Towards Translational Execution of Action Language for Foundational UML

Federico Ciccozzi, Antonio Cicchetti, Mikael Sjödin
Mälardalen University (MRTC) - Västerås, Sweden
email: [federico.ciccozzi, antonio.cicchetti, mikael.sjodin]@mdh.se

***Abstract*—Model-driven engineering has prominently gained consideration as effective substitute of error-prone code-centric development approaches especially for its capability of abstracting the problem through models and then manipulating them to automatically generate target code. Nowadays, thanks to powerful modelling languages, a system can be designed by means of well-specified models that capture both structural as well as behavioural aspects. From them, target implementation is meant to be automatically generated. An example of well-established general purpose modelling language is the UML, recently enhanced with the introduction of an action language denominated ALF, both proposed by the OMG.**

In this work we focus on enabling the execution of models defined in UML–ALF and more specifically on the translational execution of ALF towards non-UML target platforms.

***Keywords*-model-driven engineering; action language; translational execution; code generation; UML; ALF**

I. Introduction

Model-Driven Engineering (MDE) proposes to face the increasing complexity of modern software systems by shifting the focus of the development from coding to modelling. Models abstract the reality by providing only those details that matter for the particular problem taken into account and are built by following a set of rules prescribed by means of a language definition, referred to as *metamodel* [1]. A core objective of MDE is to provide automated generation of code from design models, in order to tackle error-proneness typical of code-centric approaches. On the one hand, automating the code generation is a sensitive task, since its unreliability would void most of the benefits of adopting MDE, strengthening the predilection for manual approaches.

On the other hand, effective code generation can positively impact economic factors, such as time-to-market as well as overall costs and risks. This can be achieved thanks to, e.g., the ability of abstracting from details not needed at design level and that are typical of code-centric approaches. Qualitative factors are also improved, such as overall quality and maintainability of the generated code and consistency between the different artefacts along the entire development process (i.e., models at different abstraction levels and generated code) [1]. Being able to produce 100% of target code from design models enforces consistency and thereby results from model-based analysis are likely to be valid also at runtime (and the other way around). Moreover, generated code aims to achieve higher and more consistent quality with respect to errors, maintainability and readability.

Nevertheless, in order to achieve generation of 100% code, design models should provide proper means to specify complex executable behaviours. In some cases, this is achieved by means of target languages (e.g., Java) embedded somehow in the modelling language. In this way, consistency at modelling level can be hard to be achieved due to the abstraction gap between modelling and programming languages that hinders native code from being aware of the concepts defined in the models. A preferrable way is instead to define model-aware action languages. This is the case of the Action Language for Foundational UML (ALF) [2] defined by the Object Management Group (OMG) to act as the surface notation for specifying executable behaviors within a wider model that is primarily represented using the usual graphical notations of UML [2].

The contribution of this research work consists into the definition of an automatic mechanism for the translation of ALF into executable non-UML code to be executed on a non-UML target platform (namely translational execution – see Section II). This work contributes to the generation of full-fledged functional code from UML–ALF models in the context of the round-trip support for preservation of extra-functional properties already proposed in [3].

The remainder of the paper is structured as follows. In Section II we provide the identification of the problem's context while leaving the details about the related state-of-the-art to Section III. The addressed problem is formally defined in terms of contribution and assumptions in Section IV. The description of the proposed solution is presented in Section V. The validation of the approach against industrial case-studies is given in Section VI together with a discussion on the proposed contribution. The paper is concluded by Section VII with a summary of the presented work together with possible future enhancements.

II. Context

According to the MDE paradigm, a system is developed by designing models and refining them starting from higher and moving to lower levels of abstraction until code is generated; refinements are performed through transformations between models. A *model transformation* translates a source model into a target model (or text) while preserving their well-formedness [4]. Since a model is an abstraction of the system under development, rules and constraints for building it have to be properly described through a corresponding language

definition. In this respect, a metamodel describes the set of available concepts and well-formedness rules a correct model must conform to [5].

In the scope of MDE applied to industrial development, the UML by OMG has been widely adopted and many tool vendors produce support tools, some of which reaching a notable degree of industrial acceptance. Moreover, the OMG has strengthened the versatility and power of UML by equipping it with the possibility to express textual surface representation of its elements by means of the ALF. One of the primary goals of ALF is to act as the surface notation for specifying executable behaviours [2]. Being able to define models in a detailed manner both concerning structural and behavioural aspects allows to automatise the translation of them towards diverse executable formats, even targeting non-UML platforms.

As prescribed in its specification [2], the execution semantics for ALF is specified by a formal mapping to foundational UML (fUML), which is a UML's subset defining a basic virtual machine for UML, the abstractions supported by it, and thereby enabling conforming models to be translated into diverse executable forms for different purposes, such as verification, integration, and deployment [6]. There are three prescribed ways in which ALF execution semantics may be implemented [2], summarised as follows:

- **Interpretive Execution:** the ALF code is directly interpreted and executed;
- **Compilative Execution:** the ALF code is translated into a UML model conforming to the fUML subset and executes it according to the semantics specified in the fUML specification;
- **Translational Execution:** the ALF code, as well as any surrounding UML concept in the model, is translated into some executable form on a non-UML target platform, and executes on it.

