Tom-based tools to transform EMF models in avionics context

Jean-Christophe Bach[1,2,3], Pierre-Etienne Moreau[1,2,3], and Marc Pantel[4]

[1] Inria, Villers-lès-Nancy, F-54600, France
[2] Université de Lorraine, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54500, France
[3] CNRS, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54500, France
jeanchristophe.bach@inria.fr, pierre-etienne.moreau@loria.fr
[4] INPT-IRIT, Université de Toulouse, Toulouse, France
marc.pantel@enseeiht.fr

Abstract. Model Driven Engineering (MDE) is now widely used in many industrial contexts (such as AeroSpace) which require a high level of system safety. Model-checking is one of the formal techniques which are used to assess a system compliance to its requirements. It relies on verification dedicated languages to model the system under verification and the expected properties. In order to ease the use of these tools, model transformations are provided that translate the end user provided system model to the formal languages than can be verified. In order to rely on these activities for system certification, the correctness of these transformation steps must be assessed (qualification of the development and verification tools). One of the goal of our work is to provide tools to implement the transformation steps between end user source languages used for the system development and target languages used for formal verification. This paper present the Tom rule-based approach used in a research project involving industrial partners: Airbus and Ellidiss.

1 Introduction and related work

New safety critical system development chains are based on Domain Specific Modeling Languages (DSML) and on qualified model transformations (that preserve required properties) between these DSMLs. The quarteFt[5] project aims to develop technologies to make this approach easier in the field of real-time embedded systems. The associated case study relies on Fiacre[6], which is an intermediate language used for formal verification of real-time aspects in the TOPCASED[7] project. Starting from an avionic model expressed in AADL[8], the goal is to extract a behavioral model and its intended properties which can be verified by a

[5] The quarteFt project (http://quarteft.loria.fr/) is funded by FNRAE (http://www.fnrae.org/), a French research foundation for aeronautics and space.
[6] http://projects.laas.fr/fiacre/
[7] http://www.topcased.org/
[8] Architecture Analysis & Design Language, formerly Avionics Architecture Description Language, http://www.aadl.info/

model-checker such as TINA[9] or CADP[10]. To make this transformation simpler, several intermediate languages have been introduced: basic and parametrized Fiacre; and a real-time extension to Fiacre (RT-Fiacre). The following parts show how a rule based language, implemented in the Tom framework [1,2], can be used to express the AADL2Fiacre transformation in an expressive and executable way.

In MDE , many domain specific languages (DSLs) have been developed to manipulate and transform models, trying to implement the Query View Transformation QVT standard defined by the Object Management Group (OMG). QVT proposes two main approaches: relational and operational. The latter requires to describe explicitly the control as part of the transformation whereas it is handled implicitly by the execution engine for relational one. The operational approach expresses a transformation as a sequence of elementary steps that builds the resulting model step by step from the source model. It can be implemented using DSL such as Kermeta[3], QVT-Operational, or GPL (General Purpose Languages) such as Java using reflexive libraries or generated code such as Eclipse Modeling Framework (EMF) [4]. The relational approach defines a model transformation as the relations that must exist between the source and the target models at the end of the transformation. Some transformations are not directly executable. When they can be executed, they are usually translated into an operational transformation.

A third approach [5] can be considered: the introduction of intermediate representations such as XML or any textual concrete syntax. The model transformation can then be described using a language such as XSLT or any concrete language transformation toolset like ASF+SDF[11], Rascal [6], Spoofax(based on Stratego/XT) [7,8], *etc.*

Implementations of transformations may be written using GPL or DSL. Both of these approaches have advantages and drawbacks: GPL are usually well equipped with multiple libraries and external tools and are well known from the common software engineer. However some specific tasks may be difficult to implement whereas DSL may be more precise and efficient. Integration and maintainability has also to be taken in account when choosing a language to transform models. DSLs are usually not known by most software engineers and a significant cost must be spent in their learning and the development of associated tools.

ATL [9] does not rely on the QVT concrete syntax but implements most of the operational and part of the relational approaches from QVT. It fills a part of the gap (use of both relational and operational approches), but is still a full new language that must be learnt from scratch.

