From Core OCL Invariants to Nested Graph Constraints*

Thorsten Arendt[1], Annegret Habel[2], Hendrik Radke[2], and Gabriele Taentzer[1]

[1] Philipps-Universität Marburg, Germany
{arendt,taentzer}@informatik.uni-marburg.de
[2] Universität Oldenburg, Germany
{habel,radke}@informatik.uni-oldenburg.de

Abstract. Meta-modeling including the use of the Object Constraint Language (OCL) forms a well-established approach to design domain-specific modeling languages. This approach is purely declarative in the sense that instance construction is not needed and not considered. In contrast, graph grammars allow the stepwise construction of instances by the application of transformation rules. In this paper, we consider meta-models with Core OCL invariants and translate them to nested graph constraints for typed attributed graphs. Models and meta-models are translated to instance and type graphs. We show that a model satisfies a Core OCL invariant iff its corresponding instance graph satisfies the corresponding nested graph constraint. The aim of this work is to establish a first formal relation between meta-modeling and the theory of graph transformation including constraints to come up with an integrated approach for defining modeling languages in an optimal way in the future.

Keywords: Meta modeling, OCL, graph constraints, application conditions.

1 Introduction

The trend towards model-based and model-driven software development causes a need of new, mostly domain-specific modeling languages with well-designed tool support. Therefore we need methods and techniques to define modeling languages and their tooling precisely and also intuitively. A comprehensive language definition needs the declarative as well as the constructive paradigm to specify language properties, to construct and recognize language instances as well as to modify them. Nowadays, modeling languages are typically defined by meta-models following purely the declarative approach. In this approach, language properties are specified by the Object Constraint Language (OCL) [1].

* This work is partly supported by the German Research Foundation (DFG), Grant HA 2936/4-1 (Meta modeling and graph grammars: integration of two paradigms for the definition of visual modeling languages).

H. Giese and B. König (Eds.): ICGT 2014, LNCS 8571, pp. 97–112, 2014.

In contrast, graph grammars have shown to be suitable and natural to specify visual languages in a constructive way, by using graph transformation [2]. Recently, nested graph constraints [3] have been developed to include also the declarative element into graph grammars. To ensure that a graph grammar fulfills a set of graph constraints, they can be translated to application conditions of graph rules such that all graphs fulfilling the constraints in the beginning keep on fulfilling them after applying graph rules being extended by translated application conditions.

While typed attributed graphs form an adequate formalization of instance models that are typed over a meta-model [4], the relation of OCL constraints to nested graph constraints has not been considered yet. We are interested in investigating this relation, since the translation of graph constraints to application conditions for rules opens up a way to combine declarative and constructive elements in a formal approach. By translating OCL to nested graph constraints, such an integration of declarative and constructive elements becomes possible also in the meta-modeling approach. It shall open up a way to translate OCL constraints to application conditions of model transformation rules making applications as e.g. auto-completion of model editing operations to consistent models possible.

As a basis, models and meta-models (without OCL constraints) are translated to instance and type graphs. In this paper, we investigate the relation of meta-models including OCL constraints and nested graph constraints for typed attributed graphs. It turns out that Core OCL invariants [5], i.e. Boolean expressions over navigations based on the type system, can be well translated to nested graph constraints. The aim of this work is to establish a first formal relation between meta-modeling and the theory of graph transformation to come up with an integrated approach for defining modeling languages in an optimal way in the future.

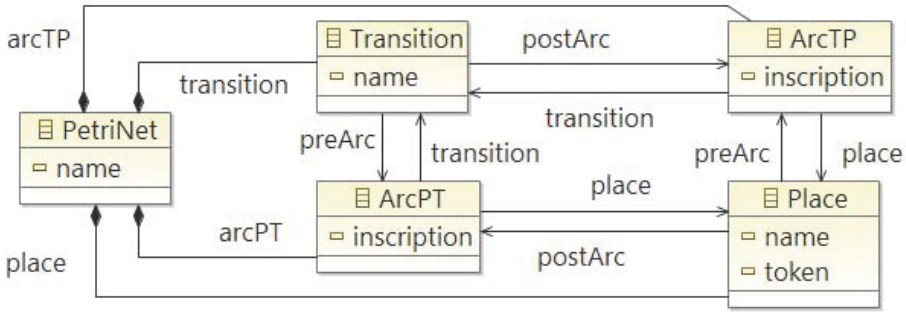
This paper is structured as follows: The next section presents OCL in a nutshell focusing on Core OCL invariants. Section 3 shows typed attributed graphs and graph morphisms as well as nested graph conditions. Section 4 presents our main contribution of this paper, the translation of Core OCL invariants to nested graph constraints. Section 5 discusses how Core OCL invariants can be translated to equivalent application conditions of graph rules. Section 6 compares to related work and concludes the paper.

2 Core OCL Invariants

In this section, we recall Core OCL constraints presenting a small example first and formally defining their syntax and semantics thereafter, according to the work by Richters [6] that went into the OCL specification by the OMG [1]. For illustration purposes, we use the following meta-model for simple Petri nets to recall OCL.

Example 1. A Petri net (*PetriNet*) is composed of places (*Place*) or transitions (*Transition*) which are linked together by arcs (*ArcTP* for linking exactly one

transition to one place; *ArcPT* for linking exactly one place to one transition). Places and transitions can have an arbitrary number of incoming (*preArc*) and outgoing (*postArc*) arcs. Finally, Petri net markings are defined by the *token* attribute of places. However, this meta-model allows to build invalid models. For example, one can model a transition having no incoming arc, i.e., the transition can never be fired. Therefore, we complement the meta-model with invariants formulated in OCL.



1. A transition has incoming arcs.
   `context Transition inv: self.preArc -> notEmpty()`
2. The number of tokens on a place is not negative.
   `context Place inv: self.token >= 0`
3. Each two places of a Petri net have different names.
   `context Petrinet inv: self.place -> forAll(p1:Place | self.place -> forAll(p2:Place | p1 <> p2 implies p1.name <> p2.name))` or alternatively
   `context Petrinet inv: self.place -> forAll(p1:Place,p2:Place | p1 <> p2 implies p1.name <> p2.name)`

Now, we consider Core OCL invariants in more detail. The `Core OCL` comprises the OCL type system and the language concepts that realize model navigation. The only kind of collections we consider are sets which conform well with using OCL for meta-modeling. Furthermore, we concentrate on selected Boolean-typed set operations only (*isEmpty*, *notEmpty*, *exists*, and *forAll*). This also means that user-defined operations are not allowed.