In this work, we aim at providing a solution towards the translation execution of ALF, using the UML–ALF implementation and related facilities (e.g., editors, parsers, metamodels) provided along with Papyrus [7], an open-source integrated environment for editing Eclipse Modeling Framework (EMF) [8] models and particularly supporting UML and related profiles such as SysML and MARTE, on the Eclipse platform.

III. Related Work

Overall, several different approaches can be found in the literature regarding translation of design models to non-modelling platform target for execution purposes (i.e. code generation). In [9] the authors propose a code generation solution to produce C tailored for real-time embedded systems from AADL focusing on flexibility of the code generator. This supports the reasoning about the multi-step approach proposed in our solution thought to be highly flexible and adaptable to different target platform languages. The usefulness of introducing intermediate representations (i.e., intermediate (meta)model in our solution) for increased abstraction, cleaner separation between the front and back ends, and introduction of possibilities for re-targeting as well as cross-generation of code has already been proposed in the late 80's [10]. In our solution we prefer to place intermediate artefacts at the same abstraction level as the design models in order to maintain domain-independence and enhance reusability. Nevertheless, similar results could have been achieved through direct translation from UML–ALF to the targeted language, since these modelling formalisms already conceive common object-oriented concepts, but to the detriment of the properties mentioned above.

Several works, such as [11]–[13], just to mention a few, provide solutions similar to ours from an abstract perspective (i.e., using UML profiles and state-machine diagrams as source artefacts), though not focusing on generating full-fledged code. Other works, such as [14], [15], generate code exploiting XML-based formalisms and scripts as well as specifying behaviours by means of target languages (e.g., Java) instead of model-aware formalisms such as ALF. In this way, consistency at modelling level may be jeopardised since the abstraction gap between modelling and programming languages does not permit native code from being aware of modelling concepts.

Finally, at the best of our knowledge, no documented attempt can be found in the literature concerning the definition and implementation of transformation mechanisms towards the translational execution of ALF, which represents the main contribution of this work.

IV. Problem Formalisation

In this section the problem is formalised in terms of intended contribution and assumptions made to bound the scope of problem and solution.

A. Contribution

The overall contribution of this research work consists into the definition of an automatic mechanism for the translational execution of ALF, meant as the translation of the ALF text, as well as surrounding UML concepts, into a non-UML target language to be executed on a non-UML target platform. Specific contributions of this paper are:

- Definition and implementation of a target-agnostic intermediate metamodel resembling common object-oriented programming languages (such as Java, C++) to which ALF concepts are translated to. The introduction of an intermediate metamodel can be beneficial for several reasons, namely, increased abstraction, cleaner separation between the front and back ends, and introduction of possibilities for re-targeting as well as cross-generation of code. Moreover, intermediate representations may also help in supporting advanced code optimizations through ad-hoc manipulations of such artefacts achieved through apposite model transformations, which are independent of the code generating transformations. Doing so, the translation process is broken down into smaller steps and thereby the validation of the translation mechanisms is simplified and the generation of different target languages would only require the modification the target-specific

translation mechanisms (i.e., model-to-text transformations from intermediate concepts to target language);
- Definition and implementation of model transformations (both model-to-model and model-to-text) to actually perform the translation from ALF concepts to a non-UML target language (i.e., C++).

The translation of surrounding UML concepts is considered as black-box in this work since the focus is on novel mechanisms for the translation of ALF.

B. Assumptions

There are three levels of syntactical conformance defined for ALF, namely *minimum*, *full*, and *extended*. In this work we focus on the minimum conformance and we provide translation of most of the entailed concepts. The set of translated concepts, although limited if considering the expressiveness provided by ALF, reflect the ones which are usually found and used in the target language (and target domain). This allowed us to generated 100% of target code for an industrial case-study in the telecommunications domain.

Moreover, regarding the surrounding UML concepts in the model needed to be translated for allowing execution of the generated implementation, we considered UML component and composite component diagrams for the specification of the structural aspects of the system under development, and state-machine diagrams together with ALF code for the behavioural description. Additionally, we delimited the number of state-machines to one per non-composite component and we define ALF code at component operation level. Behaviour of state-machine transitions is defined within the component operation triggering the specific transition[1].

V. Proposed Solution

In this section we describe the proposed solution focusing on: (A) involved intermediate modelling artefacts, (B) overview of the transformation process, (C-D) detailed description of the transformation steps and (E) application of the solution to a sample operation.

A. Intermediate Metamodel

The intermediate metamodel has been defined by means of Ecore in EMF. The employment of intermediate models eases adaptability of the transformation process to target programming languages different from C++. This is possible since the intermediate model represents a generic object-oriented (OO) abstraction of the system which is agnostic of the targeted programming language.

