Electronic Communications of the EASST
Volume 38 (2010)

Proceedings of the Fifth International Conference on Graph Transformation - Doctoral Symposium (ICGT-DS 2010)

Search-Based Refactoring based on Unfolding of Graph Transformation Systems

Fawad Qayum and Reiko Heckel

14 pages

Guest Editor: Andrea Corradini
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Search-Based Refactoring based on Unfolding of Graph Transformation Systems

Fawad Qayum[1] **and Reiko Heckel**[2]

[1] fq7@le.ac.uk
[2] reiko@mcs.le.ac.uk
Department of Computer Science
University of Leicester, Leicester
United Kingdom

Abstract: To improve scalability and understandability of search-based refactoring, in this paper, we propose a formulation based on graph transformation which allows us to make use of partial order semantics and an associated analysis technique, the approximated unfolding of graph transformation systems. We use graphs to represent object-oriented software architectures at the class level and graph transformations to describe their refactoring operations. In the unfolding we can identify dependencies and conflicts between refactoring steps leading to an implicit and therefore more scalable representation of the search space. An optimisation algorithm based on the Ant Colony paradigm is used to explore this search space, aiming to find a sequence of refactoring steps that leads to the best design at a minimal costs.

Keywords: Search-based Refactoring, Unfolding of Graph Transformation Systems, Ant Colony Optimisation Meta-heuristic.

1 Introduction

Refactoring has emerged as a successful technique to enhance object-oriented software designs by series of small, behaviour-preserving transformations [Fow99]. However, due to the number of design choices and the complex dependencies and conflicts between them it is difficult to choose an optimal sequence of refactoring steps, maximising the quality of the resulting design while minimising the cost of the transformation. In the case of large systems the situation becomes acute because existing tools offer only limited support for their automated application [MTR07]. Therefore, search-based approaches have been suggested in order to provide automation in discovering appropriate refactoring sequences [SSB06, HPJ01]. The idea is to see the design process as a combinatorial optimisation problem, attempting to derive the best solution (with respect to a given quality measure or *objective function*) from a given initial design [OM02].

Two obvious problems with search-based approaches are scalability, i.e., the ability to apply to large models [OC08], and traceability, i.e., the ability on behalf of the developer to understand the changes suggested by the optimisation [HPJ01]. In particular, heavy modifications make it difficult to relate the improvement to the original design, so that developers will struggle to understand the new structure. We believe that both problems

can be mitigated by exploiting the local nature of refactoring operations, which affect only a certain part of the design while leaving the context unchanged. In terms of scalability, local operations permit the use of partial order models representing the behaviour of a system by a set of actions (refactoring steps) equipped with relations of causality and conflict. Such models provide an implicit representation of the states (designs) of the system as conflict-free subsets of actions closed under causal dependencies, which scales better than the explicit representation of reachable states. For traceability, causal dependency provide a model of explanation of why certain steps are required to perform later steps, thus reducing the problem to understanding the benefits of the final steps in a sequence.

In this paper, we use a representation of object-oriented designs as graphs and refactoring operations as graph transformation rules [MTR07]. Such rules provide a local description, identifying and changing a specific part of the design graph only. After suitably encoding our rules into a hypergraph representation, this enables us to derive a partial order structure of causality and conflict relations, using the approximated unfolding of a graph transformation system [BCM99] and its implementation in Augur 2 [KK08]. The result is a structure called *Petri graph*, presenting the behaviour in terms of an over-approximation of its transformations and dependencies [BCK01]. Causal dependencies and conflicts, derived directly from the Petri graph, serve as input to our search problem.

Optimisation algorithms such the Ant Colony Optimisation [dor05] (ACO) metaheuristic rely on an explicit representation of the search space. Thus states and their local neighbourhoods have to be reconstructed on the fly from the partial order representation. The desired result is a sequence of transformations leading from the given design to a design of high(er) quality, using only transformation steps that are necessary to achieve that improvement.