For Core OCL, we straiten the kind of object models being allowed: attributes have primitive types only, there are no operations defined, associations are binary, roles are the default ones indicating source and target, and multiplicities are not set, i.e. range between 0 and *. (It is obvious, however, that multiplicities can be expressed by Core OCL invariants.)

Definition 1 (Core Object Model). Let $DSIG = (S, OP)$ be a data signature with $S = \{Integer, Real, Boolean, String\}$ and a family of corresponding operation symbols OP. A *core object model* over $DSIG$ is a structure $M = (CLASS, ATT, ASSOC, associates, r_{src}, r_{tgt}, \prec)$ where $CLASS$ is a finite

set of classes, $ATT = \{ATT_c\}_{c \in CLASS}$ is a family of attributes $att : c \to S$ of class c, $ASSOC$ is a set of associations, $associates$ is a function that maps each association to a pair of participating classes with $associates : ASSOC \to (CLASS \times CLASS)$, r_{src} and r_{tgt} are functions that map each association to a source respectively target role name with $r_{src}, r_{tgt} : ASSOC \to String$ and $r_{src}(assoc) = c_1$ and $r_{tgt}(assoc) = c_2$ for each $assoc \in ASSOC$ with $associates(assoc) = (c_1, c_2)$, and $\prec$ is a partial order on $CLASS$ reflecting its generalization hierarchy.

Since the evaluation of an OCL invariant requires knowledge about the complete context of an object model at a discrete point in time, we define a *system state* of a core object model M. Informally, a system state consists of a set of class objects, functions assigning attribute values to each class object for each attribute, and a finite set of links connecting class objects within the model.

Definition 2 (System State). A *system state* of a core object model M is a structure $\sigma(M) = (\sigma_{CLASS}, \sigma_{ATT}, \sigma_{ASSOC})$ where for each class $c \in CLASS$, $\sigma_{CLASS}(c)$ is a finite subset of the (infinite) set of object identifiers $oid(c) = \{\underline{c}_1, \underline{c}_2, \dots\}$, for each attribute $att : c \to t \in ATT_c^{\prec}$, $\sigma_{ATT}(att) : \sigma_{CLASS}(c) \to I(t)$ is an operation from class objects to some interpretation of the primitive data type t where $ATT_c^{\prec} := \bigcup_{c \prec c'} ATT_{c'}$ is the set of all owned and inherited attribute symbols of a class c, for each $assoc \in ASSOC$ with $associates(assoc) = (c_1, c_2)$, $\sigma_{ASSOC}(assoc) \subset \sigma_{CLASS}^{\prec}(c_1) \times \sigma_{CLASS}^{\prec}(c_2)$ where $\sigma_{CLASS}^{\prec}(c) := \bigcup_{c' \prec c} \sigma_{CLASS}(c')$ is the set of all objects with type or super type c. The set $States(M)$ consists of all system states $\sigma(M)$ of M .

Definition 3 (Core OCL Expressions). Let T be a set of types consisting of all basic types S, all class types $CLASS$, and the collection type `Set(t)` for an arbitrary $t \in T$. Let Ω be a set of operations on T consisting of OP, ATT, appropriate association end operations, and set operations. Let $Var = \{Var_t\}_{t \in T}$ be a family of variable sets. The family of *Core OCL expressions* over T and Ω is given by $Expr = \{Expr_t\}_{t \in T}$ of sets of expressions. An expression in $Expr$ is a *VariableExpression* $v \in Expr_t$ for all variables $v \in Var_t$, *OperationExpressions* $e := \omega(e_1, \cdots, e_n) \in Expr_t$ for each operation symbol $\omega : t_1 \times \cdots \times t_n \to t \in \Omega$ and for all $e_i \in Expr_{t_i} (1 \leq i \leq n)$, *IfExpressions* $e :=$ `if` e_1 `then` e_2 `else` e_3 $\in Expr_{Boolean}$ for all $e_1, e_2, e_3 \in Expr_{Boolean}$ and *IteratorExpressions* $e := s \to exists(v \mid b) \in Expr_{Boolean}$ and $e := s \to forAll(v \mid b) \in Expr_{Boolean}$, for all $s \in Expr_{Set(c)}, v \in Var_c$, and $b \in Expr_{Boolean}$.
Let $Env = \{\tau \mid \tau = (\sigma, \beta)\}$ be a set of environments with system states σ and variable assignments $\beta : Var_t \to I(t)$ which map variable names to values. The semantics of a Core OCL expression $e \in Expr_t$ is a function $I[\![e]\!] : Env \to I(t)$ for $t \in CLASS$ or $t \in S$. The corresponding semantics definition can be found in [6] and adapted to Core OCL in [7].

As mentioned above, we concentrate on invariants being formulated in Core OCL. Therefore, we consider invariants and OCL-constraints as synonyms in the remainder of this paper.

Definition 4 (Core OCL Invariant). A *Core OCL invariant* is a Boolean Core OCL expression with a free variable $v \in Var_C$ where C is a classifier type. The concrete syntax of an invariant is: `context v:C inv : <expr>`. The set $Invariant_M$ denotes the set of all Core OCL invariants over M.

Remark 1. The following properties hold for Core OCL invariants: (1) Navigation expressions to collections are not contained in other navigation expressions, e.g., `somePetriNet.arcTP.place -> notEmpty()` is replaced by `somePetriNet.arcTP -> exists(a:ArcTP | a.place -> notEmpty())`. (2) Iterator expressions are completed, i.e. the iterator variable is explicitly declared. Moreover, a variable declaration is always complete, i.e. consists of a variable name and a type name. (3) If `nav op nav` occurs for the same navigation expression nav and $op(nav, nav) = true$ then `nav op nav` can be replaced by `true`. (4) Note that constraints `v1 = v2.r` and `v1 <> v2.r` (for objects `v1`,`v2` and reference `r`) are not possible since the result of `v1` is an object and `v2.r` yields a set of objects.

