(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
20 March 2014 (20.03.2014)

WIPO | PCT

(10) International Publication Number
WO 2014/040766 A1

(51) International Patent Classification: *G06F 9/45* (2006.01)

(21) International Application Number: PCT/EP2013/062411

(22) International Filing Date: 14 June 2013 (14.06.2013)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
92071 12 September 2012 (12.09.2012) LU

(71) Applicant: **UNIVERSITE DU LUXEMBOURG** [LU/LU]; 162a, avenue de la Faïencerie, L-1511 Luxembourg (LU).

(72) Inventors: **HERMANN, Frank**; App. 610, Peter-Lambert-Strasse 6, 54292 Trier (DE). **BRAATZ, Benjamin**; Hornstrasse 42, 54294 Trier (DE). **ENGEL, Thomas**; Rubenstrasse 3, 54294 Trier (DE).

(74) Agents: **BRUCK, Mathis** et al.; Office Ernest T. Freylinger S.A., 234, route d'Arlon, BP 48, L-8001 Strassen (LU).

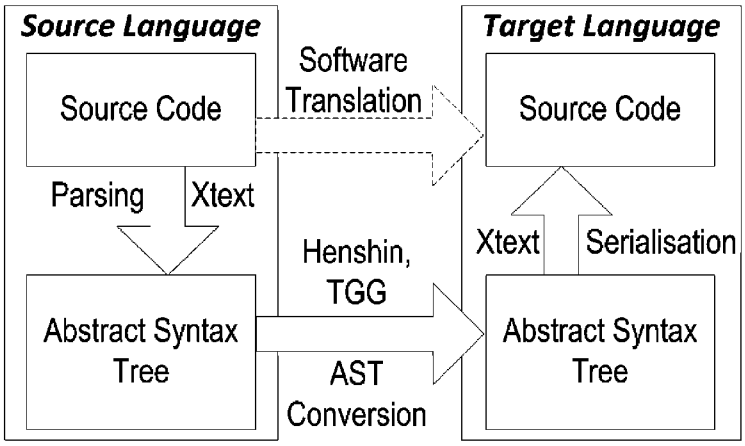
(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report (Art. 21(3))*

(54) Title: COMPUTER-IMPLEMENTED METHOD FOR COMPUTER PROGRAM TRANSLATION

Fig. 1



(57) Abstract: A computer-implemented method for translating computer programs from a source language into a target language comprises translation of a first abstract syntax tree (AST) representing the source language program into a second AST representing the target language program. Translation of the first AST is achieved through applying translation rules. The first AST is a graph typed over a type graph representing the source language grammar, whereas the second AST will be a graph typed over a type graph representing the target language grammar. Each translation rule, when applied, yields a substructure of the second AST, which is in accordance with the target language grammar, i.e. after each translation step, the graph generated on the target side is typed over the target language type graph. The translation rules, which are triple rules, also build a correspondence graph between the first AST and the graph built on the target side.

WO 2014/040766 A1

COMPUTER-IMPLEMENTED METHOD FOR COMPUTER PROGRAM TRANSLATION

Technical field

[0001] The present invention generally relates to software translation, i.e. to the transfer of source code available in one programming language to corresponding source code of another programming language.

Background Art

[0002] System migration is an important but complex task, especially for enterprises that are highly depending on the reliability of their running systems. Such system migrations often involve the need for translating computer code written in a first programming language into a second programming language (more) compatible with the new system to be set up. Software translation may also be desirable out of other considerations. Irrespective of the reasons for migration, from an application point of view, several requirements have to be met. These include, e.g., automation, usability, maintainability, and, most importantly, reliability, which means that the translated computer code shall not alter the original behaviour of the software.

[0003] Up to now, this problem was addressed based on manually written converters, parser generators, compiler-compilers or meta-programming environments using term rewriting or similar techniques.

Technical problem

[0004] It is an object of the present invention to provide an improved method for translation of computer programs, which facilitates validating the translation routines and thus contributes to achieving a high level of fidelity, precision and correctness. This object is achieved by a method as claimed in claim 1.

General Description of the Invention

[0005] The method for translating computer programs from a source programming language into a target programming language is implementable by computer and comprises the translation of a first abstract syntax tree (AST)

representing the computer program in the source language into a second AST representing the computer program in the target language. The translation is achieved by executing a model transformation based on triple graph grammars. Specifically, the translation of the first AST comprises providing a set of predefined translation rules, of which each translation rule, when applied, results in

- generation of a substructure (i.e. a set of one or more nodes and/or one or more edges) of the second AST, the substructure of the second syntax tree being a translation of a predefined specific substructure of the first AST;
- generation of a correspondence indicator indicative of correspondence between the substructure of the first AST and the generated substructure of the second AST; and
- associating with the substructure of the first AST a marking indicating that the substructure of the first AST has been translated.

Each translation rule is associated with an applicability condition, the applicability condition defining which substructure of the first AST and which substructure of the second AST and which correspondence indicator is a prerequisite for applying the translation rule. The translation of the first AST further comprises attempting to apply a translation rule of the set of translation rules, the attempting comprising checking whether the applicability condition of the translation rule is satisfied, and, if the applicability condition is satisfied, applying the translation rule, or, if the applicability condition is not satisfied, attempting to apply another translation rule of the set of translation rules.

[0006] As will be appreciated, the translation of the first AST into the second AST is achieved by executing a model transformation based on triple graph grammars (TGGs), eventually yielding the AST of the target language.

[0007] Basic concepts of triple graphs are provided in the following. A triple graph G is an integrated model consisting of a source model, a target model and explicit correspondences between them. More precisely, it consists of three graphs G_S, G_C, and G_T, called source, correspondence, and target graph, respectively, in combination with two mappings (graph morphisms) $s_G : G^C \rightarrow G^S$ and $t_G : G^C \rightarrow G^T$. Attribute values of nodes and edges are defined as links to the

actual values according to [EEHP09, EEPT06]. The two mappings in G specify a correspondence relation between elements of G^S and elements of G^T. Triple graph morphisms $m: G = (G^S, G^C, G^T) \rightarrow H = (H^S, H^C, H^T)$ [Sch94] specify mappings between triple graphs and consist of three graph morphisms, $m^S: G^S \rightarrow H^S$, $m^C: G^C \rightarrow H^C$ and $m^T: G^T \rightarrow H^T$, that preserve the associated correspondences, i.e., the diagrams below commute (diagrams are said to commute if all directed paths in the diagram with the same start and endpoints lead to the same result by composition of the corresponding morphisms).

$$\begin{array}{ccccc} G = (G^S & \xleftarrow{s_G} & G^C & \xrightarrow{t_G} & G^T) \\ m\downarrow \quad m^S\downarrow & & m^C\downarrow & & m^T\downarrow \\ H = (H^S & \xleftarrow[s_H]{} & H^C & \xrightarrow[t_H]{} & H^T) \end{array}$$

[0008] Triple graphs are “typed over” a type triple graph TG by a triple graph morphism $type_G: G \rightarrow TG$, such that TG plays the role of a metamodel. It is required that morphisms between typed triple graphs preserve the typing. For $TG = (TG^S \leftarrow TG^C \rightarrow TG^T)$, $VL(TG)$, $VL(TG^S)$ and $VL(TG^T)$ denote the classes of all graphs typed over TG, TG^S and TG^T, respectively.

[0009] A triple rule $tr = (tr^S, tr^C, tr^T)$ is an inclusion of triple graphs, represented $L \hookrightarrow R$. It specifies how a given consistent integrated model (triple graph) can be extended simultaneously on all three components, yielding again a consistent integrated model. A triple rule is applied to a triple graph G via a match morphism $m: L \rightarrow G$, resulting in the triple graph H, where L is replaced by R in G. This means that match morphism m specifies an occurrence of the left hand side L of rule tr within G, where m maps each element (node, edge, or attribute) in L to an element in G. This mapping is consistent with typing and the internal graph structure. The transformation step may be defined by a pushout diagram [EEPT06] and it may be denoted by $G \overset{tr,m}{\Rightarrow} H$. Triple rules may be extended by negative application conditions (NACs) for restricting their application to specific matches [EEHP09, HEGO10].

[0010] A triple graph grammar $TGG = (TG, S, TR)$ consists of a type triple graph TG, a start triple graph S and a set TR of triple rules, and generates the triple graph language $VL(TGG) \subseteq VL(TG)$ containing all consistent integrated models.

[0011] The first AST mentioned hereinabove is a graph typed over a type graph representing the grammar of the source programming language, whereas the second AST (to be generated) will be a graph typed over a type graph representing the grammar of the target programming language. Each translation rule, when applied, yields a substructure of the second AST, which is in accordance with the grammar of the target language, i.e. after each translation step, the graph generated on the target side (i.e. the second AST in construction) is typed over the type graph representing the grammar of the target language. The correspondence indicators generated by the translation rules form a correspondence graph between the first AST and the graph built on the target side. The translation rules used for translating the first AST into the second AST are thus triple rules. A translation rule is only applicable if the triple graph (comprising the first AST, the correspondence graph formed by the correspondence indicators, and the graph being built on the target side) generated till the current step provides an occurrence of the left hand side (LHS) of the translation rule. In this regard, it shall be noted that the “left hand side” of a graph production rule (in particular of a translation rule) designates the context graph (i.e. the graph on which the rule is applicable), whereas the “right hand side” (or RHS) designates the graph resulting from the application of the rule. Elements appearing only on the RHS of a rule form the “produced graph”.

[0012] To ascertain that each element (node edge or attribute) or substructure of the first AST is translated only once, each translation rule, when applied, generates a marking indicating that the element or substructure of the first AST has already been translated. Before a translation rule is applied (again) it is first checked whether the substructure that it would translate has not yet been translated. Only if the translation marker is absent may the translation rule be applied. In other words, the presence of the translation marker may be considered as a NAC.

[0013] The present method provides an efficient tool for ensuring that the individual translation rules together form a consistent set which correctly translates any computer code which is well-formed (syntactically correct) under the grammar of the source programming language. The method furthermore facilitates ensuring the syntactical correctness of the graph generated on the target side, since the translation rules enforce conformity of the substructure of the second AST with the type graph representing the grammar of the target programming language. Nonconformity of a translation rule can be easily detected at the draft stage. The method also helps to guarantee the completeness of the translation because residual untranslated elements of the first AST have neither an associated translation marker nor a correspondence indicator. Since each translation rule establishes a one-to-one correspondence between a substructure of the first AST and a substructure of the second AST, the desired behaviour of each substructure can be validated. Syntactical correctness of the complete program in the target language is then guaranteed through the compliance with the grammar of the target language and the correct behaviour of the complete program can be inferred from the validation of the substructure translation rules. Once a complete set of translation rules has been defined, each translation of a specific computer program available in the first programming language can be executed fully automatically, without requiring user interaction during the translation itself.

[0014] According to a preferred embodiment of the invention, the method comprises parsing the source computer program so as to generate the first AST. Parsing is executed by a parser using the grammar of the source language provided in a convenient format.

[0015] Preferably, the method comprises serializing (inverse parsing) the second AST. Serialization uses the grammar of the target language.

[0016] Prior to attempting to apply the translation rules (i.e. in a so-called initialization phase), the first abstract syntax tree is preferably extended with additional elements (nodes, edges and/or attributes) that may be used during the translation phase. For instance, during the initialization phase, one or more identifiers may be inserted into the first AST, in such a way as to disambiguate different instances of a same statement. Such identifiers are especially useful on

elements of the first AST, the translation of which involves the ad-hoc naming of a construct (node, edge, attribute and/or a more complex AST substructure) in the second AST. Such situations may arise, for instance, if the substructure in the second AST is more complex than its antecedent in the first AST and involves the generation of nodes, edges and/or attributes, which, individually, have no direct counterpart in the first AST. Translation rules translating such substructure of the first AST preferably take the identifiers to name constructs within the substructure of the second syntax tree.

[0017] According to a preferred embodiment of the invention, during the initialization phase, (e.g. Boolean-valued) translation attributes are inserted into the first AST, each translation attribute being associated with a node or an edge of the first AST and each translation attribute being initialized with a value indicating that the associated node or edge has not yet been translated. In this case, the action of associating of a marking with a substructure of the first AST being translated comprises giving the translation attributes of any node and edge of the substructure of the first AST a value indicating that the associated node or edge has been translated.