The intermediate metamodel (ALF-related portion) is depicted in Figure 1 where we can distinguish two main portions used to model the followings:

- **Statements**: identified by the abstract `Statement` metaclass and the specialising metaclasses that represent the different types of statement;
- **Expressions**: identified by the abstract `Expression` metaclass and the specialising metaclasses that represent the different expression kinds.

According to this decomposition of the intermediate metamodel, in the next paragraphs an overview on the main metaclasses related to the translation of ALF concepts (i.e., Statements and Expressions) is given and depicted in Figure 1.

Statements

In our UML models, the body of the operations is specified by means of ALF code in terms of a sequentially ordered set of statements. This portion of the intermediate metamodel is devoted to the definition of the concepts needed to model statements. The main metaclasses are:

- `Statement`: is the core abstract metaclass;
- `SimpleInvocation`: represents the invocation of a function and it contains an `Invocation` expression (see next paragraph);
- `Assign`: represents the assignment of a value to a variable. It contains a specific `InstanceAccess` expression (see next paragraph);
- `Control`: is the abstract metaclass representing statements related to the control flow of the application. It is specialised by specific metaclasses such as: `If`, for modelling *if* loops and *switch* cases, `ForRange`, `ForEver` and `ForEach`, for modelling the various different *for* loops available in ALF, `While`, for modelling *while* loops, and `Return`, for modelling the exit from an operation body. If the operation has a defined return-type, the statement includes an expression evaluating it;
- `InjectionStatement` represents the statement that can be introduced in the code through injection markers;
- `Inline`: it is used to place inline statically defined statements in the generated code. It should be employed mainly for debugging and analysis purposes (e.g., check the correctness of the control flow when testing the application) and hence be avoided when generating the final implementation version.

Expressions

Operations' body can include expressions. This portion of the metamodel defines the concepts needed to translate expressions:

- `Expression`: is the core abstract metaclass;
- `Invocation`: represents the expression related to the invocation of a function and defines input parameters and expected return type;
- `RTTI`: represents an abstract metaclass that is specialised by metaclasses, such as `InstanceOf` and `Hastype`, defining type introspection provided by ALF;
- `Alloc`: is a metaclass used to define a memory allocation in case the target programming languages would require it;
- `Define`: represents the specification of symbolic constant declarations (i.e., `#DEFINE` or similar, depending on the target programming language);
- `InstanceAccess`: is an abstract metaclass for the

[1]This is due to some limitations of the ALF editor provided in Papyrus that was still under development at the time we started our implementation.