3 Nested Graph Conditions

In the following, we recall the formal definition of typed, attributed graphs as presented in [8]. They form the basis to define attributed graph conditions. Attributed graphs as defined here allow to attribute nodes only while the original version [8] supports also the attribution of edges.

Definition 5 (A-graphs). An *A-graph* $G = (G_V, G_D, G_E, G_A, src_G, tgt_G, src_A, tgt_A)$ consists of sets G_V and G_D, called graph and data nodes (or vertices), respectively, G_E and G_A, called graph and node attribute edges, respectively, and source and target functions: $src_G \colon G_E \to G_V, tgt_G \colon G_E \to G_V$ for graph edges and $src_A \colon G_A \to G_V, tgt_A \colon G_A \to G_D$ for node attribute edges. Given two A-graphs G^1 and G^2, an *A-graph morphism* $f \colon G^1 \to G^2$ is a tuple of functions $f_V \colon G^1_V \to G^2_V, f_D \colon G^1_D \to G^2_D, f_E \colon G^1_E \to G^2_E$ and $f_A \colon G^1_A \to G^2_A$ such that f commutes with all source and target functions, e.g. $f_V \circ src^1_G = src^2_G \circ f_E$.

We assume that the reader is familiar with the basics of algebraic specification. The definition of attributed graphs generalizes largely the one in [9] by allowing variables and a set of formulas that constrain the possible values of these variables. The definition is closely related to symbolic graphs [10].

Definition 6 (Attributed graphs). Let $DSIG = (S, OP)$ be a data signature, $X = \{X_s\}_{s \in S}$ a family of variables, and $T_{DSIG}(X)$ the term algebra w.r.t. $DSIG$ and X. An *attributed graph* over $DSIG$ and X is a tuple $AG = (G, D, \Phi)$ where G is an A-graph, D is a $DSIG$-algebra with $\sum_{s \in S} D_s = G_D$, and Φ is a finite set of $DSIG$-formulas[1] with free variables in X. A set $\{F_1, \ldots, F_n\}$ of formulas can be regarded as a single formula $F_1 \wedge \ldots \wedge F_n$. An attributed graph $AG = (G, D, \emptyset)$ with an empty set of formulas is *basic* and is shortly denoted by $AG = (G, D)$.

[1] $DSIG$-formulas are meant to be $DSIG$-terms of sort BOOL. One may consider e.g. a set of literals.

Given two attributed graphs AG^1 and AG^2, an *attributed graph morphism* $f\colon AG^1 \to AG^2$ is a pair $f = (f_G, f_D)$ of an A-graph morphism $f_G\colon G^1 \to G^2$ and a $DSIG$-homomorphism $f_D\colon D^1 \to D^2$ such that for all $s \in S$, $f_{G,G_D}|_{D^1_s} = f_{D,s}$, $f_{G,G_D} = \sum_{s\in S} f_{D,s}$, and $\Phi^2 \Rightarrow f(\Phi^1)$ where $f(\Phi^1)$ is the set of formulas obtained when replacing in Φ^1 every variable x in G^1 by $f(x)$.

Remark 2. We are interested in the case where D^1_s is a $DSIG$-term algebra and D^2_s is a $DSIG$-algebra (without variables). In this case the $DSIG$-homomorphism assigns values to variables and terms.

Attributed graphs in the sense of [9] correspond to basic attributed graphs. The results for basic attributed graphs can be generalized to arbitrary attributed graphs: attributed graphs and morphisms form the category **AGraphs**. The category has pushouts and $\mathcal{E}'$-$\mathcal{M}$ pair factorization in the sense of [9].

Definition 7 (Typed attributed graph over $ATGI$). An *attributed type graph* $ATGI = (TG, Z, \Phi')$ consists of an A-graph and a final $DSIG$-algebra Z. A *typed attributed graph* $(AG, type)$ over $ATGI$, short *$ATGI$-graph*, consists of an attributed graph $AG = (G, D, \Phi)$ and a *morphism type* : $AG \to ATGI$[2]. Given two $ATGI$-graphs $AG^1 = (G^1, type^1)$ and $AG^2 = (G^2, type^2)$, an *$ATGI$-morphism* $f\colon AG^1 \to AG^2$ is an attributed graph morphism such that $type^2 \circ f = type^1$.

Typed attributed graphs and morphisms form a category that has pushouts and $\mathcal{E}'$-$\mathcal{M}$ pair factorization.

Fact 1 ([7]). $ATGI$-graphs and morphisms form the category $\mathbf{AGraphs}_{ATGI}$ with pushouts and $\mathcal{E}'$-$\mathcal{M}$ pair factorization.

In [7], also typed attributed graphs typed over attributed type graphs with inheritance [11] are considered.
Nested graph conditions [3] are nested constructs which can be represented as trees of morphisms equipped with quantifiers and Boolean connectives. In the following, we introduce $ATGI$-conditions as conditions over $ATGI$-graphs, closely related to attributed graph constraints [10] and E-conditions [12].

Definition 8 (nested graph conditions). A *(nested) graph condition* on typed attributed graphs, short *$ATGI$-condition*, over a graph P is of the form *true*, $\exists(a, c)$, or $\exists(P \sqsupseteq C, c)$[3] where $a\colon P \to C$ is an injective morphism and c is an $ATGI$-condition over C. Boolean formulas over $ATGI$-conditions over P yield $ATGI$-conditions over P, that is $\neg c$ and $\bigwedge_{i\in I} c_i$ are $ATGI$-conditions over P. Conditions over $\emptyset$ are also called *constraints*.

Notation. Graph conditions may be written in a more compact form: $\exists a$ abbreviates $\exists(a, true)$, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$, $\vee_{i\in I} c_i$ abbreviates $\neg \wedge_{i\in I} \neg c_i$,

[2] We usually set $\Phi' =$ false for $ATGI$ so that $\Phi' \Rightarrow \Phi$ is true regardless of Φ.
[3] Conditions of the form $\exists(P \sqsupseteq C, c)$ are syntactic sugar, i.e. they can be expressed in terms of the other constructs. See long version [7].