A more detailed view of the approach is given by the diagram in Figure 1. Using UML activity diagram notation, boxes represent artifacts while oval nodes are the actions or transformations performed on them. The class structure of a given Java program (excluding method bodies, but retaining call and data access dependencies) is encoded in the GXL format required by Augur 2. This is achieved with the help of the Infusion environment[1] and a subsequent transformation of the resulting MSE[2] file into GXL. The result represents the start graph of the hyper graph grammar to be unfolded. The rules of the grammar formalising the refactoring operations are derived from the standard catalogue [Fow99] shared across all Java programs. Augur 2 constructs the approximated unfolding of a system [BCM99], producing a Petri graph to serve as input to the ACO-based search algorithm.

ACO is inspired by the behaviour of foraging ants, which search for food individually and concurrently, but share information about food sources and paths leading towards them by leaving pheromone trails. This amounts to a distributed traversal of a graph whose paths represent possible solutions [DMG97]. In our case, the nodes of that graph

[1] http://www.intooitus.com/inFusion.html
[2] http://www.moosetechnology.org/docs/mse

are the designs to be explored and its edges are the refactoring steps. Rather than representing this so-called construction graph explicitly, its nodes and edges are derived from the partial order structure as and when required. As a result, a path (refactoring sequence) is produced representing the cheapest way to transform the given design into an optimal one. Since the unfolding represents an over-approximation, the existence of this sequence needs to be verified in the real model, possibly leading to a refinement of the approximation. However, this step is beyond the scope of this paper.

The remainder of the paper is organised as follow. In Section 2, we review the presentation of refactorings as graph transformations and introduce our example. Section 3 describes the partial order analysis based on unfolding. The mapping into an ACO problem is addressed in Section 4. Finally we evaluate our approach and conclude.

2 Refactoring as Graph Transformation

In order to provide a localised formal description of refactorings as input to the partial order analysis, we follow [MTR07] in representing refactoring operations as graph transformation rules. Informally, such a rule $p : L \to R$ consists of a rule name p and a pair of graphs L, R called the left- and right-hand side of p. A transformation $t : G \overset{p(m)}{\Longrightarrow} H$ changes graph G into graph H by replacing the occurrence of L specified by m with a copy of R. Following the algebraic double-pushout approach [EPS73], the change is local because elements of G outside the occurrence of L are not affected by the transformation.

The graphs we transform represent Java class structures, which can be visualised by class diagrams. As an example, consider the diagram in Figure 2. We consider the following set of refactoring operations [Fow99].

- Extract Superclass, creating a common superclass for two existing classes, usually in order to encapsulate shared features.
- Add Parameter, introducing a new parameter for a method to make data access explicit.
- Pull Up Method, transferring a method from a sub to a superclass.
- Move Method, transferring a method to any other class.
- Encapsulate Attribute, to increase the modularity by changing a visibility of attribute in a class from public to private.

The rule Extract Superclass is shown in Figure 3 in class diagram notation. Rules can be applied in different orders and locations, giving rise to a number of refactoring sequences. Below we describe and motivate some of these for future reference.

T_1**:** Extract superclass E from class B and class C, e.g., in order to encapsulate shared methods.

T_2**:** Pull up *method* from class B and class C to superclass E created by T_1.