This contribution proposes to bridge previous approaches by embedding a DSL into a GPL, to take advantages both of GPLs tooling and DSL expressivity. As the Tom language is compiled into Java relying on model management libraries like EMF, the end user can benefit from both classical programming, and model transformation technologies (operational and relational), without major

[9] http://www.laas.fr/tina/

[10] http://cadp.inria.fr/

[11] http://www.meta-environment.org/Meta-Environment/ASF%2BSDF/

efficiency penalty and without the additional cost of learning a completely new language. Therefore Java developers have only to learn few new constructs to use within a Java program to easily implement a model transformation, contrary to the use of a dedicated transformations language such as Kermeta which are more complete for this task but also much more complex. By remaining in the Java world, developers can benefit from a third advantage of our approach: keep using existing Java tooling.

The rest of the paper is organized as follows. Section 2 presents the Tom-based tools we developed to transform EMF models. Section 3 introduces the transformations chain and the industrial use case AADL2Fiacre, then it explains the use of Tom in this context. Section 4 concludes and presents current and future work.

2 Tom-based tools to transform EMF models

Tom is a language designed to extend general-purpose programming languages (Java, C, C#, Python and Caml) by providing term rewriting, pattern matching and strategic programming facilities. Tom is well-suited for implementing programs that manipulate tree structures such as AST (Abstract Syntax Tree) or XML documents. Only the main constructs of the language that are used in the case study will be presented, but the interested reader may refer to the Tom reference manual[12].

2.1 Short introduction to Tom

Pattern-matching. The main Tom feature is pattern-matching using the **%match** construct. It is a generalization of *switch-case* construct which can be applied to any data-structure, and not only to atomic types like *switch-case*. It is composed of a set of rules where the left-hand side is a pattern (having a tree structure), and the right-hand side is an action (a block of host and Tom code).

Backquote construct. The backquote (character ‘) is the Tom construct which allows to build a new algebraic term or to retrieve the value of a variable instantiated by pattern matching.

Listing 1.1 illustrates the use of **%match** and ‘: at line 1, a `TypeDeclaration` is created with ‘. It has two parameters: `rec` is the value of the type, and `"int"` is the name of the type. Then, at line 2, this `TypeDeclaration` is given as the parameter (called *subject*) of the **%match** construct. Line 3 shows how to write a transformation rule: on the left-hand side, the patterns checks that the subject corresponds to a `TypeDeclaration`. In this example, the variable `name` is instantiated by the first argument (`"int"` in this case). The second argument of the pattern is a nested pattern which recursively checks that the second subterm of `typeDecl` is rooted by `RecordType_fiacre`, *etc.*. It is also possible to consider

[12] `http://tom.loria.fr/`

disjunction (||) and conjunction (&&) of patterns. The right-hand side of the rule is the action (-> {...}), composed of a block of Tom+Java code. In this example, the variable name is printed.

```
1 TypeDeclaration typeDecl = `TypeDeclaration("int",rec);
2 %match(typeDecl) {
3   TypeDeclaration(name,RecordType_fiacre(RecordTypeLabelEList())) -> {
4     System.out.println(`name);
5     ...
6   }
7 }
```

Listing 1.1: Example of %match and backquote constructs

Strategic-programming. Strategic programming is a way to increase the control over the application of rewriting rules. Strategies — implemented by the **%strategy** construct — allow to dissociate the treatment (rewriting rules) and the control (tree traversal). Then complex strategies can be obtained by composition of elementary transformation strategies and combinators such as Sequence, Repeat, TopDown, and the recursion for instance. For the interested reader, more detail can be found in [10]. Listing 1.2 shows the use of **%strategy** construct and how a strategy can be composed and called (line 8). DataTrans() is an elementary strategy. TopDown(DataTrans()) is also a strategy. visit is a method which *apply* a strategy on a given term (subject in this example).

```
1 %strategy DataTrans() extends Identity() {
2   visit DataImplementation {
3     dt@DataImplementation[name=name] -> { <Tom+Java code block> }
4   }
5 }
6 public static void main(String[] args) {
7   ...
8   `TopDown(DataTrans()).visit(subject);
9   ...
10 }
```

Listing 1.2: Example of %strategy construct

Algebraic views (mappings). In order to be able to use pattern-matching feature, one has to establish an *anchor* between the implementation data-structure (*i.e.* the Java classes) and the Tom algebraic signature. This is done by a mechanism called *mapping*. It is composed of two constructs: **%typeterm** and **%op**. As shown in Listing 1.3, **%typeterm** associates the algebraic sort (DataImplementation) to the implementation data-type (org.osate.aadl2.DataImplementation) with the construct **implement**. The **equals** construct defines an equality predicate.

```
1 %typeterm DataImplementation {
2   implement { org.osate.aadl2.DataImplementation }
3   equals(t1,t2) { t1.equals(t2) }
4 }
```