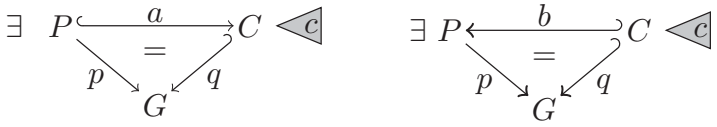
and $c \Rightarrow c'$ abbreviates $\neg c \vee c'$. For an injective morphism $a\colon P \hookrightarrow C$ in a condition, we just depict the codomain C, if the domain P can be unambiguously inferred, i.e. if it is known over which graph a condition is.

Example 2 (OCL constraints as graph constraints). OCL constraint `context Place inv: self.token >= 0` in Example 1 is represented as an attributed graph constraint in full and in abbreviated form. The last graph in the condition is decorated by the formula $x \geq 0$, with the notation as in [12]. The attributing DSIG-algebra is the quotient term algebra $T_{\mathrm{DSIG}_{\equiv}}(X)$ where $\equiv$ is the congruence relation on $T_{DSIG}(X)$ induced by $\geq (x, 0)$.

$$\neg\exists\left(\emptyset \to \boxed{\text{self:Place}}, \neg\exists\left(\boxed{\text{self:Place}} \to \boxed{\begin{array}{c}\text{self:Place}\\ \hline \text{token = x}\end{array}} \mid x \geq 0\right)\right)$$

$$\text{or, in short form: } \forall\left(\boxed{\text{self:Place}}, \exists\ \boxed{\begin{array}{c}\text{self:Place}\\ \hline \text{token} \geq 0\end{array}}\right)$$

Definition 9 (Semantics of nested graph conditions). Let $p\colon P \to G$ be a morphism. *Satisfiability* of a condition over P is inductively defined as follows: Every morphism satisfies *true*. Morphism p satisfies $\exists(P \to C, c)$ if there exists an injective morphism $q\colon C \hookrightarrow G$ such that the left diagram below commutes and q satisfies c. Morphism p satisfies $\exists(P \sqsupseteq C, c)$ if there exist injective morphisms $b\colon C \hookrightarrow P$ and $q\colon C \hookrightarrow G$ such that $q = p \circ b$ and q satisfies c (see right diagram below). Morphism p satisfies $\neg c$ if p does not satisfy c, and p satisfies $\bigwedge_{i\in I} c_i$ if p satisfies each c_i $(i \in I)$. We write $p \models c$ if $p\colon P \to G$ satisfies the condition c over P.



Satisfiability of a constraint (i.e. a condition over $\emptyset$) by a graph is defined as follows: A graph G satisfies a constraint c, short $G \models c$, if the morphism $p\colon \emptyset \hookrightarrow G$ satisfies c.

4 Translation of Meta-Models with Core OCL Invariants

To translate Core OCL invariants, we first show how to translate the type information of meta-models, i.e. core object models, to attributed type graphs with inheritance [11] are considered. Thereafter, system states are translated to typed attributed graphs. Having these ingredients available, our main contribution, the translation of Core OCL invariants is presented, together with two example translations. Finally, completeness and correctness of the translation are shown.

4.1 Type and State Correspondences

To define the translation of Core OCL invariants to graph constraints, we translate a given object model to its corresponding type graph.

Definition 10 (Type Correspondence). Let $DSIG = (S, OP)$ be a data signature and Z the final DSIG-algebra. Given a core object model $M = (CLASS, ATT, ASSOC, associates, \prec)$ over $DSIG$, it corresponds to an attributed type graph with inheritance $ATGI = ((TG, Z), Inh)$ with type graph $TG = (TG_V, TG_D, TG_E, TG_A, src_G, tgt_G, src_A, tgt_A)$ and inheritance relation Inh if there is a *correspondence relation* $corr_{type} = (corr_{CLASS}, corr_{ATT}, corr_{ASSOC})$ with bijective mappings

- $corr_{CLASS} : CLASS \to TG_V$ with $\forall c1, c2 \in CLASS$:
  $c_1 \prec c_2 \iff (corr_{CLASS}(c_1), corr_{CLASS}(c_2)) \in Inh$,
- $corr_{ATT} : ATT \to TG_A$ with
  $src_A(corr_{ATT}(att)) = corr_{CLASS}(c)$ for $c \in CLASS$ and
  $tgt_A(corr_{ATT}(att)) = x$ if $att : c \to s \in ATT_c$ and $\{x\} = Z_s$ with $s \in S$,
- $corr_{ASSOC} : ASSOC \to TG_E$ with $src_G \circ corr_{ASSOC} = corr_{CLASS} \circ pr_1$ and $tgt_G \circ corr_{ASSOC} = corr_{CLASS} \circ pr_2$ with $associates(a) = \langle c1, c2 \rangle$, $pr_i(a) = c_i$ for $i = 1, 2$, $c1, c2 \in Class$ and $a \in ASSOC$.

To show the correctness of our Core OCL invariant translation, we also need to establish a correspondence relation between system states and typed attributed graphs.

Definition 11 (State Correspondence). Let M be a core object model and $ATGI$ an attributed type graph, both defined over data signature $DSIG = (S, OP)$. We assume that $I(s) = D_s$ for all sorts $s \in S$. Furthermore, let $corr_{type}(M) = ATGI$ be a type correspondence.
Given a system state $\sigma(M) = (\sigma_{CLASS}, \sigma_{ATT}, \sigma_{ASSOC})$, it corresponds to an attributed graph $AG = (G, D)$ with $G = (G_V, G_D, G_E, G_A, src_G, tgt_G, src_A, tgt_A)$ typed over $ATGI$ by clan morphism *type* if there is a *state correspondence relation* $corr_{state} = (c_{CLASS}, c_{ATT}, c_{ASSOC}) : States(M) \to Graph_{ATGI}$ defined by the following bijective mappings:

- $c_{CLASS} : \sigma_{CLASS} \to G_V$ with
  $type_{G_V}(c_{CLASS}(o)) = corr_{CLASS}(c)$ with $o \in \sigma_{CLASS}(c)$ and $c \in CLASS$,
- $c_{ATT} : \sigma_{ATT} \to G_A$ with $src_A(c_{ATT}(a)) = c_{CLASS}(o)$ and
  $tgt_A(c_{ATT}(a)) = d$ as well as $type_{G_A}(c_{ATT}(\sigma_{ATT}(att))) = corr_{ATT}(att)$ and
  $a \in \sigma_{ATT}(att)$ if $att : c \to s \in ATT_c^{\prec}$, $\sigma_{ATT}(att) : \sigma_{CLASS}(c) \to D_s$,
  $o \in \sigma_{CLASS}(c)$, $c \in CLASS$ and $\sigma_{ATT}(att)(o) = d$,
- $c_{ASSOC} : \sigma_{ASSOC} \to G_E$ with
  $src_G \circ c_{ASSOC} = c_{CLASS} \circ pr_1$ and $tgt_G \circ c_{ASSOC} = c_{CLASS} \circ pr_2$
  with $l = (o1, o2) \in \sigma_{ASSOC}(assoc)$ and $pr_i(l) = o_i$ for $i = 1, 2$.
  Furthermore, $type_{G_E} \circ c_{ASSOC}(\sigma_{ASSOC}) = corr_{ASSOC}(ASSOC)$.

4.2 Translation of Core OCL Invariants

To get an initial understanding on how Core OCL invariants shall be translated to graph conditions, we take a pattern-based approach. The principle idea is that navigation expressions are translated to graphs and graph morphisms while the usual Boolean operations correspond to each other directly. The subset-operator of graph conditions is useful to correspond to iterating variables in iterator expressions. In Figure 1, basic OCL patterns and their corresponding graph constraint patterns are depicted. In these patterns, "Class", "v", "v1","v2", "b" , "c", and "r" are variables for model elements. Non-terminal `<op>` can be replaced by some comparator such as $=, <>, <$. Non-terminal `<expr>` may be replaced by any Core OCL expression.

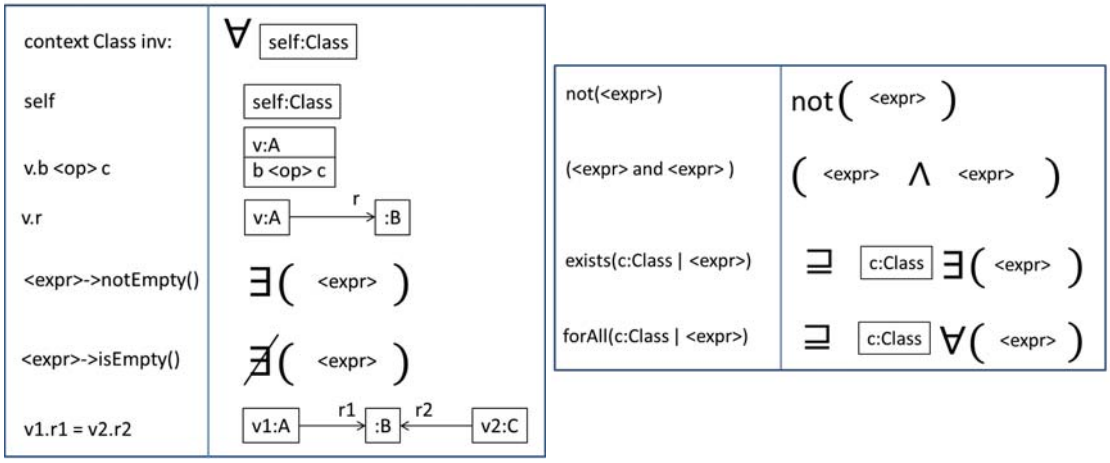


Fig. 1. Translation of basic OCL patterns to graph constraint patterns

In the following, we define the translation of Core OCL invariants as outlined in the beginning of this section. The translation is basically structured along the definition of Core OCL expressions given in Def. 3. However, operation expressions, If expressions and iterator expressions are distinguished along their result type yielding navigations (with $Set(t)$, $t \in CLASS$ as result type) and (Boolean) expressions. In the following definition, rules 1 and 2 translate the header of a CoreOCL invariant, rules 4 - 12 translate Boolean expressions, rules 13 - 15 translate basic object comparisons, rules 16 - 17 translate attribute value comparisons, and rules 18 - 19 translate basic navigation expressions and variables.

Definition 12 (Constraint translation). Let $M = (CLASS, ATT, ASSOC, associates, \prec)$ be a core object model with $ATGI = corr_{type}(M)$ being the corresponding attributed type graph and $t : Expr \to T$ a typing function which returns the type of an OCL expression (for T see Section 2). Let furthermore Invariant_M be the set of Core OCL invariants over M as defined in Def. 4 and $\text{GraphConstraint}_{ATGI}$ be the set of all graph constraints as defined in Definition 8. Then, the *translation functions*

- invariant translation: tr_I: Invariant$_M$ $\rightarrow$ GraphConstraint$_{ATGI}$
- expression translation tr_E: $Expr_{Boolean}$ $\rightarrow$ GraphConstraint$_{ATGI}$
- navigation translation tr_N: $Expr_c$ $\rightarrow$ Graph$_{ATGI}$ with $c \in CLASS$
- variable translation tr_V: Var_c $\rightarrow$ Graph$_{ATGI}$ with $c \in CLASS$

are defined as follows:

1. tr_I('context' C 'inv:' expr) = $\forall$ (self:C, tr_E(expr))
2. tr_I('context' var ':' C 'inv:' expr) = $\forall$ (tr_V(*var*), tr_E(expr))
3. tr_E (expr) = tr_E(setOpCallExpr) | tr_E(basicExpr) | tr_E(boolExpr) | tr_E(iteratorExpr)
4. tr_E(boolExpr) = *true* if boolExpr ::= '*true*'
5. tr_E(boolExpr) = ($\neg$ tr_E(expr)) if boolExpr ::= '(' 'not' expr ')'
6. tr_E(boolExpr) = (tr_E(expr1) op_g tr_E(expr2)) with $op_g \in \{\wedge, \vee\}$ if boolExpr ::= '(' expr1 op_b expr2 ')' with $op_b \in$ { 'and', 'or'}
7. tr_E(boolExpr) = ($\neg$ tr_E(expr1) $\vee$ tr_E(expr2)) if boolExpr ::= '(' expr1 'implies' expr2 ')'
8. tr_E(boolExpr) = ((tr_E(cond) $\wedge$ tr_E(expr1)) $\vee$ ($\neg$ tr_E(cond) $\wedge$ tr_E(expr2))) if boolExpr ::= '(' 'if' cond 'then' expr1 'else' expr2')'
9. tr_E(setOpCallExpr) = $\neg\exists$ (tr_N(navExpr)) if setOpCallExpr ::= navExpr '$\rightarrow$ isEmpty()'
10. tr_E(setOpCallExpr) = $\exists$ (tr_N(navExpr)) if setOpCallExpr ::= navExpr '$\rightarrow$ notEmpty()'
11. tr_E(iteratorExpr) = $\exists$ (tr_N(navExpr) $\sqsupseteq$ tr_V(var), tr_E(expr)) if iteratorExpr ::= navExpr '$\rightarrow$' exists (' var '|' expr ')'
12. tr_E(iteratorExpr) = $\forall$ (tr_N(navExpr) $\sqsupseteq$ tr_V(var), tr_E(expr)) if iteratorExpr ::= navExpr '$\rightarrow$' forAll (' var '|' expr ')'
13. (a) tr_E(basicExpr) = $\exists$ (v:t(v) v2:t(v2) $\rightarrow$ v,v2:t(v))[4] if basicExpr ::= v '=' v2
    (b) tr_E(basicExpr) = $\exists$ (v:t(v) v2:t(v2)) if basicExpr ::= v '<>' v2
14. (a) tr_E(basicExpr) = $\exists$ (tr_V(v ':' t(v)) $\overset{\text{as}}{\underset{\text{as2}}{\rightrightarrows}}$:t(r)) if basicExpr ::= v '.' r '=' v '.' r2, r is a role of as, r2 is a role of $as2$, and t(r) = t(r2) $\in CLASS$
    (b) tr_E(basicExpr) = $\exists$ (tr_V(v ':' t(v)) $\xrightarrow{as}$:t(r) $\xleftarrow{as2}$ tr_V(v2 ':' t(v2))) if basicExpr ::= v '.' r '=' v2 '.' r2, r is a role of as, r2 is a role of $as2$, and t(r) = t(r2) $\in CLASS$
15. (a) tr_E(basicExpr) = $\exists$ (:t(r) $\xleftarrow{as}$ tr_V(v) $\xrightarrow{as2}$:t(r2)) if basicExpr ::= v '.' r '<>' v '.' r2, r is a role of as, r2 is a role of $as2$, and t(r) = t(r2) $\in CLASS$

[4] Note that this a "non-injective" condition in the sense of [3], i.e. a condition with non-injective morphism. By [13], for each non-injective condition c and each morphism $p = m \circ e$ with e surjective and m injective, there is an injective condition Shift(e, c) in the sense of Definition 8 such that we have $p \models c \Leftrightarrow m \models$ Shift(e, c). Whenever a non-injective morphism occurs in a condition, we have to replace the whole condition by Shift(e, c).

(b) $tr_E(\text{basicExpr}) = \exists\ (tr_V(\text{v}) \xrightarrow{as} \boxed{\text{:t(r)}} \quad tr_V(\text{v2}) \xrightarrow{as2} \boxed{\text{:t(r2)}})$
if basicExpr ::= v '.' r '<>' v2 '.' r2, r is a role of as, r2 is a role of $as2$, and t(r) = t(r2) $\in CLASS$

16. $tr_E(\text{basicExpr}) = \exists\ (\boxed{\frac{v{:}t(v)}{attr\ op\ x}})$[5] if basicExpr ::= v '.' attr op x and x is a constant or a variable

17. (a) $tr_E(\text{basicExpr}) = \exists\ (\boxed{\frac{v{:}t(v)}{\begin{array}{c}attr\ =\ x\\ attr2\ op\ x\end{array}}})$
if basicExpr ::= v '.' attr op v '.' attr2, attr $\neq$ attr2, and x is a new variable with t(x) = t(attr) = t(attr2).

(b) $tr_E(\text{basicExpr}) = \exists\ (\boxed{\frac{v{:}t(v)}{attr = x}}\ \boxed{\frac{v2{:}t(v2)}{attr2\ op\ x}})$
if basicExpr ::= v '.' attr op v2 '.' attr2, v $\neq$ v2, and x is a new variable with t(x) = t(attr).

18. $tr_N(\text{navExpr}) = tr_V(\text{v}) \xrightarrow{as} \boxed{\text{:t(r)}}$
if navExpr ::= v '.' r and r is a role of as

19. $tr_V(\text{var}) = \boxed{\text{v:t(v)}}$
for var::= v ':' t with t = t(v) or var::= v

where $expr, expr1, expr2, boolExpr, setOpCallExpr, basicExpr, iteratorExpr \in Expr_{Boolean}, var \in Var_c, navExpr, navExpr1, navExpr2 \in Expr_c, c \in CLASS$, $v, v2 \in Var_t$, $t \in T_M$, $as, as2 \in ASSOC$, $r = r_{tgt}(as)$, $r2 = r_{tgt}(as2)$, $op \in \{<, >, \leq, \geq, =, <>\}$, $attr \in ATT_c$ and w.l.o.g., $corr_{CLASS}(c) = c$ for $c = t(r)$, $c = t(r2)$, $c = t(v)$ or $c = t(attr)$, $corr_{ASSOC}(as) = as$, $corr_{ASSOC}(as2) = as2$, and $corr_{ATT}(attr) = attr$.

In the following, we show two example translations using OCL invariants of Example 1. Small numbers behind equality signs denote the rules being used.

Example 3 (Translation of OCL constraint 1).

$tr_I($ 'context Transition inv: self.preArc→notEmpty()') $=^1$
$\forall\ (\boxed{\text{self:Transition}}\ tr_E(\text{'self.preArc→notEmpty()'})\ ) =^{10}$
$\forall\ (\boxed{\text{self:Transition}}, \exists\ (tr_N(\text{'self.preArc'}))) =^{18}$
$\forall\ (\boxed{\text{self:Transition}}, \exists\ (tr_V(\text{'self:Transition'}) \xrightarrow{preArc} \boxed{\text{:ArcPT}})) =^{19}$
$\forall\ (\boxed{\text{self:Transition}}, \exists\ (\boxed{\text{self:Transition}} \xrightarrow{preArc} \boxed{\text{:ArcPT}}))$

Example 4 (Translation of OCL constraint 3.1).