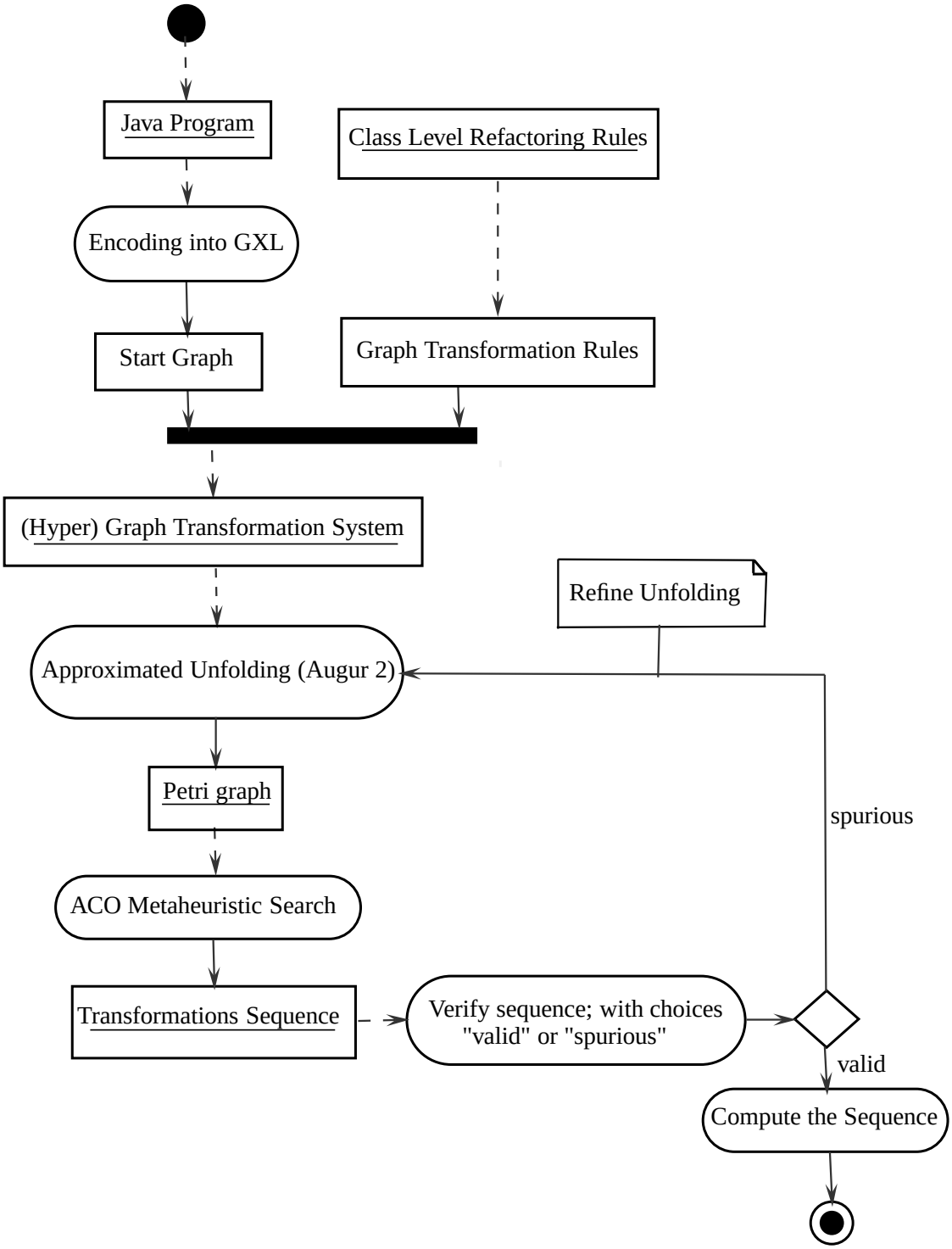

Figure 1: Abstract view of the Approach

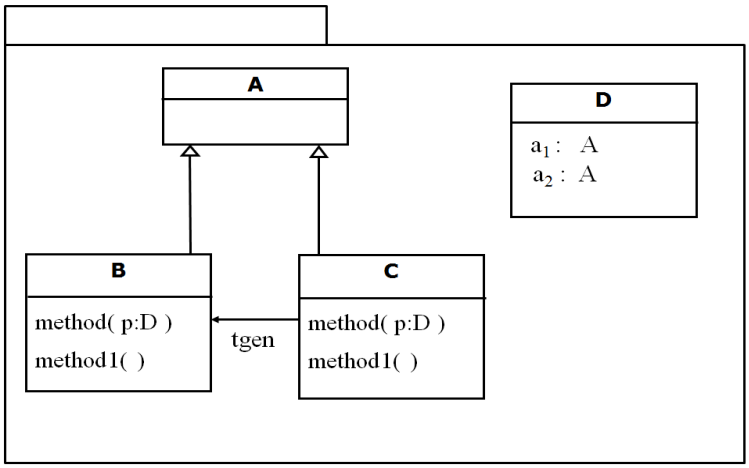

Figure 2: Initial class model

T_3**:** Move *method* from class B to class D, because it may be more tightly coupled to that class (e.g., accessing its attribute).