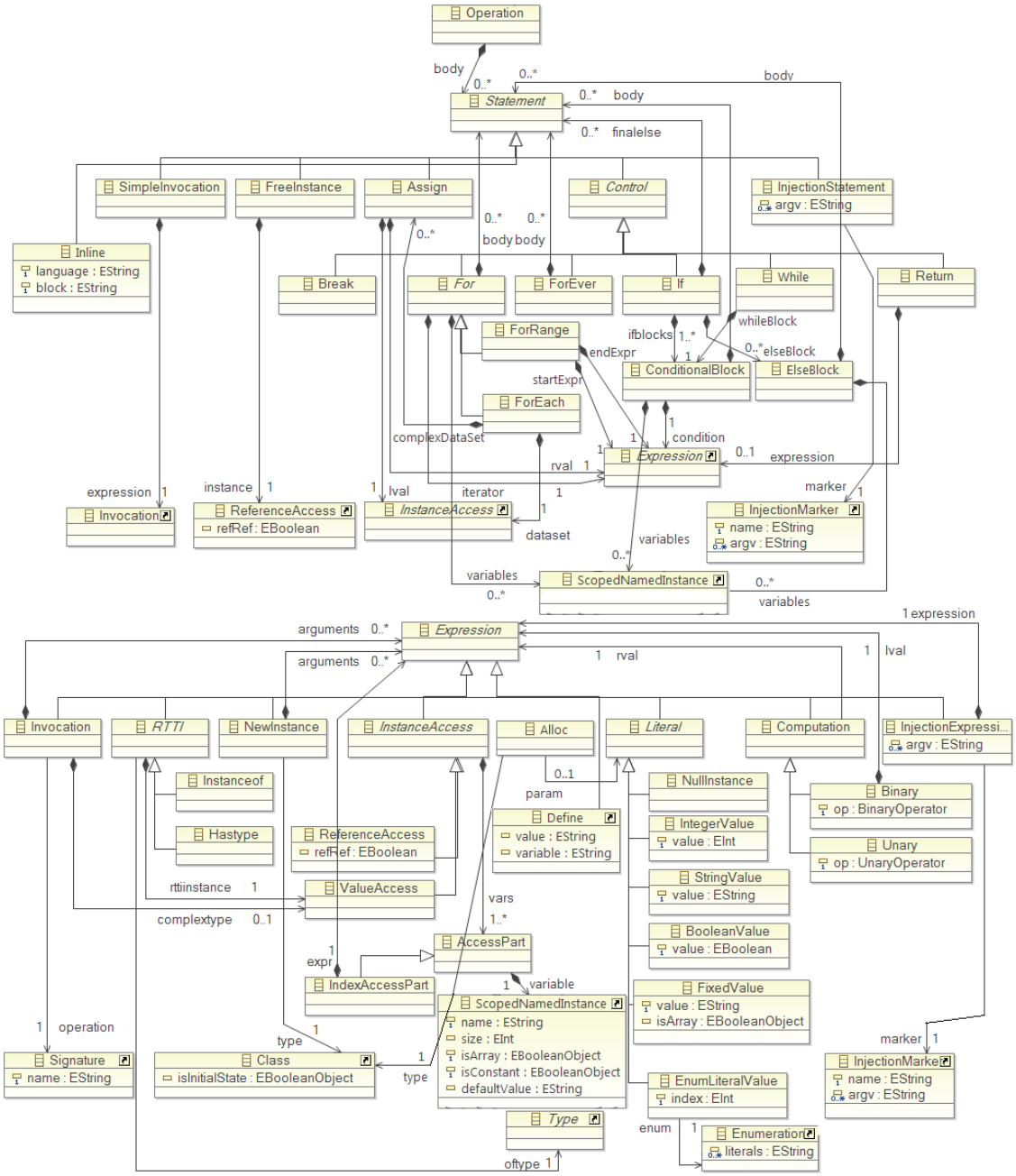


Fig. 1. Portion of the Intermediate Metamodel

definition of access to variables;

- `ReferenceAccess`: represents the access to a variable's reference;
- `ValueAccess`: represents the access to the value of a variable;
- `Literal`: is an abstract metaclass representing the different constant literal values. It is specialised by other metaclasses, such as `IntegerValue`, `StringValue`, `BooleanValue`, that define the specific literal type;
- `Computation`: represents an abstract metaclass which is specialised by `Unary` and `Binary` respectively defining an expression that executes a unary operator and an expression that executes a binary operator.

The intermediate metamodel provides the means for hosting a wide range of concepts that are then interpreted in a certain way by the model-to-text transformation specific for the selected target programming language. That is to say that, while the syntax is fixed, the semantics that the various metaconcepts assume might change from one target programming language to another and is therefore embedded in the model-to-text transformation. In the next sections, the transformations from ALF to C++ passing through the intermediate model are

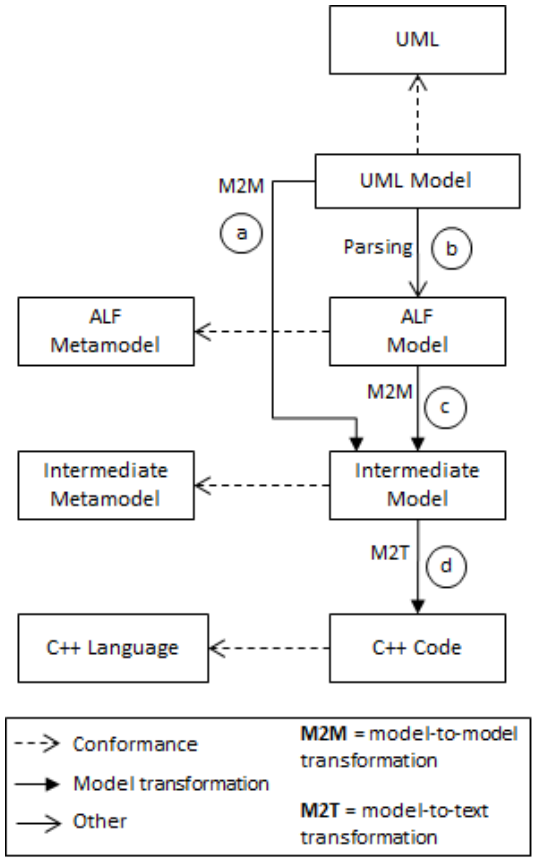


Fig. 2. Overview of the Transformation Process

described in detail. Due to the complexity and verbosity of the transformations (around 4000 code lines divided into 180 transformation rules), only a set of core rules are reported, by means of pseudo-code, in order to permit reasoning about the overall transformation approach. Within the algorithms describing the transformations the following abbreviations will be employed:

- UML: UML metamodel
- UmlM: UML model
- AlfMM: ALF metamodel
- AlfOpM: ALF operation model
- InterMM: intermediate metamodel
- InterM: intermediate model

B. Transformation Process

The proposed solution is depicted in Figure 2 and is constituted by a set of model transformations. Starting from the UML model of the system under development, we start by translating the structural definition from component, composite component and state-machine diagrams into appropriate intermediate concepts through a model-to-model transformation (Figure 2.a). Regarding the translation of state-machines, our approach resembles the *state design pattern*, as defined in [16], considering the component owning the state-machine as the *context* for the related states. At this point, the behavioural definition specified by means of ALF code for components' operations has to be translated into intermediate concepts to complete the process (Figure 2.c). This is achieved through an in-place model-to-model transformation, whose details are given in the following sections. Note that, thanks to a dedicated parser provided by Papyrus, the action code related to each operation can be retrieved (Figure 2.b) and manipulated as a model, which would be conforming to the ALF operation metamodel (in turn part of the ALF metamodel [2]; for simplicity reasons we will consider the ALF operation models as conforming to the ALF metamodel leaving apart the ALF operation metamodel). The last step of the process entails the transformation of the intermediate model into a specific target language, in our case C++. This is achieved through a model-to-text transformation (Figure 2.d). We will now focus on the translation of ALF to intermediate concepts and thereby to C++.