$tr_I($ 'context Petrinet inv: self.place→forAll(p1:Place | self.place → forAll(p2:Place | p1 <> p2 implies p1.name <> p2.name))') $=^1$
$\forall\ (\boxed{\text{self:Petrinet}}, tr_E(\text{'self.place→forAll(p1:Place | self.place →}$ forAll(p2:Place | p1 <> p2 implies p1.name <> p2.name))')) $=^{12}$
$\forall\ (\boxed{\text{self:Petrinet}}, \forall\ (tr_E(\text{'self.place'}) \sqsupseteq tr_V(\text{'p1:Place'}),$
$tr_E(\text{'self.place→forAll(p2:Place | p1<>p2 implies p1.name<>p2.name)'}))) =^{18,12}$

[5] Compare the short notation of attribute conditions in Example 2.

$\forall$ (self:Petrinet , $\forall$ (tr_V('self') $\overset{place}{\longrightarrow}$:Place $\sqsupseteq$ p1:Place ,
$\forall$ (tr_E('self.place') $\sqsupseteq$ tr_V('p2:Place'),
tr_E('p1 <> p2 implies p1.name <> p2.name')))) $=^{19,18,6}$
$\forall$ (self:Petrinet , $\forall$ (self:Petrinet $\overset{place}{\longrightarrow}$:Place $\sqsupseteq$ p1:Place ,
$\forall$ (tr_V('self') $\overset{place}{\longrightarrow}$:Place $\sqsupseteq$ p2:Place ,
$\neg$ tr_E('p1 <> p2') $\vee$ tr_E('p1.name <> p2.name')))) $=^{19}$
$\forall$ (self:Petrinet , $\forall$ (self:Petrinet $\overset{place}{\longrightarrow}$:Place $\sqsupseteq$ p1:Place ,
$\forall$ (self:Petrinet $\overset{place}{\longrightarrow}$:Place $\sqsupseteq$ p2:Place ,
($\neg$ tr_E('p1 <> p2') $\vee$ tr_E('p1.name <> p2.name'))))) $=^{10,13.b,17.b}$
$\forall$ (self:Petrinet , $\forall$ (self:Petrinet $\overset{place}{\longrightarrow}$:Place $\sqsupseteq$ p1:Place ,
$\forall$ (self:Petrinet $\overset{place}{\longrightarrow}$:Place $\sqsupseteq$ p2:Place ,
$\neg$ $\exists$ (p1:Place p2:Place) $\vee$ $\exists$ ($\frac{p1:Place}{name = x}$ $\frac{p2:Place}{name <> x}$))))

4.3 Correctness and Completeness

To be sure that the translation of Core OCL invariants is well-defined, we show its correctness and completeness. Moreover, we want to ensure that each translation terminates. The proofs of all the following results are given in [7] in their complete form. Here, we just present the main proof ideas.

Proposition 1 (Termination). The invariant translation tr_I as defined in Definition 12 terminates.

tr_I terminates since all invariants are finite and each application of a translation rule decreases the number of syntactic tokens in an invariant.

Theorem 1 (Completeness of translation). Given a core object model M and its corresponding attributed type graph $ATGI = corr_{type}(M)$, all Core OCL invariants over M are translated to some graph constraint over $ATGI$.

We have to show that all Core OCL invariants can be translated to graph constraints. The proof is performed by induction on the structure of Core OCL invariants. First, we start to translate Core OCL invariants and continue to show the completeness of the translation for Core OCL expressions.
To show that the translation of Core OCL invariants is correct, we consider their semantics and the semantics of graph constraints. If an invariant holds for a system state, the corresponding graph constraint is fulfilled by the corresponding graph.

Theorem 2 (Correct Translation of Core OCL invariants). Given an object model M and its corresponding attributed type graph $ATGI = corr_{type}(M)$,

the following statement holds for all Core OCL invariants $inv \in \text{Invariant}_M$: For all environments $env = (\sigma, \beta) \in Env$

$$I[\![inv]\!](env) = true \iff G = corr_{state}(\sigma) \models tr_I(inv).$$

The proof is performed by induction on the translation rules given in Def. 12.

5 From Core OCL Invariants to Application Conditions

After having translated Core OCL invariants to graph constraints, we connect this new result with the existing theory on graph constraints [3,14]. A main result shows how nested graph constraints can be translated to right, and thereafter, to left application conditions of transformation rules. In the following, we illustrate at an example how a Core OCL invariant is translated to a left application condition.

By the results in [3,13], for each category with pushouts and $\mathcal{E}'$-$\mathcal{M}$ pair factorization, nested conditions in this category can be shifted over morphisms and rules. By Fact 1, $ATGI$-graphs and morphisms form the category $\mathbf{AGraphs}_{ATGI}$ with pushouts and $\mathcal{E}'$-$\mathcal{M}$ pair factorization. Consequently, $ATGI$-conditions can be shifted over $ATGI$-morphisms and rules.

Lemma 1 (shift of $ATGI$-conditions over morphisms and rules [7]).
1. There is a Shift-construction such that, for each $ATGI$-condition c over P and for each $ATGI$-morphism $b\colon P \to P'$, $n\colon P' \to H$, $n \circ b \models c \iff n \models \text{Shift}(b, c)$.
2. There is a construction Left such that, for each $ATGI$-rule $\varrho = \langle L \hookleftarrow K \hookrightarrow R \rangle$, each $ATGI$-condition ac over R, and each direct transformation $G \Rightarrow_{\varrho,g,h} H$, we have $g \models \text{Left}(\varrho, \text{ac}) \iff h \models \text{ac}$.

In the following, we illustrate at an example how a Core OCL invariant is translated to a left application condition.

We present a simple rule to create places in a Petri net. The graph constraint from Example 2 shall be translated to a left application condition. The conditions are given in abbreviated form (i.e. whenever it is unambiguous, only the codomain of a morphism is shown); node mappings are obvious from their relative position. Edge labels are omitted for brevity.