T_4**:** Move *method* from class C to class D, with the same motivation as in T_3.

T_5**:** Encapsulate attribute a_1 in class D, making the attribute private and creating setter and getter methods.

T_6**:** Add parameter p of type D to *method1* in class B to make explicit the access to instances of D.

T_7**:** Add parameter p of type class D to *method1* in class C, for the same motivation.

Note that the transformations listed are not part of a single sequence. For example T_2,T_3 are potentially in conflict.

3 Analysis of Dependencies and Conflicts

As outlined in the introduction, we use an implicit representation of the search space based on causality and conflict relations over rule applications representing refactoring steps such as T_1 to T_7 above. These partial orders are derived in two steps. First, the approximated unfolding of the grammar given by the start graph representing the initial design and the generic refactoring rules is produced and second partial orders are derived by analysing the overlaps of pre and postconditions of these rules in the resulting Petri graph.

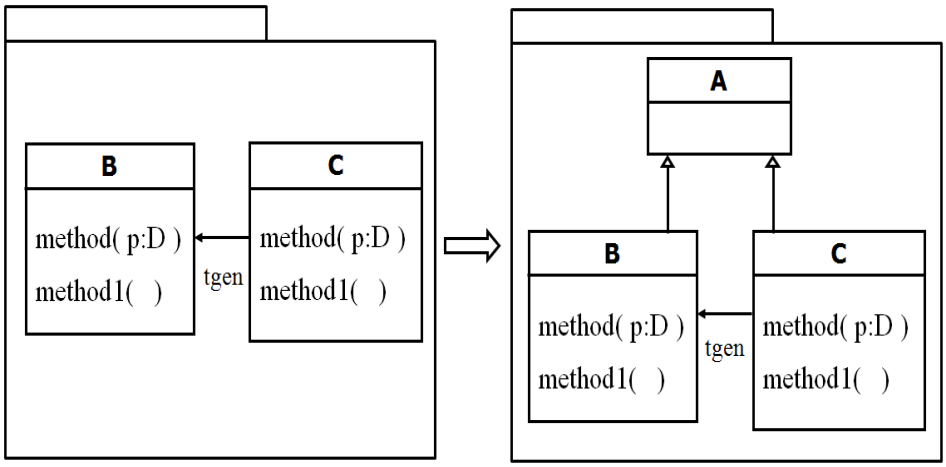

Figure 3: Rule for refactoring Extract Superclass, in class diagram notation

3.1 Unfolding of the Refactoring Grammar

The approximated unfolding and its implementation in Augur 2 [BCM99] are defined for hypergraph grammars. Thus, we have to encode the initial design and refactoring rules into a hypergraph representation. According to [BCK01] a hypergraph G is a tuple (V_G, E_G, C_G, l_G) where V_G and E_G are finite sets of nodes and edges respectively, $C_G : E_G \to V_G^*$ is a connection function, while l_G is a labelling function for edges. The difference with the more common notions of typed or labelled graphs with binary edges is that, in hypergraphs, only edges are labelled and that each edge can be connected to a finite sequence of nodes, rather than just one source and one target. The hypergraph of the initial class model is depicted in Figure 5. The idea is to introduce a node for each node in the original graph and plus one edge to carry a label representing the type of the node. Additional binary hyperedges are introduced to represent edges in the original binary graphs. Rules undergo a similar transformation, but an additional restriction (imposed by the theory of unfolding) is that rules can delete and produce, but not preserve edges, while nodes cannot be deleted. The left-hand side of a rule must be connected [BCK01]. This does not directly impact on the expressivity of the rules, but requires us to delete and regenerate edges that are meant to be preserved. The result for Extract Superclass is shown in Fig 6. Nodes of the left-hand side are mapped to those in the right-hand side with the same number, while the unnumbered nodes in the right are newly created. Edges in the left- and right-hand side are disjoint.