C. Translating ALF to Intermediate

Let us consider the structural specification of the system defined by means of UML concepts (i.e., components, ports, states, transitions) as already translated. The intermediate model would then need to be completed by the behavioural description which is defined in terms of ALF action code within components' operations. An in-place model-to-model transformation defined using Operational QVT (QVTo) [17] takes as input an ALF operation model and translates its elements into their counterpart in the intermediate representation. The overall transformation workflow is summarised by means of pseudo-code in Algorithm 1.

The transformation takes as input AlfOpM, UmlM and InterM and as output it provides an enriched version of InterM. For each of the parsed ALF operations present in UmlM, the related operation body is navigated and for each of the found ALF statements the appropriate handler function is called in order to translate it into intermediate concepts. Each of these handlers employs in turn further helpers and queries whose size varies from few to several hundreds of lines of code (e.g., 280 lines is the size of the transformation rule translating boolean expressions). An example partially representing the translation of the `if` statement is depicted in Section V-E.

The transformation is able to translate most of the concepts defined within the minimum conformance in the ALF specification [2]. The concepts currently left out of the translation process, most of which not commonly used in the target language (and domain), are: Behavior Invocation Expressions, Feature Invocation Expressions, Super Invocation Expressions, Link Operation Expressions, Class Extent Expressions, Sequence Operation Expressions, Sequence Reduction Expressions, Sequence Expansion Extensions, Isolation Expressions, Classification Expressions, Conditional-Test Expressions, Annotated Statements, Empty Statements, accept Statements, and classify Statements. It is important to notice that the translation of ALF concepts is independent of the underlying UML, and that makes the related transformation process reusable in other development processes which are, e.g., based on UML profiles.

Algorithm 1 M2M Transformation from ALF operation model to intermediate model

```
Alf2Intermediate(in UmlM, in AlfOpM, out InterM){
for each AlfOperation op in UmlM do
  transformAlfBlock(op.body){
  for each statement st in op.body do
    switch (st.type)
    case Invocation/Assignment/Declaration:
      handleInvAssDecl(st);
    case IfStatement:
      handleIf(st);
    case ForStatement:
      handleFor(st);
    case ReturnStatement:
      handleReturn(st);
    case SwitchStatement:
      handleSwitch(st);
    case ThisInvocationStatement:
      handleThis(st);
    case InlineStatement:
      handleInline(st);
    case WhileStatement:
      handleWhile(st);
    case AnnotatedStatement:
      handleAnnotation(st);
    case EmptyStatement:
      handleEmpty(st);
    default:
      nop;
    end switch

  end for
end for
}
```

D. Translating Intermediate to C++

Once the intermediate model is complete, the final step of generating C++ code can be carried out. A model-to-text transformation defined by using the Xpand [18] language is in charge of generating the actual C++ taking as input the intermediate model. The transformation takes as input InterM and produces in output a header (.h) and an implementation (.cpp) file, while configuration and `make` files are statically defined since they do not depend on the concepts carried by the intermediate model.

Overall, the transformation is composed of the following five template files containing the actual transformation rules:

- **Expressions**: definition of the transformation rules translating the intermediate concepts concerning expressions (i.e., `Expression` and specialising elements in Figure 1) to C++;
- **Statements**: definition of the rules that take care of transforming the intermediate concepts concerning statements (i.e., `Statement` and specialising elements in Figure 1) to C++;
- **Declarations**: definition of the transformation rules that transform the intermediate concepts (e.g., variables, methods, classes) into C++ forward declarations;
- **Implementations**: definition of the transformation rules that generate the implementation of the methods defined in the intermediate model;
- **Main**: representation of the core template that, exploiting the other ones, generates a C++ header and a C++ implementation file.

Additionally, functional extensions have been defined in terms of the Xtend [18] language in order to lighten the verbosity of transformation rules and increase their readability and understandability. Moreover, by exploiting the notion of polymorphic template invocation, we were able to considerably contain the size of the transformation both in number of rules and lines of code.