Constraint: $\forall$ (self:Place, $\exists$ (self:Place | token $\geq$ 0))

Rule: :Petrinet $\Rightarrow$:Petrinet $\longrightarrow$:Place | token = 0

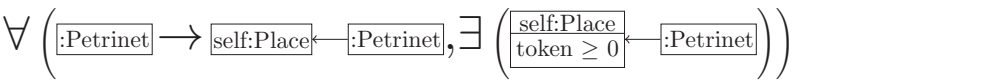
Right application condition:

$\forall$ (:Petrinet $\longrightarrow$ self:Place | token = 0, $\exists$ (:Petrinet $\longrightarrow$ self:Place | token = 0)) $\wedge$

$\forall$ (:Petrinet $\longrightarrow$:Place | token = 0 $\longrightarrow$:Petrinet $\longrightarrow$:Place | token = 0, $\exists$ (self:Place | token $\geq$ 0 $\longleftarrow$:Petrinet $\longrightarrow$:Place | token = 0))

(:Petrinet $\downarrow$ self:Place)

This condition states that new and existing places have to come with non-negative numbers of tokens. The left application condition looks as follows (after trivial simplifying):



This states that every place in the Petri net has a non-negative token count.

6 Related Work and Conclusion

In the literature, there are several significant approaches to define a formal semantics for OCL. The motivations for a formal OCL semantics are manifold and include defining a clear semantics, generating model instances, and performing formal verification of UML/OCL models. All main approaches are logic-oriented, in contrast to ours being the first one that relates OCL to a graph-based approach. In the following, we sketch logic-oriented approaches using the Key prover, the Alloy project, and Constraint Logic Programming, respectively.

In [15], Beckert et al. present a translation of UML class diagrams with OCL constraints into first-order logic; the goal is logical reasoning about UML models. The translation has been implemented as a part of the KeY system, but can also be used stand-alone. Formal methods such as Alloy [16] can be used for instance generation: After translating a class diagram to Alloy, an instance can be generated or it can be shown that no instances exist. This generation relies on the use of SAT solvers and can also enumerate all possible instances. In [17], UML models are automatically transformed to corresponding Alloy representations. Alloy models can then be analyzed automatically, with the help of the Alloy Analyzer. A recent work translating OCL to relational logic is presented in [18] covering more features than UML2Alloy. The USE tool [6,19] can be used for generating snapshots that conform to the model or for checking the conformity of a specific instance. In [20], Cabot et al. present UMLtoCSP, a tool that is able to automatically check correctness properties of UML class model with OCL constraints based on Constraint Logic Programming.

All these approaches have in common that they translate class models with OCL constraints into logical facts and formulas forgetting about the graph properties of class models and their instances. Hence, the reasoning is performed on the level of model elements. Translating OCL invariants to graph constraints allows to keep graph structures as units of abstraction while checking for satisfiability. Pennemann has shown in [14] that a theorem prover for graph conditions works more efficient than theorem provers for logical formulas being applied to graph conditions. The key idea is here that graph axioms are always satisfied by default when using a theorem prover for graph conditions. Furthermore, a translation of OCL to graph constraints yields a new visualization of OCL which can help understanding. And finally, our translation offers a way to translate Core OCL invariants to application conditions of transformation rules. This is a new form to apply an OCL translation which might lead to number of new applications including test model generation as well as auto-completion of model editing

operations. The backward translation from graph conditions to OCL might also be interesting to come up with model transformation rules restricted by OCL. In future work, we plan to extend this work towards the whole range of OCL invariants being translated to more powerful graph conditions.

Acknowledgement. We are grateful to Christoph Peuser and the anonymous referees for their helpful comments on a draft version of this paper.

References

1. OMG: Object Constraint Language, http://www.omg.org/spec/OCL/
2. Bardohl, R., Minas, M., Schürr, A., Taentzer, G.: Application of Graph Transformation to Visual Languages. In: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2, pp. 105–180. World Scientific (1999)
3. Habel, A., Pennemann, K.H.: Correctness of High-Level Transformation Systems Relative to Nested Conditions. Mathematical Structures in Computer Science 19, 245–296 (2009)
4. Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. Software and System Modeling 11(2), 227–250 (2012)
5. Chiorean, D., Bortes, M., Corutiu, D.: Proposals for a Widespread Use of OCL. In: Workshop on Tool Support for OCL and Related Formalisms, Technical Report LGL-REPORT-2005-001, EPFL, 68–82 (2005)
6. Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universität Bremen, Logos Verlag, Berlin (2002)
7. Arendt, T., Habel, A., Radke, H., Taentzer, G.: From Core OCL Invariants to Nested Graph Constraints: Extended version (2014), http://www.uni-marburg.de/fb12/forschung/berichte/berichteinformtk/pdfbi/bi2014-01.pdf
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental Theory of Typed Attributed Graph Transformation based on Adhesive HLR Categories. Fundamenta Informaticae 74(1), 31–61 (2006)
9. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs of Theoretical Computer Science. Springer (2006)
10. Orejas, F.: Symbolic Graphs for Attributed Graph Constraints. J. Symb. Comput. 46(3), 294–315 (2011)
11. Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 214–228. Springer, Heidelberg (2004)
12. Poskitt, C.M., Plump, D.: Hoare-Style Verification of Graph Programs. Fundamenta Informaticae 118(1-2), 135–175 (2012)
13. Ehrig, H., Golas, U., Habel, A., Lambers, L., Orejas, F.: $\mathcal{M}$-Adhesive Transformation Systems with Nested Application Conditions. Part 1: Parallelism, Concurrency and Amalgamation. Mathematical Structures in Computer Science 24 (2014)
14. Pennemann, K.H.: Development of Correct Graph Transformation Systems. PhD thesis, Universität Oldenburg (2009)

15. Beckert, B., Keller, U., Schmitt, P.H.: Translating the Object Constraint Language into First-order Predicate Logic. In: VERIFY, Workshop at Federated Logic Conferences, FLoC (2002)
16. Jackson, D.: Alloy Analyzer website (2012), `http://alloy.mit.edu/`
17. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and System Modeling 9(1), 69–86 (2010)
18. Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 415–431. Springer, Heidelberg (2012)
19. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. SoSyM 4(4), 386–398 (2009)
20. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models using Constraint Programming. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 547–548 (2007)