Given the hypergraph grammar, the unfolding starts with the initial hypergraph and produces a branching structure by applying all possible rules on the system at all possible matches. The resulting Petri graph contains both the graph structure of the system (essentially the union of all reachable graphs) and a Petri net with hyperedges as places and rule applications as transitions. The *approximated* unfolding creates a more abstract structure, potentially folding into one several graph elements or rule applica-

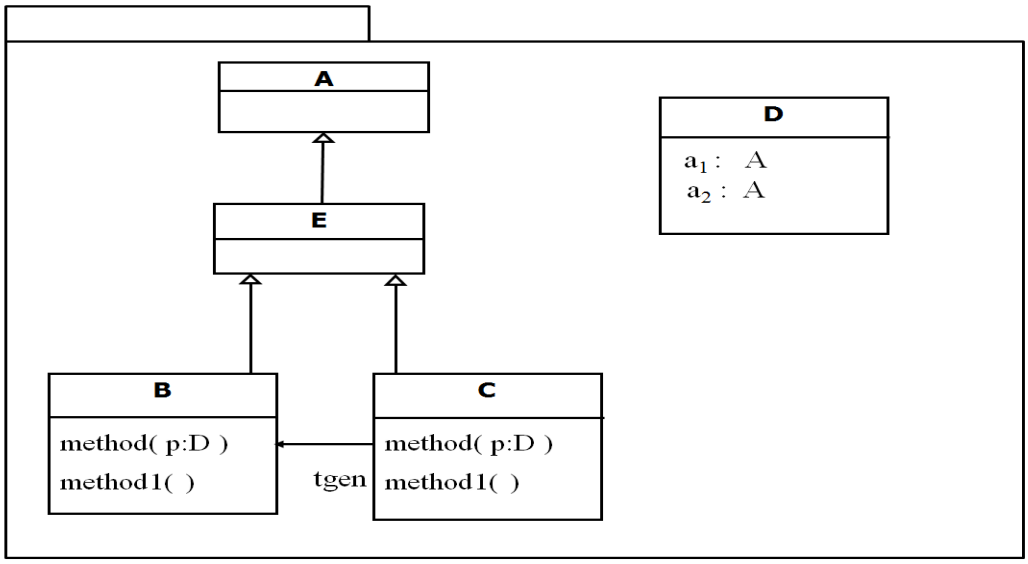

Figure 4: Result of applying rule Extract Superclass to the initial class model

tions [BCK01]. The result is an over approximation of the behaviour, i.e., spurious sequences may appear that do not exist in the actual behaviour. We use the Augur 2 implementation of this construction [KK08] where initial hypergraph and rules are presented in the Graph Exchange Language GXL [Tae01]. The output Petri graph produced is in GXL format as well [DKSR04].

3.2 Analysis of the Unfolding

A Petri graph [BCK01] is a finite data structure which records the behaviour of a graph transformation system. It combines hypergraphs with Petri nets used to approximate the behaviour. The hyperedges of the graph component are at the same time the places of the Petri net. The GXL representation produced by Augur 2 [DKSR04] is a low-level graph format, which knows about graph elements and their attributes, but not about transformations. To create a problem-specific data structure to allow for dependency analysis, we have to extract information about rules and transformations, the graphs they consist of, etc. Then we can derive conflict and dependency relations by comparing the pre- and post-sets of transformations. The class diagram in Figure 7 provides the conceptual data model for the unfolding. A Java object graph representing an instance of this model is extracted from the GXL representation produced by Augur 2.