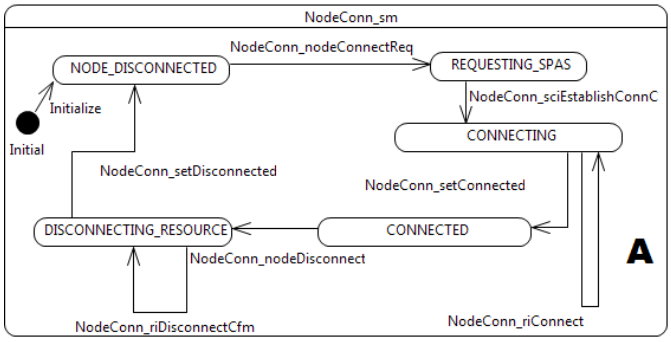
Thanks to the model-to-model transformation process that manipulates the UML–ALF concepts to get an intermediate representation, the model-to-text transformation task can be considered as a fairly straightforward translation of modelling concepts into C++. Moreover, the generality of the concepts defined in the intermediate model makes it possible to easily implement other transformations for generating code targeting different languages (e.g., Java, C#).

E. Applying the Solution

In order to show an example of input and output of the transformation process we consider the state-machine depicted in Figure 3.A, which represents the behaviour of the `NodeConn` component (part of an industrial case-study, see Section VI) and for which we focus on the ALF code specifying the behaviour of the operation `NodeConn_riDisconnectCfm` (Figure 3.B). The C++ code generated for the `NodeConn_riDisconnectCfm` operation is shown in Figure 3.C.

In order to better understand the transformation process, let us first focus on the step between ALF and intermediate concepts (refer to Figure 1). First, an `Operation` is created with `Signature` set as *NodeConn_riDisconnectCfm* and owning passed parameters *portId* and *serverConn_r* defined as `ScopedNamedInstance`. Then, an IfStatement is encountered and handled. A control statement `If` is created as well as a `ConditionalBlock` and an `ElseBlock`. `ConditionalBlock` will contain a `Binary` expression as condition (representing the if condition) and two `Assign` statements. The `Binary` expression is of type *AND* (i.e., logic AND) and combines two furter `Binary` expressions of type *EQ* (i.e., equal to). Moreover, value accesses are used to represent the single variables within them. `ElseBlock` will contain the body of the else (empty for the two outer if statements in Figure 3). Within the IfStatement two more nested IfStatement are encountered and properly handled. The transformation rules translating an IfStatement are depicted in Fig. 4.

To grasp how value accesses, assignments and invocations work, let us consider the two `Assign` statements previously mentioned. The first will be composed of a `ValueAccess`, in turn composed of an `IndexAccessPart` (for 'connHalf[serverConn_r.connHalf]') and an `AccessPart` (for '.respondState') as left hand side, as well as a




```
public NodeConn_riDisconnectCfm ( in portId : model::modelComponentView::DataTypes::Cello_Port,
in serverConn_r : model::modelComponentView::DataTypes::Cello_RiServerConn)
{
/*@inline('C++')
   "ENTER( \"NodeConn_riDisconnectCfm. \" );"
*/

if(serverConn_r.serverConnId == serverConnId && connHalf[serverConn_r.connHalf].portId == portId)
{
   connHalf[serverConn_r.connHalf].respondState =
       model::modelComponentView::DataTypes::NCCdataTypes::RespondState[RI_RESPONDED_CFM];

   model::modelComponentView::DataTypes::U8 secondHalf =
                                                NodeConn_getSecondHalf(serverConn_r.connHalf);
   if(connHalf[secondHalf].respondState !=
  model::modelComponentView::DataTypes::NCCdataTypes::RespondState[WATING_FOR_RESPONSE])
   {
      model::modelComponentView::DataTypes::NCCdataTypes::SciState x =
                                              SciHandler_RI.SciHandler_getSciConnectionState();
      if(x == model::modelComponentView::DataTypes::NCCdataTypes::SciState[SCI_ATTACHED])
      {
         NodeConn_setDisconnectingSpasState();
         SciHandler_RI.SciHandler_sciReleaseConnReq(spasConnId, serverConnId);
         // Simulate  NodeConn_sciReleaseConnCfm();
      }
      else
      {
         NodeConn_setDisconnectedState();
         NetConn_RI.NetConn_nodeDisconnectCfm();
         NodeConn_removeData();
      }
   }
   else
   {
      //wait for second response
   }
}
else
{
   //illegal parameters received
}
}
```

B

```
void NodeConn_ci_StateMachine::NodeConn_riDisconnectCfm(Cello_Port portId,
                                          Cello_RiServerConn serverConn_r) {
/* local variables: */
/* statements: */
//start inline statements (language: 'C++')
ENTER( "NodeConn_riDisconnectCfm. " );
//end inline statements

if(((serverConn_r.serverConnId == serverConnId) &&
                    (connHalf[serverConn_r.connHalf].portId == portId))){
U8 secondHalf;
connHalf[serverConn_r.connHalf].respondState = RI_RESPONDED_CFM;
secondHalf = NodeConn_getSecondHalf(serverConn_r.connHalf);

if(connHalf[secondHalf].respondState){
SciState x;
x = ((SciHandler_ci_StateMachine*)stateMachines[port_SciHandler_RI])->
                                          SciHandler_getSciConnectionState();

if((x == SCI_ATTACHED)){
NodeConn_setDisconnectingSpasState();
sendInternal_I_SciHandler_NodeConn_SciHandler_sciReleaseConnReq(cid,
                           port_SciHandler_RI, spasConnId, serverConnId);
}
else {
NodeConn_setDisconnectedState();
sendInternal_I_NetConn_NodeConn_NetConn_nodeDisconnectCfm(cid,
                                                          port_NetConn_RI);
NodeConn_removeData();
}}
else {}}
else {}
```

