported and maintained.

Note that as input of the first activity of the STREAM process is based on an i* goal model. The iStarTool [6] is used to generate XMI file of the goal-oriented requirements model. This file is read by the Eclipse QVTO plugin, which generates the XMI file of the Acme architectural model. Note that this file is consistent with the metamodel created in accordance with the AcmeStudio tool.

The rest of the paper is organized as follows. Section II presents the theoretical background. Section III describes the horizontal transformations rules in QVTO. In Section IV, we present the vertical transformation rules in QVTO. In order to illustrate our approach, in Section V we use the BTW example [10]. Section VI presents some related works. Finally, Section VII concludes the paper with a brief explanation of the contributions achieved and the proposal of future work.

1 Note that it is out of scope of this paper to support the other two phases of the approach (Select Architectural Solution, Refine Architecture).
II. BACKGROUND

In this section we present the baseline of this research: the original rules from the STREAM approach and the model transformation language (QVT) used to implement HTRs and VTRs of STREAM.

A. STREAM

STREAM is a systematic approach to generate architectural models from requirements models based on model transformation [1]. The source and target modelling languages are i* for requirements modelling and Acme for architectural description, respectively.

The STREAM process consists of the following activities: 1) Prepare requirements models, 2) Generate architectural solutions, 3) Choose an architectural solution and 4) Derive architecture.

Horizontal Transformation Rules (HTRs) are part of the first activity. They are useful to increase the modularity of the i* requirements models. Vertical Transformation Rules (VTRs) are proposed in second activity. They are used to derive architectural models from the modularized i* requirements model. Non-functional requirements (NFRs) are used in the third activity to select one of the possible architectural descriptions obtained. Depending on the NFR to be satisfied, some architectural patterns can be applied, in activity 4.

The first STREAM activity is concerned with improving the modularity of the expanded system actor. It allows delegation of different parts of a problem to different software actors (instead of having a unique software actor). In particular, it is sub-divided into three steps: (i) analysis of internal elements (identify which internal elements can be extracted from the original software actor and relocated to a new software actor); (ii) application of horizontal transformation rules (the actual extraction and relocation of the identified internal elements); and, (iii) evaluation of the i* model (checking if the model needs to be modularized again, i.e., return to the step 1).

In order to develop these steps, it is necessary to use, respectively:

- Heuristics to guide the decomposition of the software's actor;
- A set of rules to transform i* models;
- Metrics for assessing the degree of modularization of both the initial and modularized i* models.

This is a semi-automatic process, since not all the activities can be automated. For example, the step 1 of the first activity cannot be automated because the analyst is the one in charge to choose the sub-graph to be moved to another actor. The Horizontal Transformation Rule 1 (HTR1) moves a sub-graph previously selected. Hence, HTR1 cannot be fully automated because it always depends on the sub-graph chosen by the analyst. Observe that after applying the HTR1, the resulting model may not be in compliance with the i* syntax. So, the next HTRs are to correct possible syntax errors.

The Horizontal Transformation Rule 2 (HTR2) moves a means-end link crossing actor’s boundary. HTR2 considers the situation where the sub-graph moved to another actor has the root element as a “means” in a means-end relationship.

The Horizontal Transformation Rule 3 (HTR3) moves a contribution link crossing the actor’s boundary. HTR3 considers the situation where the sub-graph moved to another actor has a contribution relationship with others elements that were not moved.