From the pre- and post-conditions in this high-level representation we can extract causality and conflict relations on transitions. Using Petri net-like notation, we represent the pre- and post-sets for a transition t by ${}^\bullet t$ and $t^\bullet$, respectively [BCM99]. Then, two transitions are in conflict, $t_1 \# t_2$, if and only if ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \phi$. They are causally dependent, $t_1 < t_2$, if and only if $t_1^\bullet \cap {}^\bullet t_2 \neq \phi$.

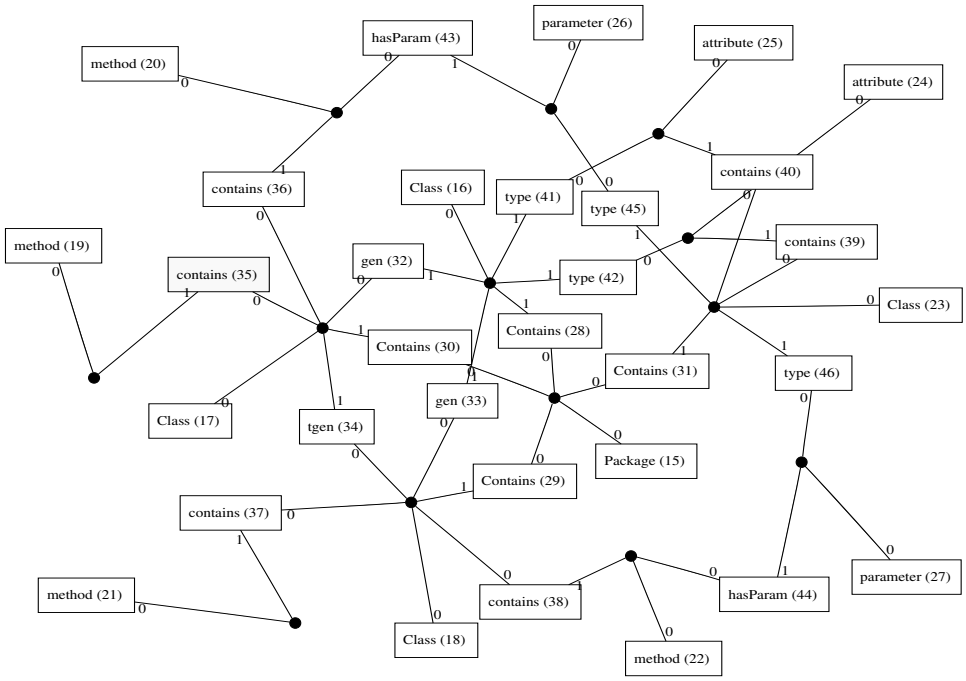

Figure 5: Initial Hypergraph

The set of transitions $t_1, t_2, \ldots$ representing refactoring steps and relations $\#$ and $<$ provide the input to our search for an optimal sequence of refactorings.

4 Refactoring as ACO Problem

We employ Ant Colony Optimisation (ACO) [dor05] meta-heuristic search to find an optimal solution. ACO is applicable to a wide range of combinatorial optimisation problems [dor05]. It is based on a set of artificial ants cooperating to find a solution by searching a graph independently, but leaving pheromone deposits on the graph's edges to indicate promising paths. To do this, ants have to know the local neighbourhood of their current solution node, from which they will select the most likely edge to traverse based on the evaluation of the successor node and the pheromone values of the edge itself. Formally, ACO problem [dor05] consists of the following elements.

1. A finite set of solution components $C = \{c_1, c_2, \cdots, c_n\}$, and a set of arcs E connecting the components in C.

2. The states of the search problem, defined as sequences of components $x = \langle c_i, c_j, \cdots \rangle$ in C. The set of all possible states x is denoted X. The length (number of components) of a sequence is denoted by $\mid x \mid$.

3. A finite set S of candidate solutions with distinguished subset $\bar{S} \subseteq S$ of *feasible* candidate solutions determined by a set of constraints Ω.