[0018] The action of checking the applicability condition associated with a translation rule is preferably carried out as a pattern matching step. Before a specific substructure can be translated by application of the corresponding translation rule, all information needed for the correct translation must be available, especially in the target graph. Through checking the applicability condition of each rule, it is made sure that a certain substructure is ready for translation. Intuitively, the translation of a parent node of the first AST requires that any child node of that parent node has been translated previously (unless parent and child node belong to a substructure, which is always translated as a whole) in order for the translation rule to be able to link the translated child node to the translated parent node with an appropriate edge. From this it follows that certain translation rules may typically be executed earlier than others.

[0019] This is advantageously used for runtime optimization of the translation. The most inefficient part in graph transformation is the matching phase, i.e., finding an occurrence of the left hand side of a translation rule within the current

host graph. The basic approach to the execution of a model translation via TGGs is to take the complete set of translation rules and apply them as long as possible. Thus, for each translation step, the search for the next rule to apply can be very inefficient, since, in the worst case, all rules have to be checked for valid matches until a valid match is found. The translation rules are thus preferably distributed in at least a first and a second subset of the set of translation rules. During the translation phase, it is then attempted to apply translation rules of the second subset of translation rules only if the applicability condition of each of the translation rules of the first subset of translation rules is no longer satisfied. The first subset comprises the translation rules that are applied first. Among these translation rules are preferably those, which translate terminal nodes of the first AST into terminal nodes of the second AST. The rules of the second subset are only translated after those of the first subset have been exhaustively applied. Among the translation rules of the second subset are preferably those, which translate residual non translated edges of the first AST into edges of the second AST. It is worthwhile noting that there may be more than two classes (subsets) of translation rules, which are applied one after the other. For software translation, there are usually several translation rules that show cyclic dependencies. This is the case for example for translation rules concerning expression structures. When distributing the translation rules into the different subsets that determine the order of their application during the translation, the translation rules are preferably analyzed according to their potential dependencies. This includes the case of rules that have application conditions. The automatic dependency analysis may e.g. be performed using the tool AGG, e.g. its critical pair analysis engine. If there are cyclic dependencies, then all rules involved in one cycle are placed in one group (subset). Rules that do not belong to a cycle are placed in a separate group. All resulting groups are ordered according to the dependency relation on the rules. The execution of the translation is then preferably performed as follows: start with the first subset and iterate over all subsets according to the dependency order of the subsets; apply the rules of the current subset as long as possible until no rule of this subset is applicable.

[0020] Model translation based on TGGs may encounter efficiency problems that are caused by defining translation rules that translate a graph node together with

all possible attribute values. The problem concerning efficiency is that for each combination of possible attribute values, one may need to define a separate translation rule. Therefore, the amount of rules increases. Moreover, if some attribute values are not mandatory in the corresponding model domain, some rules may need to be extended with additional application conditions (in particular, with negative application conditions). Application conditions are likely to increase the complexity for the matching process, making it less efficient. Therefore, in accordance with a preferred embodiment of the invention, if several rules are used to translate one node type together with the contained attributes, then these rules are replaced by a set of rules that translate each attribute and the node separately. The resulting rules are potentially smaller in size and amount. This reduces the amount of execution time needed for the matching and execution.

[0021] Model translation based on TGGs may apply the translation rules directly to the given input model. In some cases, however, it may be important to preserve the ordering of elements from the source domain model and to propagate the ordering to the resulting output model in the target domain. If the ordering needs to be preserved in the target domain to avoid corruption of the resulting output model, the ordering of elements is preferably rendered explicit in the input model by explicit edges connecting a predecessor with its successor node. According to a preferred embodiment of the invention, a pre-processing phase may be carried out, during which the graph representation of the input model (which is the abstract syntax tree to be translated) is traversed. The pre-processing preferably creates a link from each parent node to the first child node. Additional links between all child nodes may be created to obtain an explicit list structure. This structure can be a doubly linked list (next and previous pointers) to improve efficiency for matching. Additionally, if the child lists are grouped in several node types, further pointers can be created to mark the beginning of the sub lists. The extended input model is then used for the translation. The translation rules use the additional explicit links to specify the order between the elements that is propagated to the target domain. As an alternative to this pre-processing, one could extend the graph transformation tool in the matching component and the user interface to allow for the explicit specification of ordering information between rule elements. However, this would come at the cost of reduced efficiency of the

matching engine and the translation rules would need additional application conditions to specify that an element has either no predecessor or no successor. This would most probably slow down the translation engine as well.

[0022] A further aspect of the present invention relates to a computer program comprising computer-executable instructions, which when executed by a computer, cause the computer to implement a method as described herein. Another aspect of the invention relates to a data processing installation comprising a memory and a processor, the memory having stored therein instructions executable by the processor, which when executed by said processor, cause the processor to implement a method as described herein and to produce output data representing the translated computer program, i.e. the computer program in the target language.

Brief Description of the Drawings

[0023] A preferred embodiment of the invention will now be described with reference to a compact running example relating to the translation from one very simple programming language to another. The running example is illustrated by the accompanying drawings in which:

Fig. 1 is a block schematic diagram of a preferred method for translating computer code from one programming language to another;

Fig. 2 is an illustration of the grammar of the source programming language of the example;

Fig. 3 is an illustration of the EMF model corresponding to the grammar of Fig. 2;

Fig. 4 is an illustration of the abstract syntax tree corresponding to the example computer program in the source language;

Fig. 5 is an illustration of the grammar of the target programming language of the example;

Fig. 6 is an illustration of the EMF model corresponding to the grammar of Fig. 5.

Fig. 7 is a block schematic diagram illustrating different sub-phases of a preferred AST conversion process;

Fig. 8 is a representation of the initialization rules used in the translation example;

Fig. 9 is a representation of a first part of translation rules used in the translation example;

Fig. 10 is a representation of a second part of translation rules used in the translation example;

Fig. 11 is a representation of a third part of translation rules used in the translation example;

Fig. 13 is a representation of the refactoring rules used in the translation example;

Fig. 14 is an illustration of the abstract syntax tree corresponding to the example computer program in the target language.

Description of Preferred Embodiments

[0024] *Introduction of an example translation*

[0025] The running example is based on the two simple programming languages "L_{repeat}" (source domain) and "L_{while}" (target domain). The example program, which will be referred to hereinafter, is the following:

```
/# "myloop" #/
REPEAT
   READ a
   READ b
   READ c
UNTIL    ((a EQ b) AND
              (b EQ c))
```

[0026] Language L_{repeat} contains logical expressions (comparisons EQ of two variables, composed by connectives AND, OR, NOT and grouped by parentheses), assignments (:=) to variables, READ statements and comments (/#…#/).

[0027] The example program is to be translated into language L_{while}, which contains conditions with connectors && (logical AND), || (logical OR), ! (logical

negation), variables, comparisons (==), assignments to variables (=), input() functions and comments (/#…#/). L_{repeat} furthermore allows defining and calling functions without parameters.

[0028] The main task is to translate REPEAT-UNTIL-loop structures into while structures. It has to be taken into account that, whereas the termination condition of a REPEAT loop is checked after executing the loop body, the termination condition of a while loop is checked before. It must also be taken into account that loops can be nested.

[0029] The main idea for the translation from L_{repeat} to L_{while} according to the example program is to create a while loop in the target domain for each REPEAT loop occurring in the given source code in L_{repeat}. In order to preserve the semantics of the REPEAT loop, the translated loop body has to appear once before the translated loop to ensure at least one execution of the body (as in the source programming language). In order to avoid code duplication, the body of the loop is encapsulated in an external function, which will be called at the appropriate places, i. e., before starting the while loop and within the body of the same loop. In this example, the function name is generated out of the prefix "_f" extended by user comments, which are placed directly below the REPEAT statement in the source language, or extended by an automated numbering created during a so-called initialisation phase.

[0030] Furthermore, the translation will refactor the source code in the target domain by applying De Morgan laws in order to evaluate the negated condition into a simpler statement. The resulting desired source code of the translation including is the following:

```
1        /# "myloop" #/
2        def _f0()   {
3            a = input() ; b = input() ; c = input() ;
4        } ;
5        _f0();
6        while((!(a == b) || !(b == c))) {
```

7 _f0() ; }

[0031] *Phases of the translation*

[0032] The solution concept in Fig. 1 illustrates the successive steps of the translation. Software written in the source language is parsed, resulting in a first AST that represents the whole source code (phase 1). In the second phase (AST conversion), graph transformation rules are applied to the first AST, yielding a graph that contains an AST (second AST) of the program in the target language. This phase comprises of three sub-phases, which will be addressed in more detail hereinafter. The last phase is the serialisation of the second AST, whereby the corresponding source code in the target language is generated.

[0033] The software translation is preferably based on the Eclipse Modeling Framework (EMF) tools Xtext [Xte12] and Henshin [Hen11]. Xtext supports the syntax specification of textual domain specific languages (DSLs), in particular of programming languages. Based on the Extended Backus-Naur Form (EBNF) grammar specification of a DSL, the Xtext framework may generate the corresponding parser and serialiser. Henshin is an Eclipse plugin supporting the visual specification and execution of EMF transformation systems.

[0034] *Grammar of the source language*

[0035] Fig. 2 shows the Xtext grammar of language L_{repeat}. Based upon this grammar, Xtext is able to generate a parser (and a serialiser) based on the Eclipse Modeling Framework (EMF). Fig. 3 represents the EMF model of language L_{repeat}. Each production rule of the Xtext grammar (Fig. 2) is mapped to a node in the EMF model in Fig. 3 with the non-terminal of the left hand side (LHS) as name. Each labelled non-terminal symbol on the right hand side (RHS) of a rule is mapped to an edge in the EMF model with the label as name, the node of the rule as source and the node of that rule which is referenced by the non-terminal as target. Each unlabelled non-terminal symbol on the RHS of a rule is mapped to an inheritance relation in the EMF model with the node of the rule as source and the node of that rule which is referenced by the non-terminal as target. The last rule of the Xtext grammar depicted in Fig. 2 specifies an abstract node type “Source”, which allows reducing the amount of TGG rules required for the translation.

[0036] Given an input file of the source language, the parser yields an EMF instance conforming to the EMF model of the source language, which is contained in the generated parser plugin. Additionally, the parser checks that the input source code is well-formed (i.e. in conformity with the provided grammar). After parsing the source code in language L_{repeat}, using the generated Xtext parser plugin, one obtains the first AST, illustrated in Fig. 4. The first AST is a concrete graph typed over the EMF model (type graph) of language L_{repeat} shown in Fig. 3. The first AST is used as input for the AST conversion, leading to an AST (the second AST) of the target domain language L_{while}.

[0037] *Grammar of the target language*

[0038] Fig. 5 shows the Xtext grammar, Fig. 6 the EMF model (type graph) of language L_{while}.

[0039] The first line of the grammar defines its name and specifies the import of additional terminal production rules “ID” and “STRING” from grammar *org.eclipse.xtext.common.Terminals* which are provided by Xtext. Rule “ID” allows alphanumerical sequences not starting with a number and rule “STRING” allows arbitrary sequences of characters. Programs written in language L_{while} consist of a list of fragments. Start production rule “Wprogram” in line 5 denotes the root element of each target program and refers to the first fragment list element of the program with relation “fst”. Each fragment can optionally refer to its succeeding fragment with relation “next” in line 6. The unlabelled non-terminal symbols “While“, “Var_Def“, “Fn_Call”, “Fn_Def” and “Comment” of rule “Fgmnt_LST_Elem” define while statements, variable definitions, function calls, function definitions and comments as being special types of fragments. Rule “While” claims that each while statements contains a Boolean expression “expr” as termination condition and at least one fragment “fgmnts” in the body of the loop. Boolean expressions can be of type unary or binary as indicated by production rules “Expr” and “Expr_T”. Production rule “Unary” defines unary Boolean expressions as either logical negations, Boolean variables or input commands that return a Boolean value when being executed. Rule “Neg” allows negating each logical expression “expr”. Rule “Var” defines variables as alpha-numerical sequences not starting with a number. Input commands are of the form input() as defined by rule “Input”.