C

Fig. 3. UML State-machine (A) and `NodeConn_riDisconnectCfm` Operation in ALF (B) and Related Generated C++ (C)

```
/* Create the MMOOP::If statement from the ALF model */
helper handleIf(inout smt : ALF::IfStatement){
  var ris = smt.mapIf();
  CURRENT_OPERATION.body += getIf(ris.expr, ris.blocks, ris.blockELSE);
  return null;
}

/**    Return all if conditional expressions, blocks and else block
 *     Else-If conditional expressions and blocks are also included in the
       returned sets */
helper ALF::IfStatement::mapIf() : expr:OrderedSet(MMOOP::Expression),
       blocks:OrderedSet(ALF::Block), blockELSE:ALF::Block {
  var allif = self.sequentialClausses->asOrderedSet().getAllIfStatements();
  var e : OrderedSet(MMOOP::Expression);
  allif.cond->forEach(c){
    var condExp := c.getExpression(false);
    if(not condExp.oclIsUndefined())then{
      e += condExp;
    }endif;
  };
  expr += e;
  blocks := allif.block->asOrderedSet();
  blockELSE := self.finalClause.block;
}

/* Create the MMOOP::If statement */
query getIf(in conds : OrderedSet(MMOOP::Expression), in bodiesEIF :
        OrderedSet(ALF::Block), in bodyElse : ALF::Block) : MMOOP::If{
  var IF := object MMOOP::If{};
  if(conds->size() = bodiesEIF->size())then{
    var index := 0;
    conds->forEach(con){
      index := index + 1;
      var cb := object MMOOP::ConditionalBlock{};
      cb.condition := con;
      handleAlfBlockInIF(bodiesEIF->at(index).oclAsType(ALF::Block),cb);
      IF.ifblocks += cb;
    };
    if(not bodyElse.oclIsInvalid())then{
      var elseBl := object MMOOP::ElseBlock{};
      handleAlfBlockInELSE(bodyElse, elseBl);
      IF.elseBlock := elseBl;
    }endif;
  }endif;
  return IF;
}
```

Fig. 4. Transformation from ALF's `if` statement to Intermediate MM's `if` statement

`FixedValue` set to *RI_RESPONDED_CFM* as right hand side. The second `Assign` will contain, except for an `AccessPart` to a newly defined variable *secondHalf* (as `ScopedNamedInstance`) as left hand side, an `Invocation` of the operation with `Signature` 'NodeConn_getSecondHalf' and a `ValueAccess` for the parameter 'serverConn_r.connHalf'.

VI. Validation & Discussion

Besides several tests upon UML models, the approach has been successfully validated within the CHESS project (see Acknowledgements) against an industrial case-study [19] at Ericsson Nikola Tesla (Zagreb, Croatia) under the leadership of Ericsson AB (Stockholm, Sweden) using the CHESS Modelling Language. The cross-domain Composition with Guarantees for High-integrity Embedded Software Components Assembly modelling language (CHESS-ML) [20] has been defined as a UML profile, including tailored subsets of SysML and MARTE profiles. The CHESS-ML's development style follows the component-based pattern (as defined in the UML Superstructure Specification [21]) and exploits component and composite component diagrams for structural design while state-machines and ALF for expressing functional behaviour. Especially thanks to the possibility of defining executable

behaviours by means of ALF, we reached the necessary expressive power to be able to generate 100% of the implementation directly from the functional models with no need for manual fine-tuning of the code after its generation [3].

The transformation process described in this work mainly consists of two model transformations. The model-to-model transformation transforming ALF into intermediate concepts is composed of 104 transformation rules for a total of 3296 code lines (the most complex transformation within the whole UML–ALF to C++ process). The model-to-text transformation from intermediate model to C++ is made of 82 rules (most of which dedicated to the translation of behavioural concepts) for a total of 832 code lines.

Adopting a multi-step transformation approach and therefore exploiting intermediate modelling artefacts (i.e., intermediate metamodel) gives us the possibility to easily extend the transformation process to enable the generation of code in programming languages different from C++. This can be achieved by acting only on the model-to-text transformation, which represents the least intricate transformation step in the generation chain. Generation of C++ sister languages such as Java or C# would require lightweight adaptations to the existing model-to-text transformation while, for generating other types of languages, the transformation would need to be implemented from scratch. Instead, in the case of ALF undergoing modifications, the model-to-model transformation would need to be properly adapted.