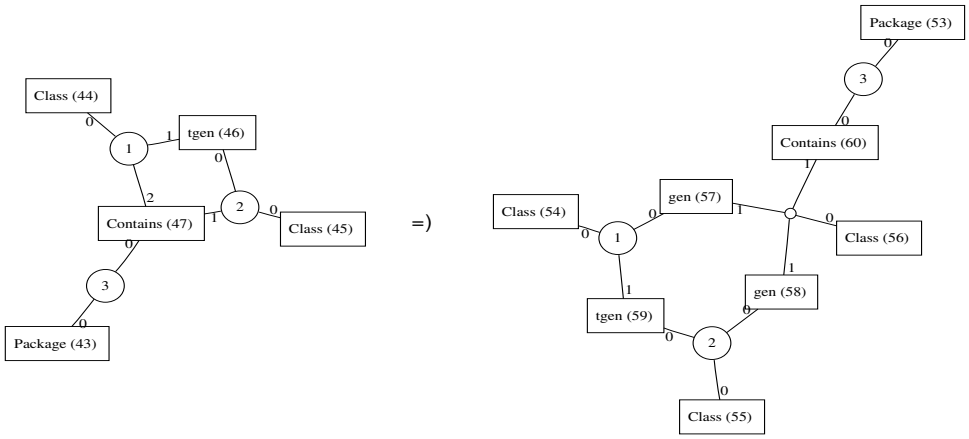

Figure 6: Hypergraphs for Extract Superclass Refactoring Rule

4. A non-empty subset S^* of optimal solutions.

5. An evaluation $f(s)$ for each candidate solution s. For some problems it is possible to calculate *partial evaluations* $f_p(x)$ associated with intermediate states x of the problem.

Using the formulation above, artificial ants build solutions by performing randomised walks on the connected graph $G = (C, E)$, based on the following basic operations [DMG97].

- A *state transitions* takes an ant from a one node to another across an arc;
- A *local update* changes the pheromone deposit on the arc it currently walks on;
- A *global update* changes the pheromone deposits on all arcs an ant has traversed when this ant successfully ends its trip;

In addition, we require a *comparison function* to evaluate different paths and an *end of activity* condition to specify when an ant has completed its trip.

To state refactoring as an ACO problem we consider a graph defined by the set of transformation steps as nodes with edges representing potential successor relations derived from dependencies and conflicts as obtained from the unfolding construction. These conflicts include symmetric ones, requiring mutual exclusion of two refactoring steps, and asymmetric ones, prohibiting two steps to occur in a certain order, but allowing for the reverse order. Pheromone values τ_{ij} and heuristic values η_{ij} are associated with the edges of the graph. The values are determined by partial evaluations $f_p(x)$, associated with incomplete candidate solutions x, which represent preliminary feedback on the success of the search.

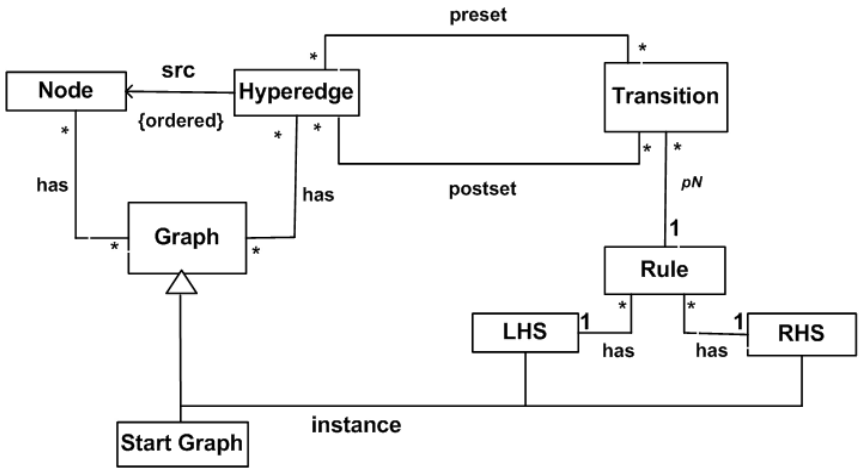

Figure 7: Metamodel for the Unfolding process

The problem is thus expressed as the search for an optimal path representing the best sequence of refactoring steps applicable to the original system. The optimisation depends on an evaluation of paths representing candidate solutions, which takes into account both the cost of the refactoring transformations and the quality of the end result.