Binary Boolean expressions contain a first expression “fst” as left argument, a second expression “snd” as right argument and an infix operand “&&”, “||” or “==” as defined by rule “Binary”. Variable definitions are composed of a variable on the left and a Boolean expression on the right side that are connected by an equality terminal symbol as depicted by rule “Var_Def” in line 8. Production rule “Fn_Call” defines function calls as the name of the function “nameF” which is being called, followed by opening and closing brackets and a semicolon. Function definitions contain the name of the function “nameF” consisting of an alpha-numerical sequence not starting with a number and at least one fragment as the body of the function definition as depicted by rule “Fn_Def”. Comments consist of arbitrary strings enclosed by a “/#” terminal on the left and a “#/” terminal on the right side. To define the correspondences between source and target language elements in the operational translation rules, rule “Target” specifies an abstract type, from which all other types in the target language inherit, i.e., target programs (“Wprogram”), fragments (“Fgmnt_LST_Elem”), Boolean expressions (“Expr”) and expression types (“Expr T”) as elements of the target programming language.

[0040] As for the source language, Xtext may be used to automatically generate an EMF model out of the defined Xtext grammar for the target language. The EMF model serves as type graph restricting the creation of instance models of that language. For the target language L_{while}, the type graph presented in Fig. 6 is created according to the corresponding grammar in Fig. 5. Each production rule of the Xtext grammar is mapped to a node in the EMF model, whereas the name of the node is taken from the name of the non-terminal in the LHS of the production rule. Labels for non-terminal symbols of the production rules are mapped to edges in the EMF model connecting two nodes, i.e., between two non-terminals. The source of the edge is the node of the rule corresponding to the LHS of the rule, the target of the edge is given by a node corresponding to the referenced non-terminal symbol. Unlabeled non-terminal symbols are mapped to inheritance relations in the EMF model, where the source of the inheritance relation is given by the node of the rule. The target is given by the referenced node.

[0041] Based upon this grammar, Xtext is able to generate a serialiser (and a parser) based on the EMF. Once the second AST has been generated, that

serialiser may be used to generate the source code of the program in the target language (here L_{while}).

[0042] *AST conversion*

[0043] Fig. 7 schematically shows the different sub-phases of the AST conversion according to a preferred embodiment of the invention. The first sub-phase, referred to as “initialization”, extends the given source AST (AST_S) of the source language L_s with additional elements that provide information derived from AST_S leading to an extended graph G_S. The added elements are useful for the subsequent sub-phase, i.e. the AST translation, during which a model transformation based on triple graph grammars (TGGs) is executed, yielding the corresponding AST (second AST or AST_T) representing the computer program in the target language L_T. The final sub-phase (called refactoring) refines the second AST in order to satisfy certain coding guidelines that may be required in the target domain, while preserving the behaviour of the program in the target language. Moreover, refactoring rules may be used for optimization of the code in terms of execution time. It is worthwhile noting that the refactoring sub-phase may normally be considered optional.

[0044] Each step of the TGG model transformation in the translation sub-phase (i.e. each application of a translation rule) takes some substructure of the given source model and creates the corresponding structure in the target domain. All information needed to translate a substructure must be available in a single transformation step (an instance of application of a specific translation rule) in the translation sub-phase, because each element of the source model is translated exactly once. This requirement means that the matching process for a TGG transformation rule must ensure that all the relevant information for performing a correct translation step is available. A match of a graph transformation rule (such as a translation rule) is restricted by the LHS of the rule and by optional additional application conditions [EEPT06, EEHP09]. Thus, matching can take into account some bounded context of the structure to be translated. However, information that depends on more complex expressions, such as path expressions, cannot be computed in this phase.

[0045] For this reason, the initialisation sub-phase precedes the application of the translation rules. During the initialization sub-phase, a set of graph transformation rules is applied to compute this information. In the running example, the initialisation rules are used to compute a disjoint numbering of the loops and to store this information in additional comments in the first AST. These numbers serve as unique identifiers of the generated function names in the resulting target program.

[0046] Moreover, translation markers are added during the initialization sub-phase. Each element (node, edge or attribute) of the first AST is extended by a Boolean valued translation attribute. Intuitively, translation attributes serve to mark the elements that have been translated so far. All the translation attributes ("tr") are initially set to "false". During the translation phase, each translation rule marks the element(s) of the first AST it translates by changing its (their) translation attribute(s) to "true". This marks each translated element of the first AST as having been translated and prevents that any of the translated elements is translated again, since it is one of the applicability conditions of each translation rule that elements to be translated must not contain the marking that they have already been translated.

[0047] Fig. 8 presents the initialisation rules, which are applied to graphs in the source domain, i.e., graphs typed over the source type graph shown in Fig. 3. Rules "init_TR", "init_TR_fst" and "init_TR_next" are applied exactly once, such that for each initially found match, the rules are executed. They create translation attributes that are used in the AST translation sub-phase.

[0048] Rule "init_TR" creates translation attributes within each node derived from the node "Source".

[0049] Henshin does not provide the specification of attributes for edges, because this feature is not available for EMF instance models. For this reason, the rules "init_TR_fst" and "init_TR_next" generate nodes for each edge "fst" and "next", respectively, containing translation attributes for those edges. This encoding has to be handled carefully, since it has to be ensured that for each edge, exactly one marker is created.

[0050] As indicated above, all translation attributes are initially set to "false".

[0051] Rule “init_Program_Counter” creates an extra node containing a counter value initially set to “0”. This counter is used for annotating REPEAT statements in order to achieve unique function names, which will be generated in applying rule “t_Repeat2While” in Fig. 9 during the translation sub-phase in order to externalize the body of each REPEAT loop. The negative application condition (NAC) prevents the generation of more than one counter node. Note that it is assumed that each graph contains exactly one program, i.e., one node of type ”RProgram”, to guarantee valid results when applying rule “init_Repeat_Counter”. Rule “init_Repeat_Counter” numbers each REPEAT statement in copying the value of the program counter and in creating a comment node containing the newly created identifier for the REPEAT loop, i.e., for creating a unique function name during the translation sub-phase. In addition, this rule increases the program counter. If the REPEAT statement already has a comment, the value of this comment will be used as identifier for the function (c.f. NAC). Then, this rule will not be applied.

[0052] Figs. 9, 10 and 11 show forward translation rules for translating statements of source programming language L_{repeat} to fragments of target programming language L_{while}.

[0053] The translation rules presented in this document are shown in short notation, which means that the LHS and the RHS of the rule are depicted in one and the same triple graph. The LHS (context graph) comprises all elements (nodes and edges) not labeled “++”, the produced graph comprises all elements labeled “++”, whereas the RHS of the rule comprises the entire graph shown in the respective figure.

[0054] Rule “t_RProgram2Wprogram$_{FT}$” concerns the translation of the program root element. When applying the rule, the untranslated root element “Rprogram” in the source domain is matched and the corresponding root element “Wprogram” is created in the target domain together with an explicit correspondence (node of type “Corr”). The translation attribute of the root element “Rprogram” is updated to “true”.

[0055] Rule ”t_Comment2Comment$_{FT}$” translates comments by matching the untranslated comment “Comment” with content “c” in the source domain and creating the corresponding comment “Comment” with the same content “c” in the

target domain together with an explicit correspondence (node of type “Corr”) when being applied. Finally, the translation attribute of the source comment “Comment” is updated to “true”.

[0056] Rule ”t_Asg2Var_Def$_{FT}$” translates assignment statements of the source domain to variable definitions of the target domain. The application of the rule leads to the matching of the untranslated assignment statement ”Asg” and the creating of the corresponding variable definition “Var_Def” in the target domain together with an explicit correspondence (node of type “Corr”). The translation attribute of the assignment statement “Asg” is updated to “true”. The variable “Var” that corresponds to the already translated right symbol “Sym” of the assignment statement “Asg” is taken as the right expression of the variable definition “Var_Def” by creating an expression “Expr” of type “Var” which is connected by edge “right” with node “Var_Def” in the target domain. Analogously, the corresponding variable of the left symbol of the assignment statement is taken as the left element of the variable definition by creating edge “left” in the target domain.

[0057] Translation rule ”t_Repeat2While$_{FT}$” translates the repeat loop of the source language L_{repeat} into the while loop of the language L_{while}. For rule ”t_Repeat2While$_{FT}$” to be applicable, the translation attribute of the elements to be translated (in this case the node “Repeat” and the edges “stmnt” and “expr”) must initially be “false”. It is further required that the “Comment” context element containing the loop number and the “Log_Expr” logical expression for the termination condition have already been translated (tr is “true”, corresponding translated elements are present on the target side and correspondence nodes are present in the correspondence graph). When applying the rule, an untranslated ”Repeat” loop structure in the source domain is matched and the corresponding ”While” loop structure is created in the target domain together with an explicit correspondence (node of type “Corr”). The translation attribute of the “Repeat” node is updated to ”true”. As can be seen from Fig. 9, a more complex substructure is generated in the target domain. Specifically, a function definition is created together with function calls before as well as inside the While loop. Furthermore, the “Expr” conditional expression in the target domain is negated (Neg) and attached to the “While” node as its termination condition. The translation

of the actual body and the translation of the remaining parts of the program are handled by further translation rules.

[0058] Rule "t_Read2Var_Def$_{FT}$" translates a read statement of the source domain to a variable definition in the target domain and requires that the parameter symbol "Sym" is already translated (translation attribute "tr" is "true"). When applying the rule, an untranslated read statement "Read" in the source domain is matched and the corresponding variable definition "Var_Def" is created in the target domain together with an explicit correspondence (node of type "Corr"). The translation attribute of the read statement "Read" is updated to "true" and in the target domain, an expression of type "input" is created as the right parameter of the variable definition "Var_Def" to represent the "read" command of the source domain in the target domain.

[0059] Fig. 10 shows the forward translation rules for translating logical expressions of source programming language L_{repeat} to Boolean expressions of target programming language L_{while}.

[0060] Rule "t_Sym2Var$_{FT}$" translates symbols of the source domain to variables of the target domain. When applying the rule, an untranslated symbol "Sym" with label "x" in the source domain is matched and the corresponding variable "Var" with the same label "x" is created in the target domain together with an explicit correspondence (node of type "Corr"). The translation attribute of the symbol "Sym" is updated to true.

[0061] Rule "t_Log_Neg2Neg$_{FT}$" concerns the translation of a logical negation and requires that the logical expressions "Log_Expr", which is being negated is already translated (translation attribute "tr" is "true"). When applying the rule, an untranslated negation "Log_Neg" in the source domain is matched and the corresponding negation "Neg" is created in the target domain together with an explicit correspondence (node of type "Corr"). The translation attribute of the negation "Log_Neg" is updated to "true" and in the target domain, the translated negation "Neg" is referenced to the already translated expression "Expr" by creating edge "expr".

[0062] Rule "t_Log_Expr_Binary2Binary$_{FT}$" concerns the translation of binary logical expressions of the source domain to binary Boolean expressions of the

target domain. The rule requires that both logical expressions “Log_Expr”, i.e., the first and second expression, that are referenced by a binary logical expression “Log_Expr_Binary” with edges “fst” and “snd” are already translated (translation attribute “tr” is “true”). When applying the rule, an untranslated binary logical expression “Log_Expr_Binary” in the source domain is matched and the corresponding binary expression “Binary” is created in the target domain together with an explicit correspondence (node of type “Corr”). The operator of the binary expression is mapped as follows by the expression in node “Binary” of the target domain: Operator “AND” is mapped to “&&”, “OR” to “||” and “EQ” is mapped to “==”. Alternatively, the rule can be splitted into three rules, i.e., one rule for each operator. The translation attribute of the logical binary expression “Log_Expr_Binary” is updated to “true”. The expression “Expr” that corresponds to the already translated first logical expression “Log_Expr” is taken as first expression of binary expression “Binary” by creating edge “fst” in the target domain. Analogously, the corresponding expression “Expr” of the second logical expression “Log_Expr” is taken as the second expression of the binary expression “Binary” by creating edge “snd” in the target domain.