Listing 1.3: Example of %typeterm construct

The **%op** construct illustrated in Listing 1.4 specifies how to view a Java object as a Tom constructor. The predicate **is_fsym(t)** should be true when the Java object t can be seen as the considered constructor (DataImplementation

in this example). **get_slot** specifies how to retrieve a given field of the data-structure. **make** defines how to build an instance of the constructor and is used by ‘ construct to build terms. This example relies on the EMF reflexive framework. It could also rely on the generated classes. However, these ones are also implemented using the reflexive framework. This work thus also follows that scheme.

```
1 %op DataImplementation DataImplementation(name : String, ..., noPrototypes : boolean) {
2   is_fsym(t) { t instanceof org.osate.aadl2.DataImplementation }
3   get_slot(name, t) { t.eGet(t.eClass().getEStructuralFeature("name")) }
4   get_slot(noPrototypes, t) { t.eGet(t.eClass().getEStructuralFeature("noPrototypes")) }
5    ...
6   make(name, ..., noPrototypes, ...) { new DataImplementation(name, ..., noPrototypes) }
7 }
```

Listing 1.4: Example of %op construct

%oplist and **%oparray** are variant of **%op** for list operators (see Listing 1.5) used for associative and associative/commutative operators. Their use is similar to **%op**, but they define additional constructs to build a list and to retrieve a given element in a list, and the size of a list.

```
1 %typeterm RecordTypeLabelEList {
2   implement { org.eclipse.emf.common.util.EList<fiacre.RecordTypeLabel> }
3   equals(t1,t2) {  ...  }
4 }
5 %oparray RecordTypeLabelEList RecordTypeLabelEList ( RecordTypeLabel* ) {
6   is_fsym(t) { ... }
7   make_empty(n) { /* allocate a list of size n */ }
8   make_append(e,l) { /* add an element e to the end of the list l */ }
9   get_element(l,n) { /* get the n-th element of the list l */ }
10  get_size(l)      { /* returns the size of the list l */ }
11 }
```

Listing 1.5: Example of variadic operator mapping

2.2 Tom-EMF: from EMF representation to Tom representation

Tom-EMF is a tool whose goal is to offer a support to manipulate EMF data-structures with Tom. It is composed of two main elements: `EcoreContainmentIntrospector` which is used for using strategies on EMF elements, and a mappings generator. This latter loads and inspects a Java meta-model implementation generated by Eclipse from an ECore file. It retrieves all the *EClassifier* (*EClass* and *EDataType*) and *EStructuralFeatures* in order to generate the corresponding **%typeterm** and **%op** constructs. These mappings will be the view for EMF elements in Tom. For instance, the mappings presented in Listing 1.3 and Listing 1.4 have been automatically generated by Tom-EMF from the `aadl2fiacre.jar` archive, which corresponds to java classes which are automatically generated by EMF from an ECore meta-model. If a reference shows that an element may have many instances (which is represented by a collection), the Tom-EMF tool generates list operators such as `RecordTypeLabelEList` illustrated in Listing 1.5.

2.3 %transformation: an easy way to express a model transformation in Tom+Java

To write a model transformation, one can use full Java or Tom+Java. In the case of Tom+Java, every elementary transformation composing the whole transformation can be encoded by a strategy. This approach is appropriate for simple transformations, but in some cases, an elementary transformation is dependent on elements created by another one. For instance, let us consider two transformations transA and transB, which respectively transform elements of sorts A and B. The result of transA may need to be connected to some elements produced by transB. A way to express these connexions is thus needed.

This problem can be addressed in two ways:

1. a first solution consists in finding a general strategy (usually a recursive one) which inspect the model and transforms the elements in an order such that all the links can be built during the transformation itself. Unfortunately such a strategy does not always exist.
2. a second approach consists in not considering any particular order for the transformation steps. In that case, an additional mechanism has to be provided in order to denote and to build a link to an element which results from a transformation. This operation is called *link resolution* in the following. This approach is derived from the one available in ATL and the relational part of QVT.

A method using the second solution to implement a models transformation with Tom+Java is explained in [11]. It constitutes the core contribution of the new higher-level Tom **%transformation** construct. It addresses the problem and hides a part of the work from the developer. **%transformation** is a construct composed of a set of *definitions* (corresponding to Tom strategies) which may have one or many rules (corresponding to rules of rewriting strategies). An example is given in the next section (see Listing 1.9). The grammar of **%transformation** construct is given by Listing 1.6 in Appendix A.