The Horizontal Transformation Rule 4 (HTR4) moves a task-decomposition link crossing the actor’s boundary. HTR4 considers the situation where the sub-graph moved has a task-decomposition relationship with other elements that were not moved. Table 1 shows examples of these rules. The graph to be moved in HTR1 is highlighted with a dashed line and labelled with G.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Original Model</th>
<th>Resulting model after applying the rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTR1</td>
<td><img src="image1" alt="Original Model" /></td>
<td><img src="image2" alt="Resulting model" /></td>
</tr>
<tr>
<td>HTR2</td>
<td><img src="image3" alt="Original Model" /></td>
<td><img src="image4" alt="Resulting model" /></td>
</tr>
<tr>
<td>HTR3</td>
<td><img src="image5" alt="Original Model" /></td>
<td><img src="image6" alt="Resulting model" /></td>
</tr>
<tr>
<td>HTR4</td>
<td><img src="image7" alt="Original Model" /></td>
<td><img src="image8" alt="Resulting model" /></td>
</tr>
</tbody>
</table>

The transformation rules are intended to delegate internal elements of the software actor to other (new) software actors. This delegation must ensure that new actors have with the original actor. Thus, the original model and the final model are supposed to be semantically equivalent.

At the end of the first activity, the actors representing the software are easier to understand and maintain, since there is more actors with less internal elements.

In the second STREAM activity (Derive Architectural Solutions), transformation rules are used to transform and i* requirements model into an initial Acme architectural model. In this case, we use the VTRs.
In order to facilitate the understanding, we have separated the vertical transformation rules into four rules. VTR1 maps the i* actors into Acme components. VTR2 maps the i* dependencies into Acme connectors. VTR3 maps a depender actor as a required port of Acme connector. And last but not least, VTR4 maps the dependee actor to a provided port of an Acme connector.

Note the goal of this paper is to fully automate three HTRs (HTR2, HTR3 and HTR4) and all VTRs proposed by the STREAM. HTR1 is not amenable to automation. First, we specify them in QVTO [2]. It is worth noting that to create the i* models, we have relied on the iStarTool tool [6].

B. QVT

The QVT language has a hybrid declarative/imperative nature. The declarative part is divided into a two-tier architecture, which forms the framework for the execution semantics of the imperative part [5]. It has the following layers:

- A user-friendly Relations metamodel and language that supports standard combination of complex object and create the template object.
- A Core metamodel and language defined using minimal extensions to EMOF and OCL.

In addition to the declarative languages (Relations and Core), there are two mechanisms for invoking imperative implementations of Relations or Core transformations: a standard language (Operational Mappings) as well as non-standard implementations (Black-box MOF Operation).

The QVT Operational Mapping language allows both the definition of transformations using a complete imperative approach (i.e. operational transformations) or it lets hybrid approach in which the transformations are complemented with imperatives operations (which implements the relationships).

The operational transformation represents the definition of a unidirectional transformation that is expressed imperatively. This defines a signature indicating the models involved in the transformation and defines an input operation for its implementation (called main). An operational transformation can be instantiated as an entity with properties and operations, such as a class.

III. AUTOMATION OF HORIZONTAL TRANSFORMATION

The first activity of the STREAM process presents some transformation rules that can be defined precisely using the QVT (Query / View / Transformation) transformation language [5], in conjunction with OCL (Object Constraint Language) [9] to represent constraints.

The transformation process requires the definition of transformation rules and metamodels for the source and target languages. The first STREAM activity uses the HTRs, which aim to improve the modularity of the i* models and have the i* language as source and target language.

The rules were defined in QVTO and executed through a plugin for the Eclipse platform. Transformations were specified based on the i* language metamodel considered by the iStarTool. In QVT, it is necessary to establish a reference to the metamodel to be used.

As explained in section II, the steps of the first activity of the STREAM process (Refactor Models Requirements) are: Analysis of internal elements; Application of horizontal transformation rules, and Evaluation of i* models.

The Horizontal Transformation Rules activity takes as input two artefacts: the i* model and the selection of internal elements. The former is the i* system model, and the latter is the choice of elements to be modularized made by Engineer Requirements. The output artefact produced by the activity is refactored and more modularized i* model.

Modularization is performed by a set of horizontal transformation rules. Each rule performs a small and located transformation that produces a new model that decomposes the original model. Both the original and the produced model are described in i*. Thus, the four horizontal transformation rules proposed by [8] are capable of implementation.