[0063] Rule “t_Log_Expr2Expr$_{FT}$” translates the root of a logical expression and requires that the logical expression “Log_Expr_T”, which can be of an arbitrary type, is already translated to expression “Expr_T” (translation attribute “tr” is “true”). When applying the rule, the root “Log_Expr” in the source domain is matched and the corresponding root “Expr” is created in the target domain together with an explicit correspondence (node of type “Corr”). The translation attribute of the root “Log_Expr” is updated to “true” and the translated root “Expr” in the target domain is referenced to the corresponding already translated expression “Expr_T” by creating edge “type” in the target domain. Note that logical negations “Log_Neg”, symbols “Sym” and binary logical expressions “Log_Expr_Binary” are special types of logical expression “Log_Expr_T” as indicated by the EMF model in Fig. 3 so that expressions “Log_Expr_T” are translated by rules “t_Sym2Var$_{FT}$”, “t_Log Neg2Neg$_{FT}$” and “t_Log_Expr_Binary2Binary$_{FT}$”.

[0064] Fig. 11 relates to the translation rules for translating residual non translated edges at the end of the translation sub-phase.

[0065] Both translation rules “t_Next2Next$_{FT}$” and “t_Next2Next_Repeat$_{FT}$” create an edge “next” in the target domain between two nodes of the type “Fgmnt_LST_Elem”. Rule “t_Next2Next$_{FT}$” produces edges for all nodes of type “Fgmnt_LST_Elem”, which are connected to a node of the type “Stmnt_LST_Elem” through a node “Corr” and where the edge “next” between the nodes “Stmnt_LST_Elem” is not translated, which is indicated by the translation attribute “tr” within the corresponding node “TR_next”. The rule “t_Next2Next$_{FT}$” will not be applied if the target “Stmnt_LST_Elem” is a node of the specialised type “Repeat” because, to translate the edge “next” leading to a “Repeat” node, also function definitions and function calls need to be inserted before the corresponding node “While”. This will be achieved by executing rule “t_Next2Next_Repeat$_{FT}$”. In applying both rules, the corresponding translation attribute of the edge “next” in the source domain is set from “false” to “true” in the corresponding node “TR_next”.

[0066] Rule “t_Fst2Fst$_{FT}$” translates the edge “fst” on the source side, which is situated between the node “Rprogram” and a node of the type “Stmnt_LST_Elem”, by generating a corresponding edge “fst” from the node “Wprogram” to a node of the type “Fgmnt_LST_Elem” in the target domain. This rule will by applied once, because the translation attribute “tr” in the node “TR_fst” will be set to “true”, which indicates the translation attribute of the edge “fst” in the source domain. The edge “fst” occurs only once, because it is assumed that there exists only one node “Rprogram” (and “Wprogram”, respectively).

[0067] The graph resulting from the application of the translation rules conforms to the type graph of the target domain. In order to ensure that it forms an abstract syntax tree, the TGG translation rules have to ensure that the tree structure of the first AST is preserved by the model transformation. By definition, TGG rules (such as the translation rules) preserve the source model and the execution ensures that elements are translated exactly once. These properties massively simplify the challenge of ensuring that the target graph forms an AST. Intuitively, each path or sub-tree of the source AST is translated into a path or sub-tree in the target graph and attached to the corresponding parent node.

[0068] Fig. 12 illustrates refactoring rules applied during the refactoring sub-phase.

[0069] The refactoring rules are applied to the second AST obtained after the translation rules have been applied exhaustively. By applying the refactoring rules, the second AST is simplified and optimised, which will be reflected in the resulting source code obtained by serialization.

[0070] Rules “refactor_deMorgan_NEGAND” and “refactor_deMorgan_NEGOR” reflect the application of De Morgan laws to logical expressions, i.e., in the example, to the condition within the while loop. These rules will be applied as long as possible to the condition.

[0071] The rules “refactor_Repeat_Counter” and “refactor_Program_Counter” remove temporary comments used for unique identification of the loops. Rule “refactor_Repeat_Counter” removes comments containing all unique function names, but the comment will only be removed if it is numeric. This is achieved by adding the JavaScript attribute condition “!isNaN(parseInt(c))” to that rule in Henshin. After processing all initialisation and translation rules, a node “Comment” containing the program counter is always directly attached to the node “Wprogram” by edge “fst”. Rule “refactor_Program_Counter” removes this first node “Comment” and connects the next fragment node “Fgmnt_LST_Elem” directly to “Wprogram”. Consequently, the internal program counter will be removed by applying refactoring rule “refactor_Program_Counter”.

[0072] Fig. 13 shows the EMF instance model (second AST) of the example program in the target language, which is obtained at the end of the graph conversion phase (initialization, translation and refactoring). The second AST is typed over the EMF model (type graph) of Fig. 6.

[0073] The corresponding source code in the target language is obtained by serializing the second AST using the generated Xtext plugin based upon the Xtext grammar of Fig. 5.

[0074] While a specific example has been described in detail, those skilled in the art will appreciate that various modifications and alternatives to those details could be developed in light of the overall teachings of the disclosure. Accordingly, the

particular arrangements disclosed are meant to be illustrative only and not limiting as to the scope of the invention, which is to be given the full breadth of the appended claims and any and all equivalents thereof.

References

[AGG12] TFS-Group, TU Berlin. AGG – Version 2.0.3. 2012. http://tfs.cs.tu-berlin.de/agg.

[BKVV08] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. Science of Computer Programming 72(1-2):52–70, 2008.

[BPM04] I. Baxter, P. Pidgeon, M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In Software Engineering (ICSE 2004). IEEE Press, 2004.

[Cor06] J. R. Cordy. The TXL source transformation language. Science of Computer Programming 61(3):190–210, 2006.

[Cor11] J. R. Cordy. Excerpts from the TXL cookbook. In Generative and Transformational Techniques in Software Engineering (GTTSE 2009). LNCS 6491, pp. 27–91. Springer, 2011.

[EEHP09] H. Ehrig, C. Ermel, F. Hermann, U. Prange. On-the-Fly Construction, Correctness and Completeness of Model Transformations based on Triple Graph Grammars. In Proc. MoDELS 2009. LNCS 5795, pp. 241–255. Springer, 2009.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Fundamentals of Algebraic Graph Transformation. Springer, 2006.

[EHGB12] C. Ermel, F. Hermann, J. Gall, D. Binanzer. Visual Modeling and Analysis of EMF Model Transformations based on Triple Graph Grammars. 2012. Submitted to GraBaTs.

[GHL12] H. Giese, S. Hildebrandt, L. Lambers. Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars. Ensuring Conformance of Relational Model Transformation Specifications and Implementations. To appear in Software and Systems Modeling, available online, 2012.

[GW09] H. Giese, R.Wagner. From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8(1):21–43, 2009.

[HEGO10] F. Hermann, H. Ehrig, U. Golas, F. Orejas. Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In Model Driven Interoperability (MDI 2010). Pp. 22–31. ACM, 2010.

[Hen11] The Eclipse Foundation. EMF Henshin – Version 0.8.1. 2011. http://www.eclipse. org/modeling/emft/henshin/.

[KS06] A. Königs, A. Schürr. Tool Integration with Triple Graph Grammars. A Survey. In Foundations of Visual Modelling Techniques. ENTCS 148, pp. 113–150. Elsevier, 2006.

[KSV11] P. Klint, T. van der Storm, J. Vinju. EASY Meta-Programming with Rascal. In Generative and Transformational Techniques in Software Engineering (GTTSE 2009). LNCS 6491, pp. 222–289. Springer, 2011.

[KV10] L. C. L. Kats, E. Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010). 2010.

[KVS09] P. Klint, J. J. Vinju, T. van der Storm. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In Source Code Analysis and Manipulation. Pp. 168–177. IEEE Computer Society, 2009.

[KW07] E. Kindler, R. Wagner. Triple Graph Grammars. Concepts, Extensions, Implementations, and Application Scenarios. Technical report TR-ri-07-284, Department of Computer Science, University of Paderborn, 2007.

[PF11] T. Parr, K. Fisher. LL(*): the foundation of the ANTLR parser generator. ACM SIGPLAN Notices 46(6):425–436, 2011.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Graph-Theoretic Concepts in Computer Science. LNCS 903, pp. 151–163. Springer, 1994.

[SK08] A. Schürr, F. Klar. 15 Years of Triple Graph Grammars. In Graph Transformations (ICGT 2008). LNCS 5214, pp. 411–425. 2008.

[Xte12] The Eclipse Foundation. Xtext – Language Development Framework – Version 2.1. 2012. http://www.eclipse.org/Xtext/.

Claims

1. Computer-implemented method for translating a computer program from a source programming language into a target programming language, comprising;

   translating a first abstract syntax tree representing said computer program in said source language into a second abstract syntax tree representing said computer program in said target language;

   characterized in that said translation is achieved by executing a model transformation based on triple graph grammars, said translation comprising

   providing a set of predefined translation rules,

   of which each translation rule, when applied, results in

   - generation of a substructure of said second abstract syntax tree, said substructure of said second syntax tree being a translation of a predefined specific substructure of said first abstract syntax tree;
   - generation of a correspondence indicator indicative of correspondence between said substructure of said first abstract syntax tree and said substructure of said second abstract syntax tree;
   - associating with said substructure of said first abstract syntax tree a marking indicating that said substructure of said first abstract syntax tree has been translated;

   and of which each translation rule is associated with an applicability condition, said applicability condition defining which substructure of said first abstract syntax tree and which substructure of said second abstract syntax tree and which correspondence indicator is a prerequisite for applying said translation rule;

   attempting to apply a translation rule of said set of translation rules, said attempting comprising checking whether the applicability condition of said translation rule is satisfied, and, if said applicability condition is satisfied, applying said translation rule, or, if said applicability condition is not satisfied, attempting to apply another translation rule of said set of translation rules.

2. The method as claimed in claim 1, comprising parsing the source computer program so as to generate said first abstract syntax tree.
3. The method as claimed in claim 1 or 2, comprising serializing said second abstract syntax tree.
4. The method as claimed in any one of claims 1 to 3, wherein, prior to attempting to apply said translation rules, one or more identifiers are inserted into said first abstract syntax tree, said identifiers being inserted in such a way as to disambiguate different instances of a same statement.
5. The method as claimed in claim 4, wherein said one or more of said translation rules take said identifiers to name constructs within said substructure of said second syntax tree.
6. The method as claimed in any one of claims 1 to 5, wherein, prior to attempting to apply said translation rules, translation attributes are inserted into said first abstract syntax tree, each translation attribute being associated with a node or an edge of said first abstract syntax tree and each translation attribute being initialized with a value indicating that the associated node or edge has not yet been translated.
7. The method as claimed in claim 6, wherein the associating of a marking with said substructure of said first abstract syntax tree comprises giving the translation attributes of any node and edge of said substructure of said first abstract syntax tree a value indicating that the associated node or edge has been translated.
8. The method as claimed in claim 6 or 7, wherein said translation attributes are Boolean valued.
9. The method as claimed in any one of claims 1 to 8, wherein checking the applicability condition associated with a translation rule is carried out as a pattern matching.
10. The method as claimed in any one of claims 1 to 9, wherein said set of translation rules comprises at least a first and a second subset of translation rules and wherein it is attempted to apply translation rules of said second subset of translation rules only if the applicability condition of each of the translation rules of the first subset of translation rules is not satisfied.

11. The method as claimed in claim 10, wherein said first subset of translation rules comprises translation rules, which translate terminal nodes of said first abstract syntax tree into terminal nodes of said second abstract syntax tree.
12. The method as claimed in claims 10 or 11, wherein said second subset of translation rules comprises translation rules, which translate residual non translated edges of said first abstract syntax tree into edges of said second abstract syntax tree.
13. Computer program comprising computer-executable instructions, which when executed by a computer, cause said computer to implement a method as claimed in any one of claims 1 to 12.
14. Data processing installation comprising a memory and a processor, said memory having stored therein instructions executable by said processor, which when executed by said processor, cause said processor to implement the method as claimed in any one of claims 1 to 12 and to produce output data representing said computer program in said target language.