As the execution order of transformation steps is not considered, some *definition* may need to reference an element created in another transformation step *definition* in order to be able to assemble the elements resulting from the various steps. It also may create an element which has to be referenced in another transformation step. That is the reason why we introduced two new Tom constructs to be able to perform the link resolution: **%tracelink** and **%resolve**. They build a *link model* during the transformation steps to keep references between the source and the target. Then, it is used to *resolve* the links once all target elements have been created. These constructs ensure a form of traceability of the transformation, which is a mandatory step to be able to verify it.

%tracelink whose syntax is given by Listing 1.7 in Appendix A allows to save in the link model an element which will be necessary in another step. It takes three parameters: the name and the type the element has, and the term itself. In the link model, a field of the defined type having the given name will

be created. The object will be instantiated with the object represented by the backquote term.

%resolve construct whose grammar is shown in Listing 1.8 (in Appendix A) can be considered as the corresponding construct of **%tracelink**. It creates a *virtual* element the developer can manipulate in a *definition* as if it were the one he needs (whether it has effectively already been created or not). This kind of elements is removed at the end of the transformation during the link resolution. The first two parameters of a **%resolve** construct are the name and the type of a source element. The last two parameters are the name and the type of a resulting element obtained by the transformation of the source element in another definition.

3 An industrial use case in avionics: AADL2Fiacre

This case study consists in transforming an AADL model into a Fiacre model in order to verify this one with TINA. This concrete use case is being implemented by the Ellidiss Software[13] company in Adele[14] to address the needs of Airbus[15] in the quarteFt project.

Important note: due to industrial constraints, all source code and documents produced during quarteFt project are still unavailable for public use. All materials will be released publicly after the end of the project.

AADL. AADL [12] is an architecture description language initially developed for avionics domain. It is designed for modeling the software and hardware architecture of embedded real-time systems, more particularly in the context of automotive engineering, aeronautics and aerospace. The purpose of AADL language is to describe an architecture to be able to analyze the system, to verify properties, to generate code and documentation, *etc.*. Components may be defined to model a part of the system. Each component may contain one or many sub-components, and it may be connected to other components to build a more complex system by simulating a network. In the context of quarteFt, AADL is the starting point of the transformations chain.

Fiacre. Fiacre[16] is a high level description language designed for representing both the behavioral and timing aspects of embedded and distributed systems for formal verification and simulation purposes. In the quarteFt project, the last transformation step is the compilation of Fiacre descriptions into Time Transition Systems (TTS) for TINA.

[13] `http://www.ellidiss.com/`
[14] TOPCASED AADL editor : `http://gforge.enseeiht.fr/projects/adele/`
[15] `http://www.airbus.com`
[16] Fiacre stands for "Format Intermédiaire pour les Architectures de Composants Répartis Embarqués", french for "Intermediate Format for the Embedded Distributed Component Architectures".

For a detailed explanation of Fiacre language, the reader is invited to refer to [13] where the use case is a first proposal of AADL2Fiacre transformation written in Kermeta. Although ATL and Kermeta have been experienced and were useful tools to precise a part of the project, major issues concerning performance and integration constraints led to consider Tom as the transformation language used in quarteFt. Moreover, the declarative way to write a transformation in Tom is well adapted to correctness proof construction.

AADL2Fiacre. Because of the complexity of the whole transformation and the lack of place, this work focuses on a part of the step which transforms AADL model to RT-Fiacre (Real-Time Fiacre) model. This transformation can be divided into six elementary transformations: `AADL.Data`, `AADL.Thread`, `AADL.ThreadImpl`, `AADL.BehaviorAnnex`, `AADL.System` and `AADL.Process` translations.

Due to space constraints and because this transformation is still under development, the six elementary transformations will not be detailled. Snippets of code will be extracted from the shortest and easiest one: the translation of AADL *Data* into RT-Fiacre *Record*.

To begin this transformation, one has to generate the Tom mappings to have an algebraic representation of models. Starting from source meta-model `aadl2.ecore` [17] and target meta-model `fiacre.ecore`[18], one generates with EMF the corresponding Java code which will be used during the transformation. Then, one uses Tom-EMF tool to generate AADL and Fiacre mappings.

This transformation step consists in generating a RT-Fiacre type-record (`RecordType_fiacre`) for each *Data* component (implemented by `DataImplementation`. Each data-field declared in the *subcomponents* part corresponds to a field of the RT-Fiacre record.

By using Tom, this elementary transformation can be implemented by a single strategy composed with a `TopDown` traversal strategy or being a *definition* of a **%transformation** construct as shown in Listing 1.9 (Appendix B).