First the analyst uses the iStarTool to produce the i* requirements model. Then the HTR1 can be performed manually by him/her also using the iStarTool. The analyst may choose to move the sub-graph for a new actor or an existing actor, and then moves the sub-graph. This delegation must ensure that new actors and the original actor have a relationship of dependency. Thus, the original model and the final model are supposed to be semantically equivalent. Upon completion of HTR1, the artefact generated is used in automated transformations that perform all other HTRs at once. This is the case if the obtained model is syntactically wrong. Table 1 describes the different types of relationship between the components that have been moved to another actor and a member of the actor to which it belonged. If the relationship is a means-end rule, HTR2 should be applied. While if the relationship is a contribution, HTR3 is used. In the situation where tasks decomposition is present, HTR4 is recommended.

In the next section we detail how each of these HTRs was implemented in QVTO.

A. HTR2- Move a means-end link across the actor's boundary

If after applying the HTR1, there is a means-end link crossing the actors’ boundaries, the HTR2 corrects this syntax error since means-end links can exist only inside the actor’s boundary. The means-end link is usually used to connect a task (means) to a goal (end). Thus, the HTR2 make a copy of the task inside the actor who has the goal, in such way that the means-end link is now inside of the actor’s boundary that has the goal (Actor X in Table 1). After that, the rule establishes a dependency from that copied task to the task inside of the new actor (Actor Z in Table 1).

To accomplish this rule, the HTR2 checks if there is at least a means-end link crossing the actors’ boundaries (line 7 of the code present in Table 2). If so, it then checks if this means-end link has as source and target attributes elements present in the boundary of different actors. If this condition holds (line 10), the HTR2 creates a copy of the source element inside the boundary of the actor which possesses the target element of the means-end link (atorDaHora variable.
in line 19). A copy of the same source element is copied outside the actors’ boundaries to become a dependum (line 18). Then, a dependency is created from the element copied inside the actor to the dependum element (line 20) and from the dependum element to the original source element of the means-end link that remained inside the other actor (line 21). The result is the same presented in Table 1 for HTR2. The source code in QVTO for HTR2 is presented in Table 2.

**TABLE II. HTR2 DESCRIBED IN QVTO**

```plaintext
actorResultAmount := oriModel.rootObjects()[Model].actors.name->size();
while (actorResultAmount > 0) {
    if (self.actors- > at(actorResultAmount).type=(ActorType::ACTORBOUNDARY)) then {
        atoresBoundary += self.actors-
        >at(actorResultAmount);}
    var meansend := self.actors-
        >at(actorResultAmount).meansEnd->size();
    var atorDaHora := self.actors-
        >at(actorResultAmount);
    while (meansend > 0) {
        var sourceDaHora := atorDaHora.meansEnd->at(meansend).source;
        var targetDaHora := atorDaHora.meansEnd->at(meansend).target;
        if (sourceDaHora <> targetDaHora) then {
            var atoresBoundarySize := atoresBoundary-
                >size();
            var otherActor : Actor;
            while (atoesBoundarySize > 0) {
                if (atoesBoundary-
                    >at(atoesBoundarySize).name <> atorDaHora.name) then {
                    otherActor := atoresBoundary-
                        >at(atoesBoundarySize);
                } else {
                    otherActor := atorDaHora;
                } endif;
                atoresBoundarySize :=
                    atoresBoundarySize - 1;
            }
        } endif;
        self.elements += object Element{
            name := atorDaHora.meansEnd->at(meansend).source.name;
            type := atorDaHora.meansEnd->at(meansend).source.type;
        };
        atorDaHora.elements += object Element{
            name := atorDaHora.meansEnd->at(meansend).source.name;
            type := atorDaHora.meansEnd->at(meansend).source.type;
        };
        meansend := meansend - 1;
    }
}
actorResultAmount := actorResultAmount - 1;
```

B. HTR3- Move a contribution link across the actor’s boundary

HTR3 copies a softgoal that was moved to its source actor, if this softgoal is a target in a contribution link with some element in his initial actor. The target of the link is moved from the softgoal to its copy in the initial actor. This softgoal is still replicated as a dependum of a dependence link from the original softgoal to its copy.