Fig. 1

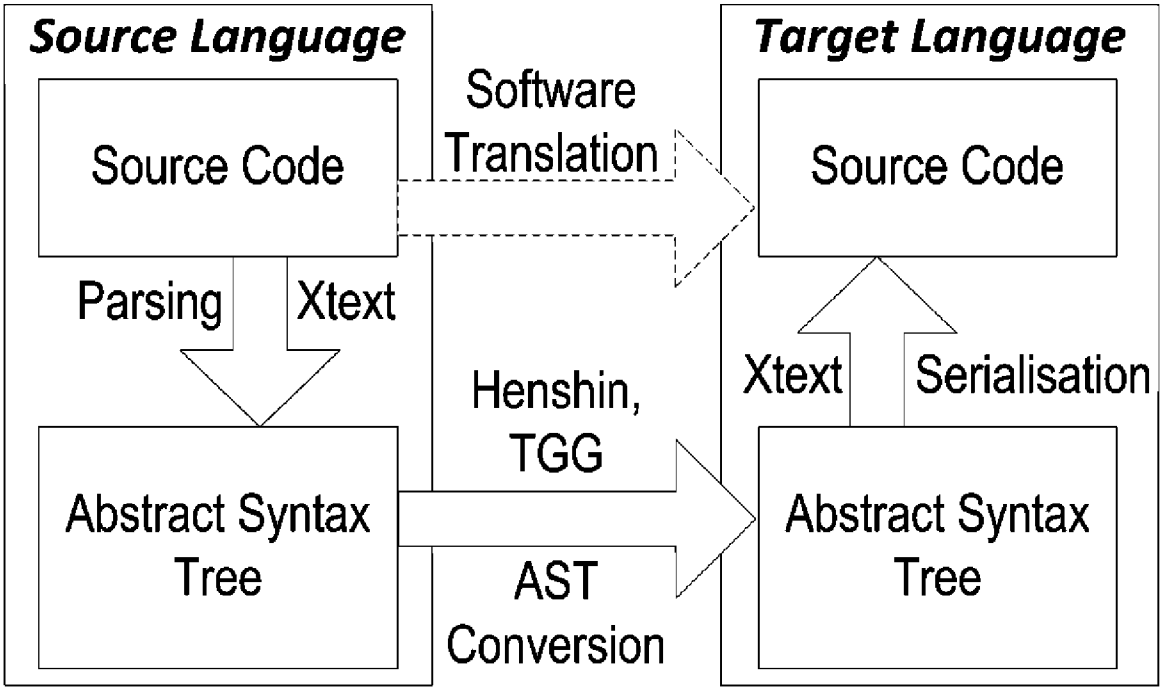


Fig. 2

```
grammar lu.uni.snt.source.Repeat with org.eclipse.xtext.common.Terminals
generate source "http://www.uni.lu/snt/source/Repeat"
RProgram        : fst=Stmnt_LST_Elem;
Stmnt_LST_Elem  : (Repeat | Asg | Read | Comment) (next=Stmnt_LST_Elem)
    ?;
Repeat          : 'REPEAT' stmnt=Stmnt_LST_Elem 'UNTIL' expr=Log_Expr;
Asg             : left=Sym ':=' right=Sym;
Read            : 'READ' param=Sym;
Comment         : '/#' comment=STRING '#/';
Log_Expr        : type=Log_Expr_T;
Log_Expr_T      : Log_Expr_Unary | Log_Expr_Binary;
Log_Expr_Unary  : Log_Neg | Sym;
Log_Expr_Binary : '(' fst=Log_Expr operator=('AND' | 'OR' | 'EQ') snd=
    Log_Expr ')';
Log_Neg         : 'NOT' expr=Log_Expr;
Sym             : sym=ID;
Source          : (RProgram | Stmnt_LST_Elem | Log_Expr | Log_Expr_T)
                  tr?='false'?;
```

Fig. 3

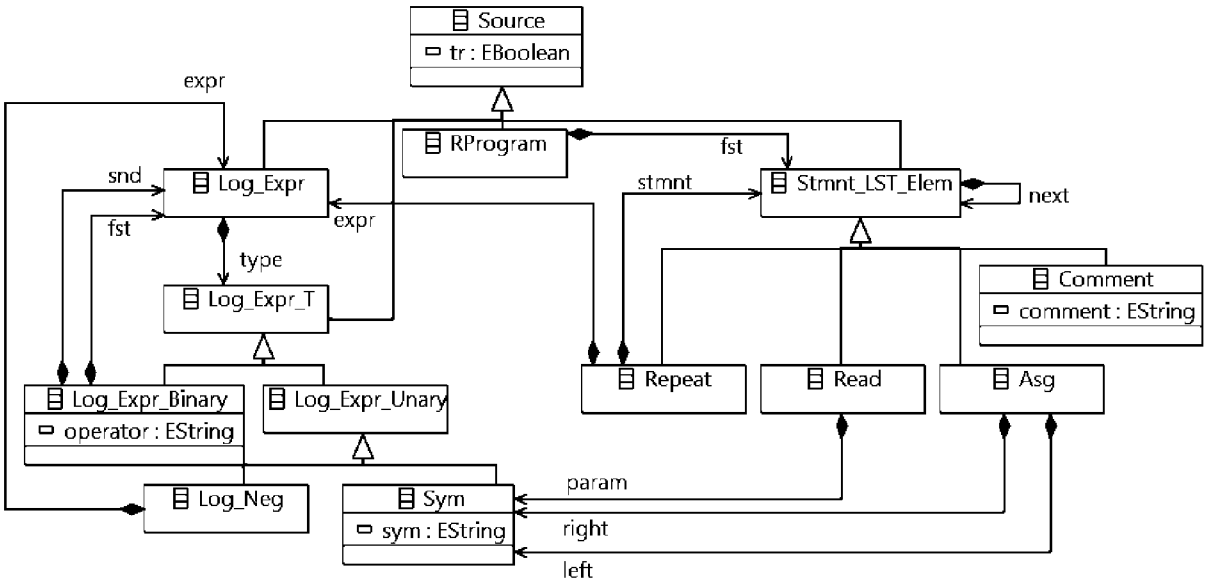

Source
tr : EBoolean
expr
RProgram
fst
snd
Log_Expr
stmnt
Stmnt_LST_Elem
next
fst
expr
type
Log_Expr_T
Comment
comment : EString
Log_Expr_Binary
operator : EString
Log_Expr_Unary
Repeat
Read
Asg
param
Log_Neg
Sym
sym : EString
right
left

Fig. 4

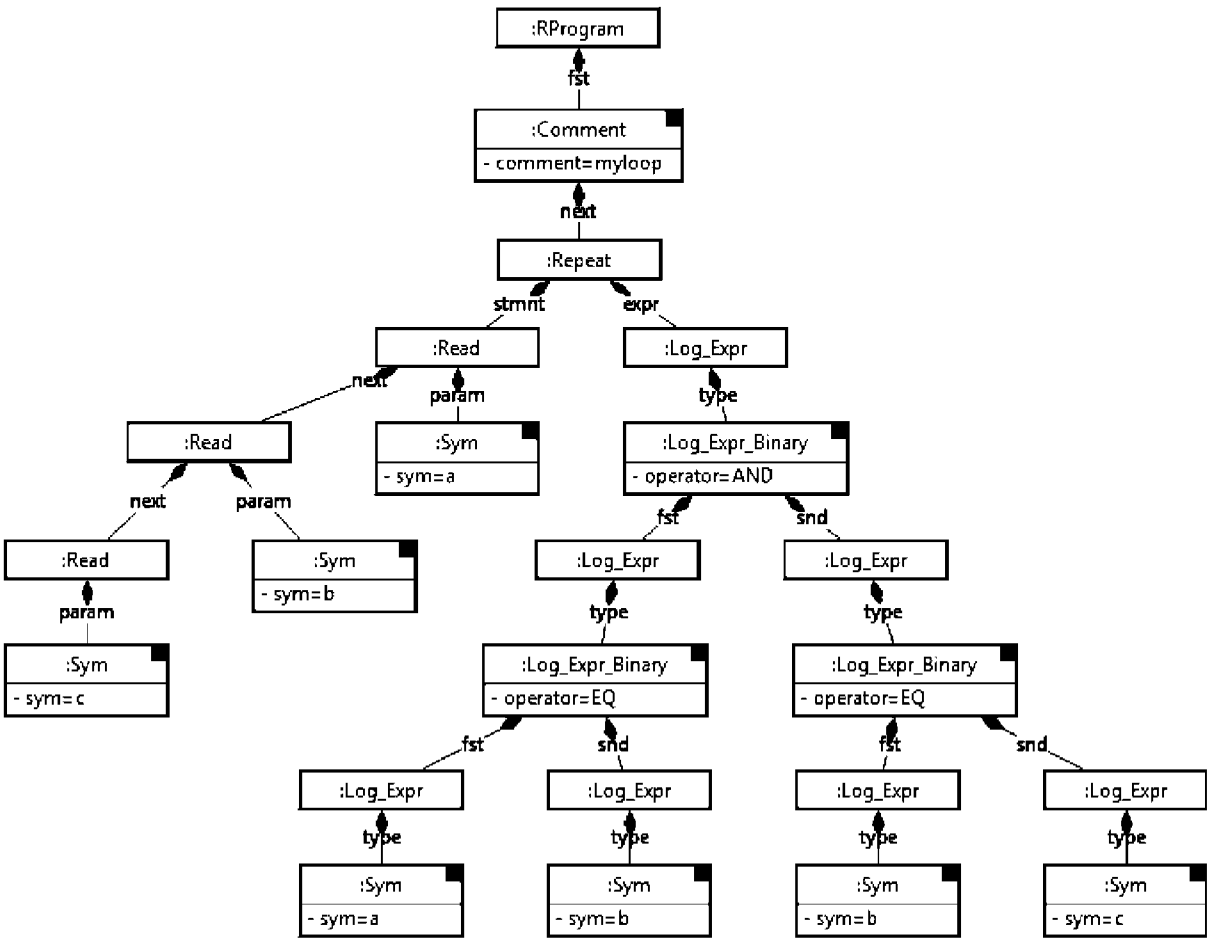

:RProgram
fst
:Comment
- comment=myloop
next
:Repeat
stmnt
expr
:Read
:Log_Expr
next
param
type
:Read
:Sym
- sym=a
:Log_Expr_Binary
- operator=AND
next
param
fst
snd
:Read
:Sym
- sym=b
:Log_Expr
:Log_Expr
param
type
type
:Sym
- sym=c
:Log_Expr_Binary
- operator=EQ
:Log_Expr_Binary
- operator=EQ
fst
snd
fst
snd
:Log_Expr
:Log_Expr
:Log_Expr
:Log_Expr
type
type
type
type
:Sym
- sym=a
:Sym
- sym=b
:Sym
- sym=b
:Sym
- sym=c

Fig. 5

```
1  grammar lu.uni.snt.target.While with org.eclipse.xtext.common.Terminals
2
3  generate target "http://www.uni.lu/snt/target/While"
4
5  WProgram       : fst=Fgmnt_LST_Elem;
6  Fgmnt_LST_Elem : (While | Var_Def | Fn_Call | Fn_Def | Comment) (next=
       Fgmnt_LST_Elem)?;
7  While          : 'while' '(' expr=Expr ')' '{' fgmnt=Fgmnt_LST_Elem '}';
8  Var_Def        : left=Var '=' right=Expr ';';
9  Fn_Call        : nameF=ID '(' ')' ';';
10 Fn_Def         : 'def' nameF=ID '(' ')' '{' body=Fgmnt_LST_Elem '}' ';';
11 Comment        : '/#' comment=STRING '#/';
12 Expr           : type=Expr_T;
13 Expr_T         : Unary | Binary;
14 Unary          : Neg | Var | Input;
15 Binary         : '(' fst=Expr operator=('&&' | '||' | '==') snd=Expr ')';
16 Neg            : '!' expr=Expr;
17 Var            : label=ID;
18 Input          : {Input} 'input' '(' ')';
19
20 Target         : WProgram | Fgmnt_LST_Elem | Expr | Expr_T;
```