As a **%transformation** is compiled like a composition of strategies, the application and use of `AADL2Fiacre` transformation is like every other strategy as illustrated in Listing 1.10 (Appendix B).

A special strategy `tom_StratResolve_AADL2Fiacre()` is generated when needed (in case of use of **%tracelink** and **%resolve** constructs). It replaces all *resolve* objects ("virtual" ones created by the **%resolve** construct) by the corresponding and therefore it resolves all links. It allows to the developer not to take care of the application order of transformation steps.

In a pure Java approach, Tom facilities such as pattern-matching should be replaced by lots of hand-written tests and iterators (for list-matching). Strategies would be replaced by hand-written traversal of the source model, using loops

[17] Can be found in Osate2 plugin from Ellidiss update site: `http://aadl.ellidiss.fr`

[18] An ECore metamodel has been defined to integrate Fiacre in the TOPCASED environment, and therefore can be found in the TOPCASED bundle.

and recursivity. In addition, the developer would have to be very careful on the instructions sequence in order not to use not yet created objects.

4 Conclusions, future work

This paper presented tools designed to transform more easily EMF models in a general purpose language such as Java. These tools are based on the Tom language which relies on strong formal foundations (rewriting calculus); has been developed since year 2000; and has been used in many research and industrial project. As the Tom language is composed of a limited set of constructs that developers include in host languages programs, it is easy to learn and its use allows developers to be quickly efficient. One the one hand its DSL approach is useful to express a model transformation in an easier way than in pure Java. On the other hand, its integration in a GPL allows to take advantage of existing tooling and libraries. To complete this goal, Eclipse plugins for Tom and Tom-EMF are also being developed.

A real industrial case study which is currently being developed in an avionics-related project was also described. Our goal is to provide an efficient tool usable in an state-of-the-art industrial context.

Tom-EMF and extensions of the Tom language are currently experiencing a testing period. The first version of **%transformation** construct will be available in the Tom 2.10 release.

These tools should be improved in many ways. For the moment, Tom models transformations tools are exclusively based on EMF. The support of multi-modeling frameworks such as GMS[19] is currently being implemented. Another focus is the extension of **%transformation** construct to support multiple inputs. With this feature, the pattern-matching possibilities will be extended to be able to match several patterns at the same time in an elementary transformation. The extension and the generalization of the EMF-specialized *introspector* to be able to easily follow other links than containment associations in a model is also studied. As the verification of models transformations is one of our core concerns, the traceability of a model transformation is being investigated. The start point of this part consists in working on the improvement of the currently generated link meta-model; and then to apply constraints (as in OCL) on this link meta-model to ensure that properties are verified.

Finally, this work represents a first step towards the increase safety in software development chains through the design of new tools and the verification of models transformations.

[19] Ada modeling framework used in the P project: `http://www.open-do.org/projects/p/`

References

1. Moreau, P.E., Ringeissen, C., Vittek, M.: A pattern matching compiler for multiple target languages. In Hedin, G., ed.: Compiler Construction. Volume 2622 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2003) 61–76
2. Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A.: Tom: piggybacking rewriting on java. In: Proceedings of the 18th international conference on Term rewriting and applications. RTA'07, Berlin, Heidelberg, Springer-Verlag (2007) 36–47
3. Jézéquel, J.M., Barais, O., Fleurey, F.: Model driven language engineering with kermeta. In: Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III. GTTSE'09, Berlin, Heidelberg, Springer-Verlag (2011) 201–221
4. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. 2nd edn. Addison-Wesley Professional (2009)
5. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. Software, IEEE **20**(5) (2003) 42–45
6. Klint, P., van der Storm, T., Vinju, J.J.: Rascal: A domain specific language for source code analysis and manipulation. In: SCAM, IEEE Computer Society (2009) 168–177
7. Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E.: Code generation by model transformation: a case study in transformation modularity. Software and System Modeling **9**(3) (2010) 375–402
8. Kats, L.C.L., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and ides. In Cook, W.R., Clarke, S., Rinard, M.C., eds.: OOPSLA, ACM (2010) 444–463
9. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming **72**(1-2) (June 2008) 31–39
10. Balland, E., Moreau, P.E., Reilles, A.: Rewriting strategies in java. Electr. Notes Theor. Comput. Sci. **219** (2008) 97–111
11. Bach, J.C., Crégut, X., Moreau, P.E., Pantel, M.: Model transformations with tom. In: LDTA, Tallinn, Estonia, ACM (2012) To appear.
12. Feiler, P.H., Lewis, B.A., Vestal, S.: The sae architecture analysis & design language (aadl) a standard for engineering performance critical systems. In: Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE. (oct. 2006) 1206 –1211
13. Berthomieu, B., Bodeveix, J.P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., Lang, F., Vernadat, F.: Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In: ERTS 2008, Toulouse, France (2008)