If an element of some actor has a contribution link with a softgoal that is within the actor that was created or received elements in HTR1, then this softgoal will be copied into the actor that has an element that has a contribution link with this softgoal. The target of the contribution link becomes that copy. This softgoal is also copied as a dependum of a softgoal dependency in its original copy.

In order to accomplish this rule, we analyse if any contribution link has the source and target attributes with elements present in different actors. If the actor element present in the source or the target is different from the actor referenced in attribute of the element, then this element (softgoal) is copied to the actor present in source or target that has the different name of the actor analysed. The target attribute of the contribution link shall refer to this copy. This same softgoal is also copied to the modelling stage and creates a dependency from the softgoal copy to original softgoal with to the copied softgoal to the stage as dependum. The target of this dependence is the copy and the source is the original softgoal.

C. HTR4- Move a task-decomposition link across the actor’s boundary

HTR4 replicates an element that is the decomposition of a task into this other actor as dependum a dependency link between this element and the task, and removes the decomposition link.
If an some actor's element is task decomposition within the actor that was created or received elements in HTR1, then that decomposition link is removed, and a copy of this element will be created and placed during the modelling stage as a dependum of a dependency between the task in the actor created or received elements in HTR1 and the element present in another actor that was the decomposition of this task. The target of this dependence is the element and the source is the task.

In order to accomplish this rule, we analyse if any decomposition task link has source and target attributes with elements present in different actors. If the actor of the element present in the source or target is different from the referenced actor in the moves attribute of element, then that element is copied during the modelling stage to create a dependency from the referenced element as the source of decomposition link to the element referenced as the target, i.e., a dependency from the original element of the task, with the copied element to the stage as a dependum. The target of this dependence is the task and the source is the element. The decomposition link is removed.

IV. AUTOMATION OF VERTICAL TRANSFORMATIONS

The second STREAM activity (Generate Architectural Solutions) uses transformation rules to map i* requirements models into an initial architecture in Acme. As these transformations have different source and target languages, they are called vertical transformations.

In order to facilitate the understanding of de VTRs as well as the description of them, we separate the vertical transformation rules in four rules [14].

Below we detail how each of these VTRs was implemented.

A. VTR1- Mapping the i* actors in Acme components

In order to describe this first VTR it is necessary to obtain the quantity of actors present in the artefact developed in the first activity. From this, we create the same quantity of Acme components (line 3 of code present in Figure 1), giving the same actors name. The Figure 1 shows an excerpt of QVTO code for VTR1.

```qvto
1 while (actorsAmount > 0) {
2   result.acmeElements += object Component{
3     name := self.actors.name->at(actorsAmount);
4     actorsAmount := actorsAmount - 1;
5   }
} 
```

Figure 1. Excerpt of the QVTO code for VTR1

The XMI output file will contain the Acme components within the system represented by the acmeElements tag (line 1 of code present in Figure 2), an attribute of that tag, xsi:type (line 1), that contain information that is a component element and the attribute name the element name, as depicted in Figure 2. Figure 3 shows graphically a component.

```xmi
1 <acmeElements xsi:type="Acme:Component" name="Advice Receiver"> ...
2 </acmeElements>
```

Figure 2. XMI tag of component

However, an Acme component has other attributes, not just the name, so it is also necessary to perform the VTR3 and VTR4 rules to obtain the other necessary component attributes.

B. VTR2- Mapping the i* dependencies in Acme connectors

Each i* dependency creates two XMI tags. One captures the link from depender to the dependum and the other defines the link from the dependum to the dependee.

```qvto
1 while (dependencyAmount > 0) {
2   if (self.links.source- > includes(self.links.target->at(dependencyAmount))
3     and self.actors.name->excludes(self.links.target->at(dependencyAmount).name)) then {
4       result.acmeElements += object Connector{
5         name := self.links.target.name->at(dependencyAmount);
6         roles += object Role{
7           name := "dependeeRole";
8           };
9         roles += object Role{
10          name := "dependeeRole";
11          };
12       } dependencyAmount := dependencyAmount - 1;
13   } 
```n
Figure 4. Excerpt of the QVTO code for VTR2

As seen in Figure 4, for the second vertical rule, which transforms i* dependencies to Acme connectors (line 3 of code present in Figure 4), each i* dependency creates two tags in XMI, one captures the connection from the depender to the dependum (line 5) and another defines the connection from the dependee to the dependum (line 7).