Fig. 6

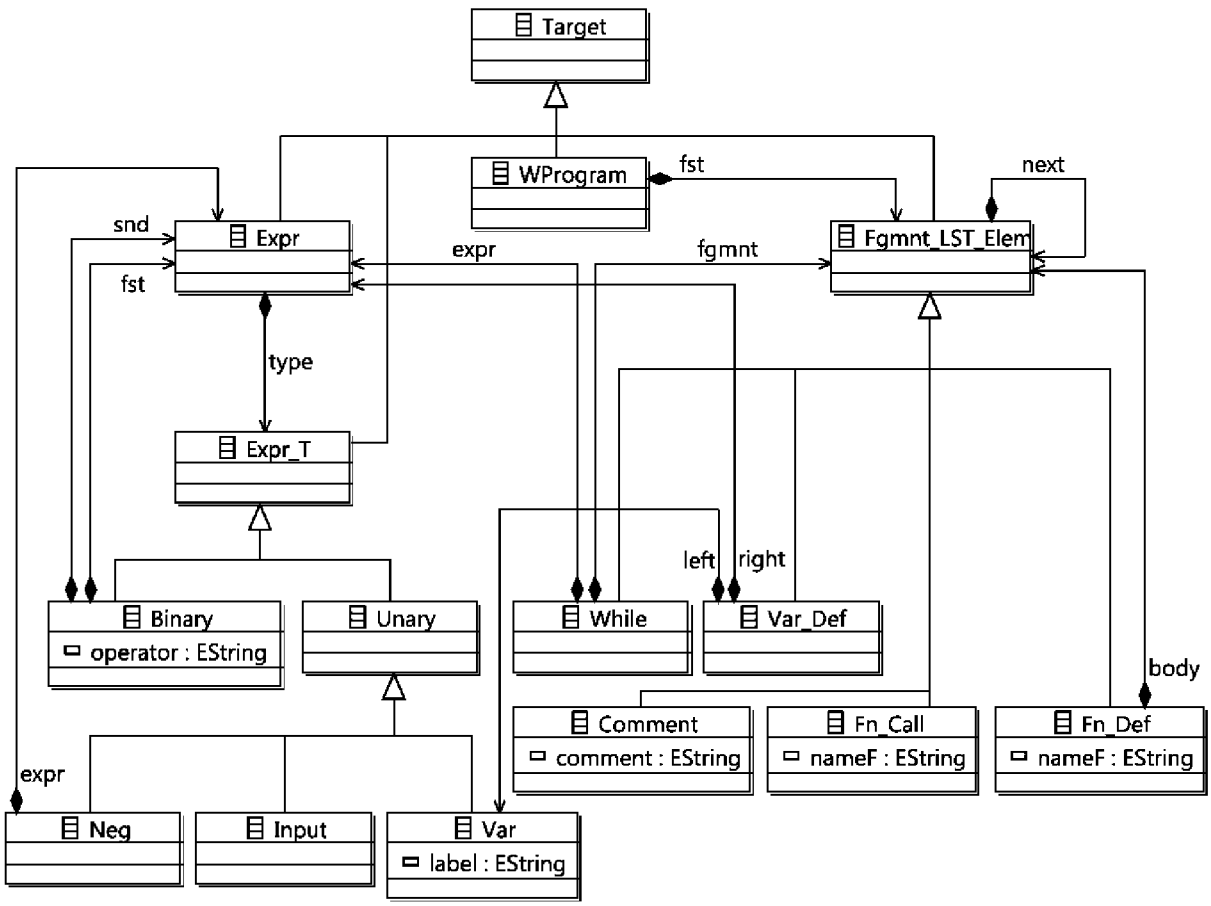


Fig. 7

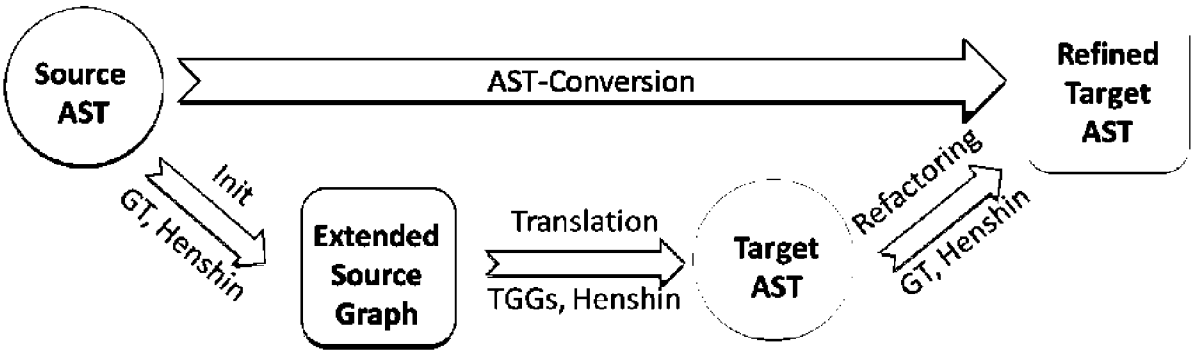

Source AST
AST-Conversion
Refined Target AST
Init
GT, Henshin
Extended Source Graph
Translation
TGGs, Henshin
Target AST
Refactoring
GT, Henshin

Fig. 8

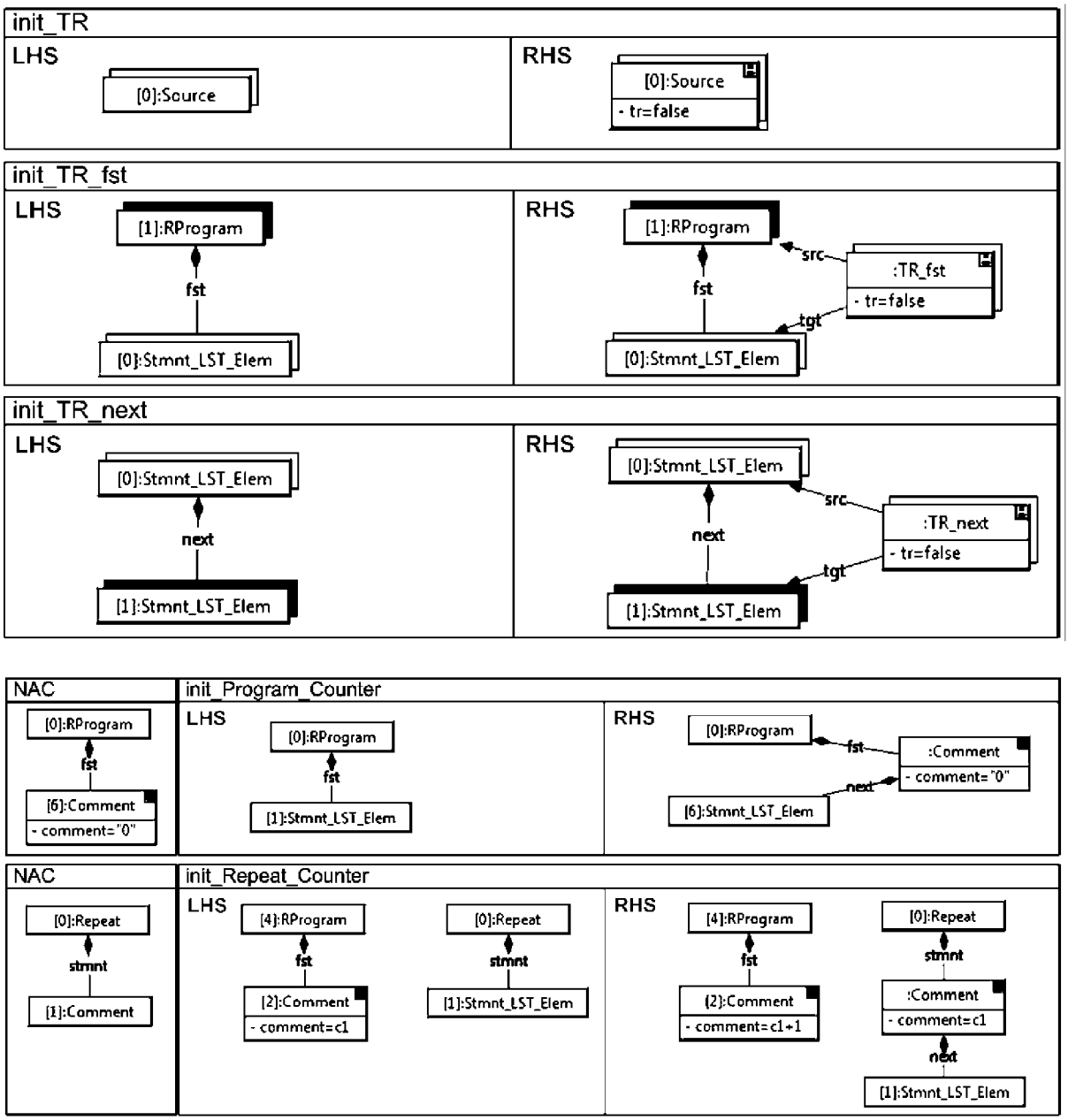

init_TR
LHS
[0]:Source
RHS
[0]:Source
- tr=false
init_TR_fst
LHS
[1]:RProgram
fst
[0]:Stmnt_LST_Elem
RHS
[1]:RProgram
fst
[0]:Stmnt_LST_Elem
src
tgt
:TR_fst
- tr=false
init_TR_next
LHS
[0]:Stmnt_LST_Elem
next
[1]:Stmnt_LST_Elem
RHS
[0]:Stmnt_LST_Elem
next
[1]:Stmnt_LST_Elem
src
tgt
:TR_next
- tr=false
NAC
[0]:RProgram
fst
[6]:Comment
- comment="0"
init_Program_Counter
LHS
[0]:RProgram
fst
[1]:Stmnt_LST_Elem
RHS
[0]:RProgram
fst
:Comment
- comment="0"
next
[6]:Stmnt_LST_Elem
NAC
[0]:Repeat
stmnt
[1]:Comment
init_Repeat_Counter
LHS
[4]:RProgram
fst
[2]:Comment
- comment=c1
[0]:Repeat
stmnt
[1]:Stmnt_LST_Elem
RHS
[4]:RProgram
fst
[2]:Comment
- comment=c1+1
[0]:Repeat
stmnt
:Comment
- comment=c1
next
[1]:Stmnt_LST_Elem