The generality of the intermediate metamodel (and somewhat its intricacy) allowed us to be able to translate any combination of (the covered) ALF statements and expressions to intermediate concepts. Examples of this could be complex conditional logic expressions embedding multiple nested invocations, value accesses, and indexed values accesses, just to mention a few. Regarding conditional logic and arithmetic expressions, we enforced operator precedence defined in the ALF specification [2] to adhere to C++ in order to achieve correct translation. This was needed since the overuse of parenthesized expressions in ALF would severely slow down the parser provided by Papyrus. Such is due to the structure of the ALF metamodel, that, for each parathesized expression (at any level of the hierarchical tree) entails the generation of a whole new tree branch. This problem, together with other ALF-related minor issues found out along the way, has been communicated to the Papyrus development team and has recently been assigned for resolution.

VII. Conclusion

In this paper we described the definition of an automatic mechanism for the translational execution of ALF, meant as the translation of the ALF text, as well as surrounding UML concepts, into a non-UML target language to be executed on a non-UML target platform. More specifically, we defined a target-agnostic intermediate metamodel resembling common object-oriented programming languages to which ALF concepts are translated to. The actual translation process from ALF concepts to a non-UML target language (i.e., C++) is defined and implemented by means of model-to-model and model-to-text transformations.

The translation of surrounding UML concepts was considered as black-box in this work since the focus was on novel mechanisms for the translation of ALF. While in this work we covered most of the concepts for the ALF minimum syntactical conformance level, future enhancements would certainly consist in completing such coverage to achieve at least full minimum conformance by entailing OCL-related concepts (e.g., expressions related to sequences) too.

Acknowledgements

This research work was supported by the RALF3 (SSF, http://www.mrtc.mdh.se/projects/ralf3/) and CHESS (ARTEMIS, http://chess-project.ning.com) projects.

References

[1] J. Bézivin, "On the Unification Power of Models," *Software and System Modeling*, vol. 4, pp. 171–188, 2005.

[2] OMG, "Action Language For FoundationalUML - ALF," http://www.omg.org/spec/ALF/1.0/Beta2/, April 2013.

[3] F. Ciccozzi, A. Cicchetti, and M. Sjödin, "Round-Trip Support for Extra-functional Property Management in Model-Driven Engineering of Embedded Systems," *Information and Software Technology*, August 2012.

[4] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Syst. J.*, vol. 45, no. 3, pp. 621–645, 2006.

[5] S. Kent, "Model Driven Engineering," in *Procs of IFM'02*. Springer-Verlag, pp. 286–298.

[6] OMG, "Foundational Subset For Executable UML Models (FUML)," http://www.omg.org/spec/FUML/1.1/, April 2013.

[7] Eclipse Projects, "Papyrus," http://www.eclipse.org/papyrus/, April 2013.

[8] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose, *Eclipse Modeling Framework*. Addison Wesley, 2003.

[9] M. Brun, J. Delatour, and Y. Trinquet, "Code Generation from AADL to a Real-Time Operating System: An Experimentation Feedback on the Use of Model Transformation," in *Procs of ICECCS 2008*, april 2008, pp. 257–262.

[10] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers. Principles, techniques, and tools"*. SAO/NASA Astrophysics Data System, 1986.

[11] M. Fredj, A. Radermacher, S. Gerard, and F. Terrier, in *Procs of NOTERE 2010, title=eC3M: Optimized model-based code generation for embedded distributed software systems, year=2010, month=june, pages=279-284,.*

[12] W. Haberl, M. Tautschnig, and U. Baumgarten, *Generating Distributed Code From COLA Models*, ser. Lecture Notes in Electrical Engineering. Springer, March 2009, vol. 33, ch. 20.

[13] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet, "A co-design approach for embedded system modeling and code generation with UML and MARTE," in *Procs of DATE 2009.*, april 2009, pp. 226–231.

[14] M. Usman, A. Nadeem, and T. hoon Kim, "UJECTOR: A Tool for Executable Code Generation from UML Models," in *Procs of ASEA 2008*, 2008, pp. 165–170.

[15] T. Moreira, M. Wehrmeister, C. Pereira, J.-F. Petin, and E. Levrat, "Automatic code generation for embedded systems: From UML specifications to VHDL code," in *Procs of INDIN 2010*, 2010, pp. 1085–1090.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[17] S. Boyko, R. Dvorak, and A. Igdalov, "The Art of Model Transformation with Operational QVT," http://www.eclipse.org/m2m/qvto/doc/EclipseCon_2009.ppt, March 2009.

[18] Eclipse Projects- Xpand, "Xpand," http://www.eclipse.org/modeling/m2t/\-?project=xpand, April 2013.

[19] N. Katanic and M. Perse, "Application of CHESS Methodology A Telecom Use Case Study," in *SoftCOM - MDE Workshop*, 2012.

[20] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi, and T. Vardanega, "CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems," in *ASE*, 2012, pp. 362–365.

[21] Object Management Group (OMG), "UML Superstructure Specification V2.3," http://www.omg.org/spec/UML/2.3/Superstructure/PDF/, 2011, [Online. Last access: 11/04/2012].