In order to map these dependencies into Acme connectors it is necessary to recover the two dependencies tags, observing that the have the same dependum, i.e., the target of a tag must be equal to the source of another tag, which can characterize the dependum. However, they should not consider the actor which plays the role of depender (source) in some dependency and dependee (target) in another. Once this is performed, there are only dependums (intentional elements) left. For each dependum, one Acme connector is created (line 1 of code present in Figure 5).

The connector created receives the name of the intentional element that represents the dependum of the dependency link. Two roles are created within the connector, one named dependeeRole and another named dependeeRole.

The XMI output file will contain the connectors represented by tags (see Figure 5).
C. VTR3- Mapping depender actors as required port of Acme connector

With the VTR3, we map all depender actors (source from some dependency) into a required port of an Acme connector. Thus, we list all model’s actors that are source in some dependency (line 2 of code present in Figure 6). Furthermore, we create an Acme port for each depender actor (line 3). Each port created has a name and a property (lines 4 and 5), the name is assigned randomly, just to help to control them. The property must have a name and a value, the property name is “Required” once we are creating the required port, as figured in Figure 6.

The XMI output file will contain within the component tag (line 1 of code present in Figure 7) the tags of the ports. Inside the port’s tag there will be a property tag with the name attribute assigned as “Provided” (line 3 of code present in Figure 9). While the value attribute is set to “true” and the type attribute as boolean.

D. VTR4- Mapping dependee actors as provided port of Acme connector

VTR4 is similar to VTR3. We map all dependee actors (target from some dependency) as a provided Acme port of a connector. Thus, we list all model’s actors that are target in some dependency (line 2 of code present in Figure 8). We create an Acme port for each dependee actor. It has a name and property, the name is assigned randomly, simply to control them (line 4). The property must have a name and a value. The property name is set to “Provided” once we are creating the provided port. Figure 8 presents an QVTO excerpt code for the provided port.

The XMI output file will contain within the component a tag to capture the ports. Inside the port’s tag that are provided there will be a property tag with the name attribute assigned as “Provided” (line 3 of code present in Figure 9). The value attribute is set to “true” and the type attribute as boolean.

V. RUNNING EXAMPLE

BTW (By The Way) [10] is a route planning system that helps users with advice on a specific route searched by them. The information is posted by other users and can be filtered for the user to provide only the relevant information about the place he wants to visit. BTW was an awarded projected presented at the ICSE SCORE competition held in 2009 [11].

In order to apply the automated rules in i* models of this example, it necessary to perform the following steps:
1. Create the i* requirements model using the iStarTool;
2. Use the three heuristics defined by STREAM to guide the selection of the sub-graph to be moved from an actor to another;
3. Manually apply the HTR1, but with support of the iStarTool. The result is an i* model with syntax errors that must be corrected using the automated transformation rules;
4. Apply the automated HTR2, HTR3 and HTR4 rules.
After step 1, we identified some elements inside the BTW software actor that are not entirely related to the application domain of the software and these elements can be placed on other (new) software actors. In fact, the sub-graphs that contain the "Map to be Handled", "User Access be Controlled", and "Information be published in map" elements can be viewed as independents of the application domain. To facilitate the reuse of the first sub-graph, it will be moved to a new actor named "Handler Mapping". The second sub-graph will be moved to a new actor named "User Access Controller" while the third sub-graph will be moved to a new actor called Information Publisher.