Fig. 9

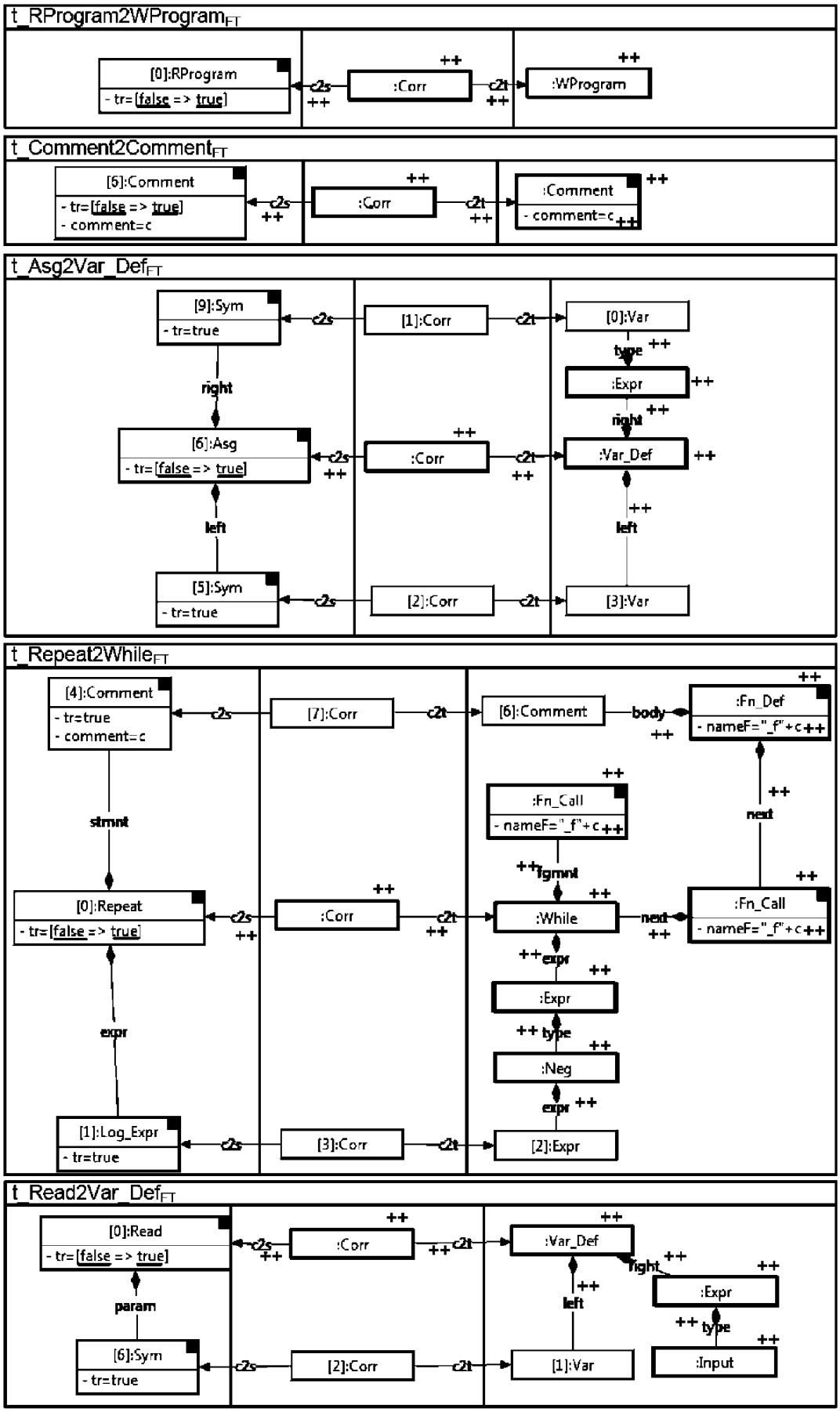

t_RProgram2WProgram_FT
[0]:RProgram
- tr=[false => true]
c2s
:Corr
c2t
:WProgram
t_Comment2Comment_FT
[6]:Comment
- tr=[false => true]
- comment=c
:Corr
:Comment
- comment=c
t_Asg2Var_Def_FT
[9]:Sym
- tr=true
[1]:Corr
[0]:Var
type
:Expr
right
[6]:Asg
- tr=[false => true]
:Corr
:Var_Def
left
[5]:Sym
- tr=true
[2]:Corr
[3]:Var
t_Repeat2While_FT
[4]:Comment
- tr=true
- comment=c
[7]:Corr
[6]:Comment
body
:Fn_Def
- nameF="_f"+c
:Fn_Call
- nameF="_f"+c
next
stmnt
[0]:Repeat
- tr=[false => true]
:Corr
:While
:Fn_Call
- nameF="_f"+c
expr
:Expr
type
:Neg
expr
[1]:Log_Expr
- tr=true
[3]:Corr
[2]:Expr
t_Read2Var_Def_FT
[0]:Read
- tr=[false => true]
:Corr
:Var_Def
right
:Expr
param
left
type
[6]:Sym
- tr=true
[2]:Corr
[1]:Var
:Input

Fig. 10

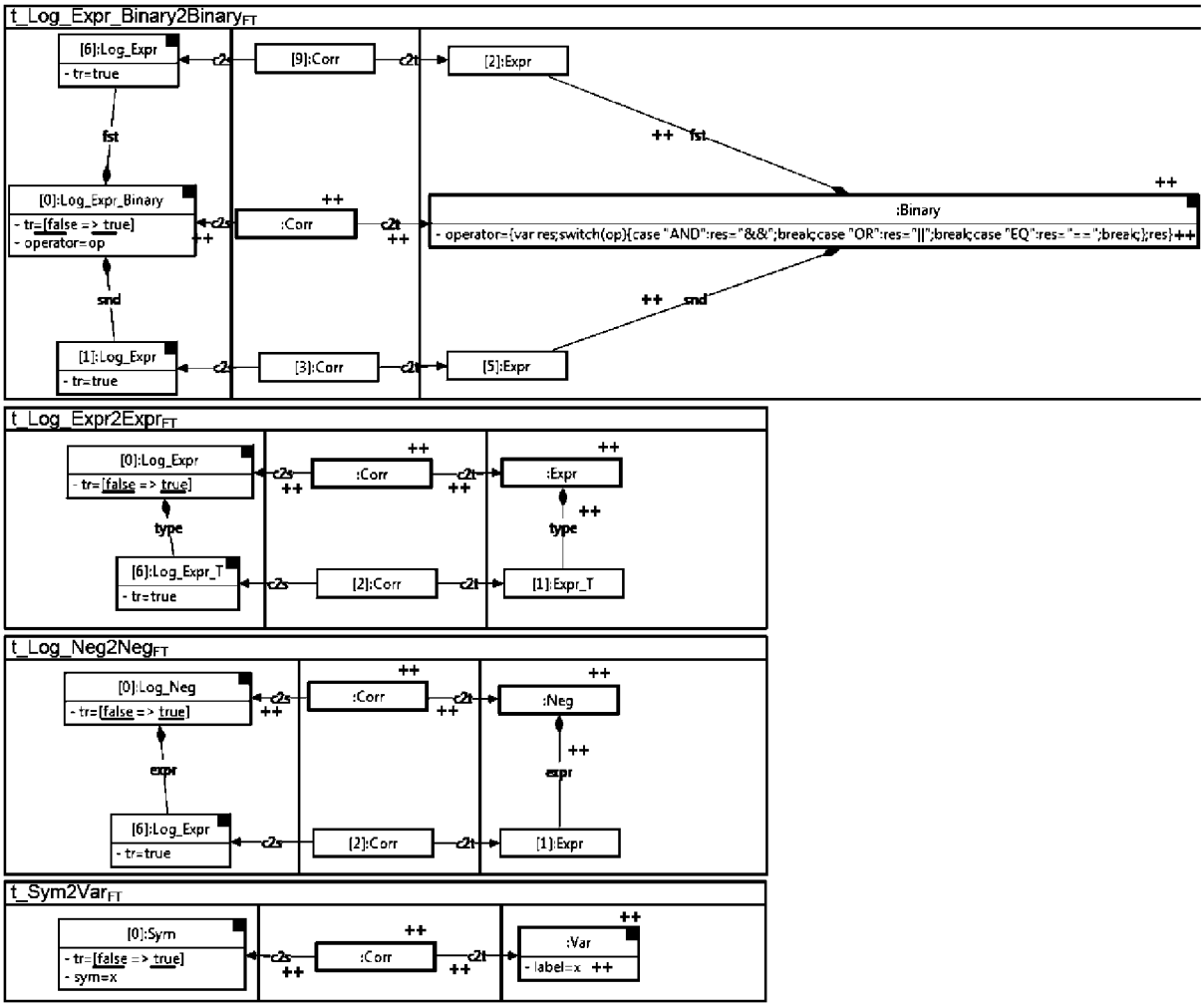

t_Log_Expr_Binary2Binary_FT
[6]:Log_Expr
- tr=true
c2s
[9]:Corr
c2t
[2]:Expr
fst
++ fst
[0]:Log_Expr_Binary
- tr=[false => true]
- operator=op
:Corr
:Binary
- operator={var res;switch(op){case "AND":res="&&";break;case "OR":res="||";break;case "EQ":res="==";break;};res}++
snd
++ snd
[1]:Log_Expr
- tr=true
[3]:Corr
[5]:Expr
t_Log_Expr2Expr_FT
[0]:Log_Expr
- tr=[false => true]
:Corr
:Expr
type
[6]:Log_Expr_T
- tr=true
[2]:Corr
[1]:Expr_T
t_Log_Neg2Neg_FT
[0]:Log_Neg
- tr=[false => true]
:Corr
:Neg
expr
[6]:Log_Expr
- tr=true
[2]:Corr
[1]:Expr
t_Sym2Var_FT
[0]:Sym
- tr=[false => true]
- sym=x
:Corr
:Var
- label=x ++

Fig. 11

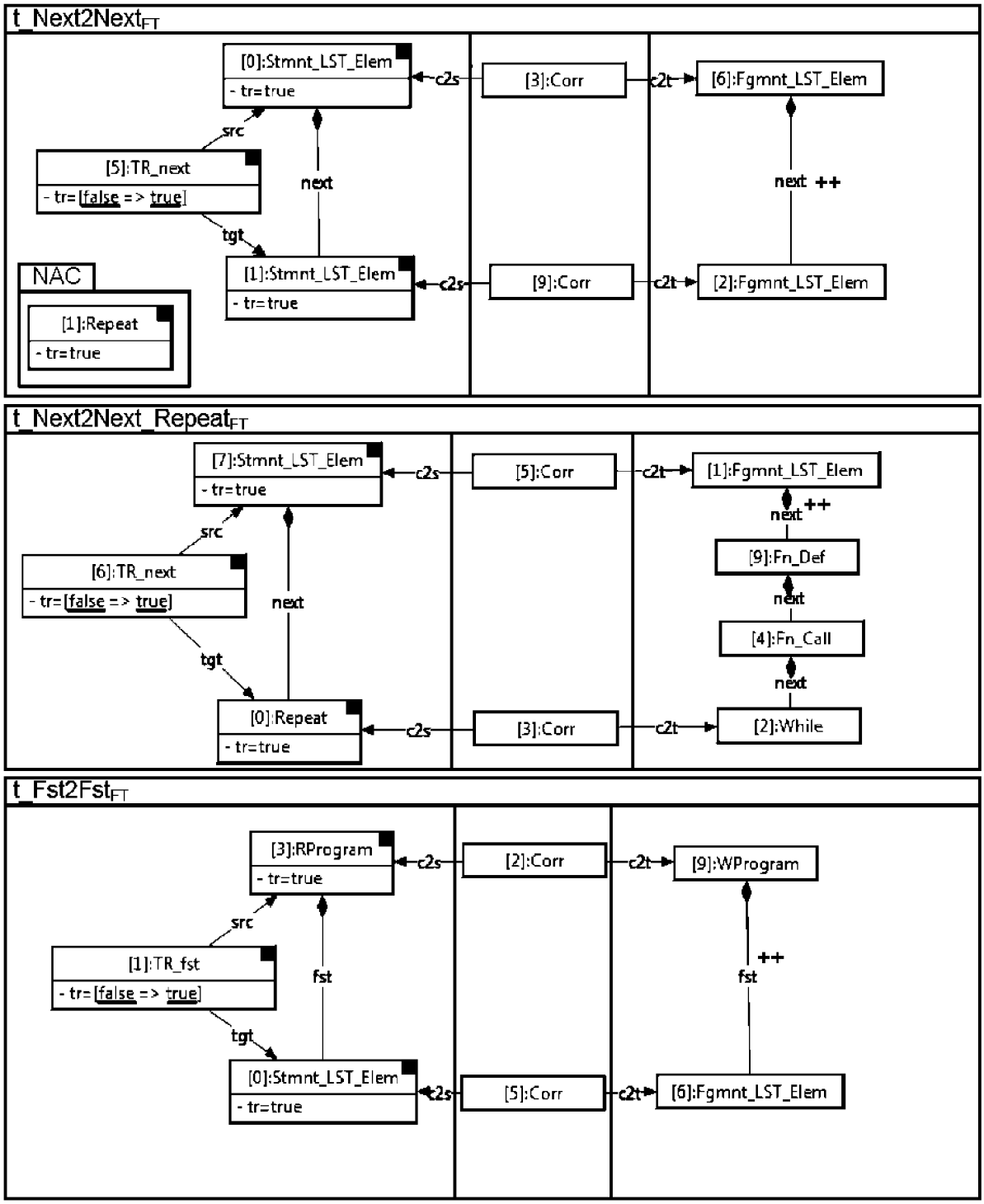

t_Next2Next_FT
[0]:Stmnt_LST_Elem
- tr=true
c2s
[3]:Corr
c2t
[6]:Fgmnt_LST_Elem
src
[5]:TR_next
- tr=[false => true]
next
next ++
tgt
NAC
[1]:Repeat
- tr=true
[1]:Stmnt_LST_Elem
- tr=true
c2s
[9]:Corr
c2t
[2]:Fgmnt_LST_Elem
t_Next2Next_Repeat_FT
[7]:Stmnt_LST_Elem
- tr=true
c2s
[5]:Corr
c2t
[1]:Fgmnt_LST_Elem
next ++
src
[9]:Fn_Def
[6]:TR_next
- tr=[false => true]
next
next
[4]:Fn_Call
tgt
next
[0]:Repeat
- tr=true
c2s
[3]:Corr
c2t
[2]:While
t_Fst2Fst_FT
[3]:RProgram
- tr=true
c2s
[2]:Corr
c2t
[9]:WProgram
src
[1]:TR_fst
- tr=[false => true]
fst
++
fst
tgt
[0]:Stmnt_LST_Elem
- tr=true
c2s
[5]:Corr
c2t
[6]:Fgmnt_LST_Elem