Appendices

A Syntax of new Tom constructs dedicated to models transformation

```
TransformationConstruct ::= '%transformation' TransformationName '('[TransformationArguments]')'
':' FileName '->' FileName '{' (Definition)+ '}'
TransformationArguments ::= SubjectName ':' AlgebraicType ( ',' SubjectName ':' AlgebraicType )*
                          | AlgebraicType SubjectName ( ',' AlgebraicType SubjectName )*
Definition        ::= 'definition' DefinitionName 'traversal' Strategy '{' (DefinitionRule)* '}'
DefinitionRule    ::= Pattern '->' '{' BlockList '}'
```

Listing 1.6: %transformation construct syntax

Many elements are note defined in this syntax block in order to have a readable **%transformation** syntax block. `TransformationName`, `FileName`, `SubjectName`, `AlgebraicType` and `DefinitionName` are `Identifier` with self-explanatory names. `Strategy` represents a Tom strategy such as `‘TopDown(MyStrategy()). Pattern` is a pattern as defined in the Tom grammar on the official Tom website. `BlockList` is a block composed of both host and Tom code, as defined in the Tom grammar.

```
TracelinkConstruct ::= '%tracelink' '(' VarName ':' TypeName ',' BackQuoteTerm ')'
VarName            ::= Identifier
TypeName           ::= Identifier
```

Listing 1.7: %tracelink construct syntax

```
ResolveConstruct ::= '%resolve' '(' VarName ':' TypeName ',' VarName ':' TypeName ')'
VarName          ::= Identifier
TypeName         ::= Identifier
```

Listing 1.8: %resolve construct syntax

In these two syntax blocks, `VarName` and `TypeName` are `Identifier`s representing variable and type names.

B Implementation on the first step of AADL2Fiacre transformation

```
1  %transformation AADL2Fiacre() with(aadl2.ecore) to(fiacre.ecore) {
2    definition DataTrans traversal ‘TopDown(DataTrans) {
3      dt@DataImplementation[name=name,ownedDataSubcomponent=dataSubc] -> {
4        TypeDeclaration typeDecl =
5          ‘TypeDeclaration(name,RecordType_fiacre(RecordTypeLabelEList()));
6        translator.fiacrePg.getDeclarations().add(typeDecl);
7        EList<RecordTypeLabel> fields = ((RecordType_fiacre)typeDecl.getValue()).getFields();
8        %match(‘dataSubc) {
9          DataSubcomponentEList(_*,child@DataSubcomponent[name=childName,
10           dataSubcomponentType=dataSubcT],_*) -> {
11             String pkgTypeName = ((AadlPackage)‘dataSubcT.getElementRoot()).getName();
12             String childTypeName = ‘dataSubcT.getName();
13             RecordTypeLabel rtl = ‘RecordTypeLabel(
14                 childName,
15                 TypeAccess(getTypeDeclaration(
```

```
                        translator.fiacreRecordTypeList,
                        pkgTypeName,
                        childTypeName))
                    );
                fields.add(rtl);
            }
        }
        typeDecl.setValue(`RecordType_fiacre(fields));
        translator.fiacreRecordTypeList.put(`name, typeDecl);
      }
    }
    definition ThreadTrans traversal `TopDown(ThreadTrans) { <Tom+Java code> }
    definition ThreadImplTrans traversal `TopDown(ThreadImplTrans) { <Tom+Java code> }
    definition BehaviorAnnexTrans traversal `TopDown(BehaviorAnnexTrans) { <Tom+Java code> }
    definition SystemTrans traversal `TopDown(SystemTrans) { <Tom+Java code> }
    definition ProcessTrans traversal `TopDown(ProcessTrans) { <Tom+Java code> }
}
```

Listing 1.9: AADL2Fiacre implementation

```
public static void main(String[] args) {
    ...
    Strategy transformer = `AAdL2Fiacre();
    transformer.visit(source, new EcoreContainmentIntrospector());
    `TopDown(tom_StratResolve_AADL2Fiacre()).visit(res, new EcoreContainmentIntrospector());
    ...
}
```

Listing 1.10: Use and call of a transformation