Steps 1 and 2 are performed using the iStarTool. This tool generates two types of files (extensions): the file "istar_diagram" has information related to the i* requirements model; the "istar" file has information related to the i* modelling language metamodel. Since the file "istar" is a XMI file, we changed its type (extension) to "xmi". XMI files are used as input and output files by the automated rules (HTR2, HTR3 and HTR4). The BTW i* model and the elements to be moved to other actors are shown in Figure 10. Figure 11 depicts the BTW i* model after applying HTR1. Note that there are some task-decomposition and contribution links crossing the actors’ boundaries, meaning that the model is syntactically incorrect and must be corrected by the automated HTRs.

In order to apply the HTR2, HTR3 and HTR4, we only need to execute a single QVTO file. Thus, with the eclipse configured to QVT transformations, along with the metamodel of i* language referenced in the project and the input files referenced in the run settings, the automated rules will be applied simultaneously by executing the QVTO project.

After applying the HTRs, a syntactically correct i* model is produced. In this model, the actors are expanded, but in order to apply the vertical transformation rules, it is necessary to contract all the actors (as shown in Figure 12) to be used as input in the second STREAM activity (Generate Architectural Solutions).

Moreover, when applying the VTRs, we only need to execute a single QVTO file. The VTRs are executed sequentially and the analyst will visualize just the result model [15].

Figure 13 presents the graphical representation of the XMI model generated after the application of the VTRs. This XMI is compatible with the Acme metamodel.
Figure 11. BTW i* model after performing HTR1

Figure 12. BTW i* model after applying all HTRs
VI. RELATED WORKS

Coelho et al. proposes an approach to relate aspect oriented goal models (described in PL-AOV-Graph) and architectural models (described in the PL-AspectualACME) [12]. It defines the mapping process between these models and a set of transformations rules between their elements. The MaRiPLA (Mapping Requirements to Product Line Architecture) tool automates this transformation, which is implemented using the Atlas Transformation Language (ATL) transformation language.

Medeiros et al. presents a MaRiSA-MDD, a strategy based on models that integrate aspect-oriented requirements, architectural and detailed design, using the languages AOV-graph, AspectualACME and aSideML, respectively [13]. MARISA-MDD defines, for each activity, models (and metamodels) and a number of model transformations rules. These transformations were specified and implemented in ATL. However, none of these works relied on i*, our source language, which has much larger community of adopters than AOV Graph.

VII. CONCLUSION

This paper presented the automation of most of the transformation rules that support the first and second STREAM activities, namely Refactor Requirements Models and Derive Architectural Solutions [1].

In order to decrease the time and effort required to perform these STREAM activities, as well as to minimize the errors introduced by the manual execution of the rules, we proposed the use of the QVTO language to automatize the execution of seven out of the eight STREAM transformation rules.

The input and output models of the Refactor Requirements Models activity are compatible with the iStarTool. While the ones generated by the Derive Architectural Solutions activity are compatible with the AcmeStudio tool.

The iStarTool was used to create the i* model and to perform the HTR1 manually. The result is the input file to be processed by the automated transformation rules (HTR2, HTR3 and HTR4). Both the input and output files handled by the transformation process are XMI files.

The STREAM transformation rules were defined in QVTO and an Eclipse based tool support was provided to enable their execution. In order to illustrate the use of the automated transformation rules the automated rules were used in the BTW example [10].

The output of the execution of the VTRs is a XMI file with the initial Acme architectural model. Currently, the AcmeStudio tool is only capable of reading XMI files, since it was designed to only process files described using the Acme textual language. As a consequence, the XMI file produced by the VTRs currently cannot be graphically displayed. Hence, we still need to define new transformation
rules to generate a description in Acme textual language from the XMI file already produced.

Moreover, more case studies are still required to assess the benefits and identify the limitations of our approach. For example we plan to run an experiment to compare the time necessary to perform the first two STREAM activities automatically against an ad-hoc way.

VIII. ACKNOWLEDGEMENTS

This work has been supported by CNPq, CAPES and FACEPE.

REFERENCES