Fig. 12

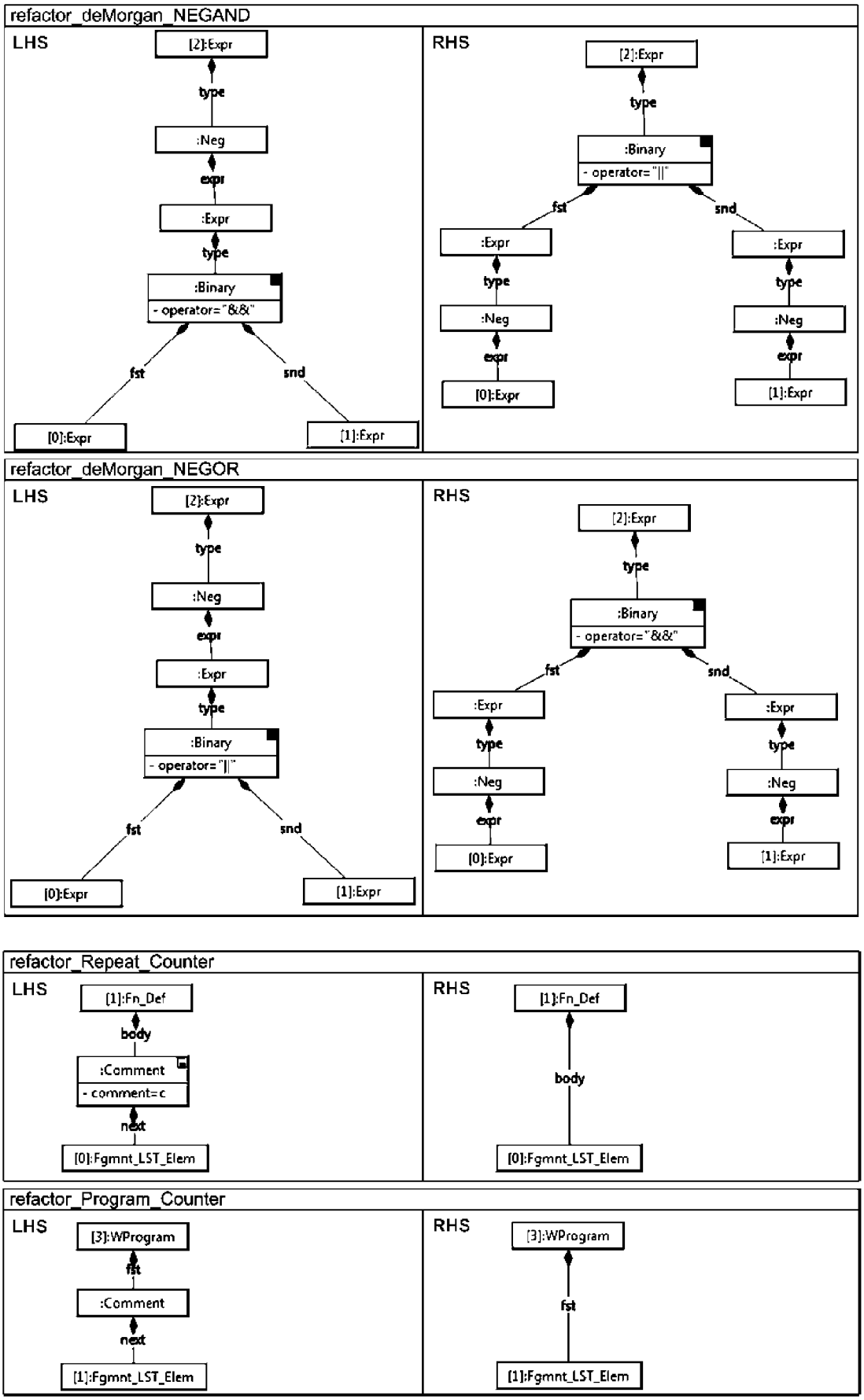
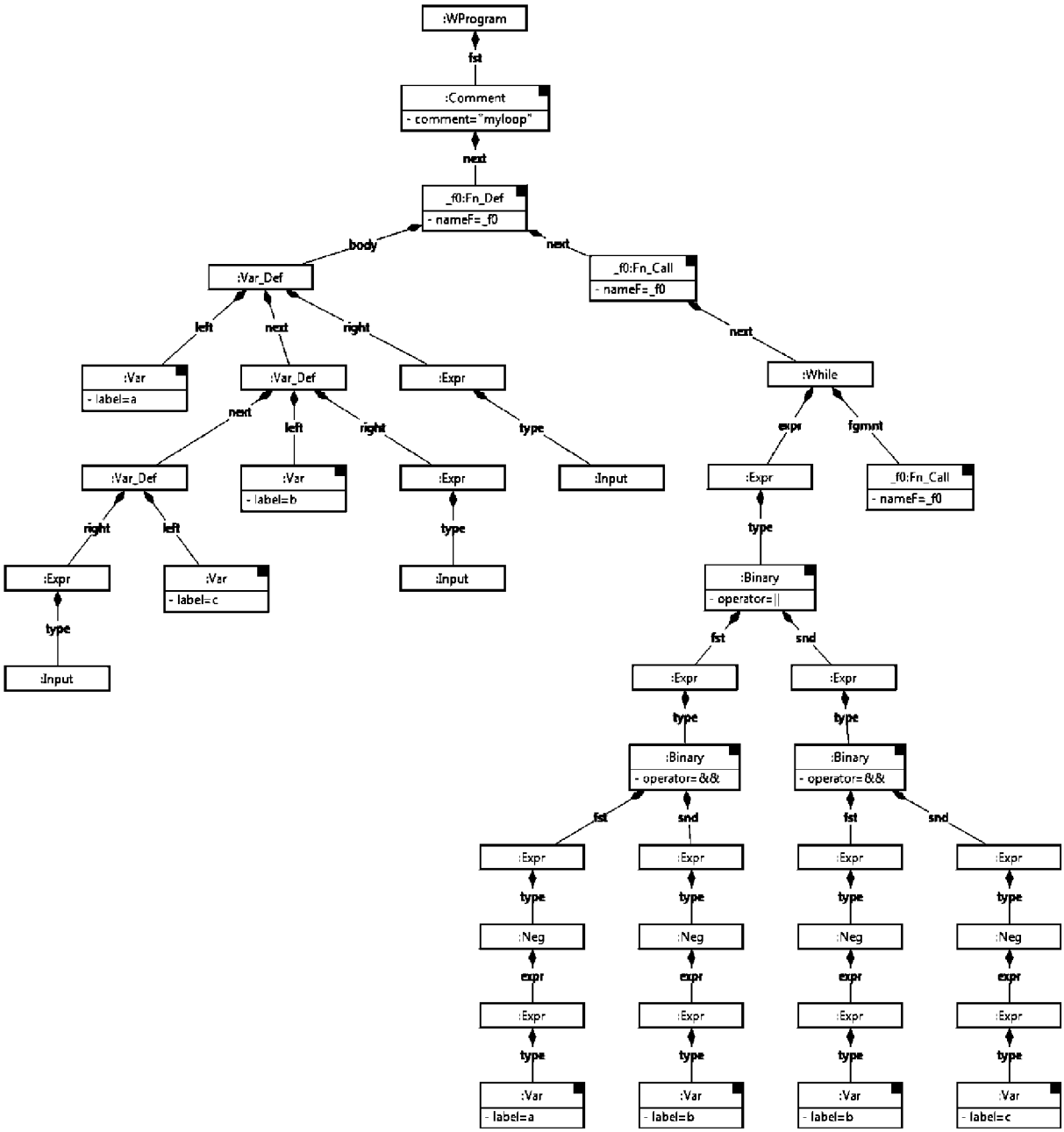

refactor_deMorgan_NEGAND
LHS
[2]:Expr
type
:Neg
expr
:Expr
type
:Binary
- operator="&&"
fst
snd
[0]:Expr
[1]:Expr
RHS
[2]:Expr
type
:Binary
- operator="||"
fst
snd
:Expr
:Expr
type
type
:Neg
:Neg
expr
expr
[0]:Expr
[1]:Expr
refactor_deMorgan_NEGOR
LHS
[2]:Expr
type
:Neg
expr
:Expr
type
:Binary
- operator="||"
fst
snd
[0]:Expr
[1]:Expr
RHS
[2]:Expr
type
:Binary
- operator="&&"
fst
snd
:Expr
:Expr
type
type
:Neg
:Neg
expr
expr
[0]:Expr
[1]:Expr
refactor_Repeat_Counter
LHS
[1]:Fn_Def
body
:Comment
- comment=c
next
[0]:Fgmnt_LST_Elem
RHS
[1]:Fn_Def
body
[0]:Fgmnt_LST_Elem
refactor_Program_Counter
LHS
[3]:WProgram
fst
:Comment
next
[1]:Fgmnt_LST_Elem
RHS
[3]:WProgram
fst
[1]:Fgmnt_LST_Elem

Fig. 13


:WProgram
fst
:Comment
- comment="myloop"
next
_f0:Fn_Def
- nameF=_f0
body
next
:Var_Def
_f0:Fn_Call
- nameF=_f0
left
next
right
next
:Var
- label=a
:Var_Def
:Expr
:While
next
left
right
type
expr
fgmnt
:Var_Def
:Var
- label=b
:Expr
:Input
:Expr
_f0:Fn_Call
- nameF=_f0
right
left
type
type
:Expr
:Var
- label=c
:Input
:Binary
- operator=||
type
fst
snd
:Input
:Expr
:Expr
type
type
:Binary
- operator=&&
:Binary
- operator=&&
fst
snd
fst
snd
:Expr
:Expr
:Expr
:Expr
type
type
type
type
:Neg
:Neg
:Neg
:Neg
expr
expr
expr
expr
:Expr
:Expr
:Expr
:Expr
type
type
type
type
:Var
- label=a
:Var
- label=b
:Var
- label=b
:Var
- label=c

INTERNATIONAL SEARCH REPORT

International application No
PCT/EP2013/062411

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F9/45
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

EPO-Internal, INSPEC, COMPENDEX

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	WO 2008/108665 A1 (INNAWORKS DEV LTD [NZ]; CHENG STEPHEN MING KO [NZ]; POTANIN ALEX [NZ];) 12 September 2008 (2008-09-12) abstract page 18, line 9 - page 24, line 10 ----- -/--	1-14

[X] Further documents are listed in the continuation of Box C.

[X] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
25 June 2013	02/07/2013

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	de Man, Ronald

2

Form PCT/ISA/210 (second sheet) (April 2005)

INTERNATIONAL SEARCH REPORT

International application No
PCT/EP2013/062411

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	Striewe M et al: "Using a Triple Graph Grammar for State Machine Implementations", TU Berlin Proceedings of the Doctoral Symposium at the International Conference on Graph Transofmration (ICGT 2008), vol. 16, no. 8 10 August 2011 (2011-08-10), pages 1-15, XP002686148, ISSN: 1863-2122 Retrieved from the Internet: URL:http://opus.kobv.de/tuberlin/volltexte/2011/3120/pdf/16_8.pdf [retrieved on 2012-10-29] abstract page 2, paragraph 2 page 3, paragraph 4 page 5, paragraph 3 - page 11, paragraph 2 -----	1-14
L	Striewe, Michael ; Goedicke, Michael: "Eingang zum Volltext in OPUS - Using a Triple Graph Grammar for State Machine Implementations", TU Berlin Internet citation, XP002686149, Retrieved from the Internet: URL:http://opus.kobv.de/tuberlin/frontdoor.php?source_opus=3120&la=de [retrieved on 2012-10-29] the whole document -----	

2

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/EP2013/062411

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 2008108665 A1	12-09-2008	EP 2122464 A1	25-11-2009
		WO 2008108665 A1	12-09-2008