We use a so-called Hybrid Ant System [GD00] where ACO is extended by local search, in particular, the Java Framework by Chiricom [Chi] implementing [DMG97] in order to implement and solve a variety of ACS problems.

We adapted this framework to an implicit representation of states based on our partial order model, deriving states and their local neighbourhood on the fly.

4.1 Deriving States and Transitions

States S are subsets of transitions that are conflict-free and closed under causal dependencies, i.e., $S = \{t \in T_N \mid t' \in S \text{ and } t < t' \text{ and } \nexists t'' \in S \text{ s.t. } t \# t''\}$, where T_N is set of transitions. The neighbourhood for a state s is characterised by all transitions enabled in s. A transition t is enabled in s if all its dependencies are satisfied by transitions in s and it is not in conflict with any transition in that set. Adding such a transition leads to a new state $s \cup \{t\}$. While computing the neighbourhood for a state in the search space, we need to check that the new transition is not yet present in the state. The conditions for enabled transitions ensure that the new state is well-defined, i.e., the added transition does not introduce conflicts or unresolved dependencies.

With these prerequisites the algorithms proceeds as follows.

- We initialise each ant by assigning an empty state $s_0 = \emptyset$.
- In each state s, an ant will determine its local neighbourhood by computing all transitions t_i enabled in s, with successor states $s_i = s \cup \{t_i\}$. It will select one of its neighbouring states based on the states' evaluation and the pheromone values

Table 1: Step sequences computed by ants

Ant ID	**Computed Path Node ID's**
1.	[5, 6, 4, 3, 0]
2.	[1, 2, 0, 6, 5]
3.	[0, 6, 5, 1, 2]
4.	[5, 6, 0, 4, 3]
5.	[6, 5, 0, 1, 2]
Best Path Node ID's:	**[5, 6, 0, 4, 3]**
Corresponding Rule ID's:	[_5291, _5290, _5296, _5292, _5293]
Optimal Sequence of Refactorings:	$[T_7, T_6, T_5, T_1, T_2]$

associated with the transition.

- Moving to the selected state, the ant will update the pheromone deposit.
- The ant stops if there are no more new transitions to be added, i.e., all remaining transitions are in conflict with transitions in the current state.
- A global update will take place to increase the pheromone deposits on all arcs leading to success, or decrease them in case of failure.

4.2 Objective Function

In order to formalise a notion of quality, we define probe rules as patterns to recognise situations that are desirable or to be avoided in object-oriented designs. Then, we will look for a state having a maximum number of desirable and a minimum number of undesirable occurrences. Using the unfolding as underlying data structure, such information about probe rule occurrences is available at little extra cost.

For every probe p and state s, we define $\#_p(s)$ as the number of occurrences of probe p in s. It will return negative integers for anti patterns. Assuming probes $p_1, p_2, \cdots, p_n$ (both positive and negative) the objective function is defined by $O(s) = \langle \#_{p_1}(s), \cdots, \#_{p_n}(s) \rangle$, returning a vector of probe counts.

Thus knowledge about good and bad patterns is embedded in occurrence function $\#_p$. We use pointwise extension of $\leq$ from integers to vectors to define a partial order on the states, i.e., $v_1 \leq v_2$ if and only if $v_1[j] \leq v_2[j]$ for all $1 \leq j \leq n$. That means the relation must holds for every entry in the vectors to holds for the vectors in total.

The probability of choosing the next transition depends on the quality of the successor state, i.e., the number of occurrence of probe rules. Each ant will compute the probe vector while it moves from one state to another in the search space.

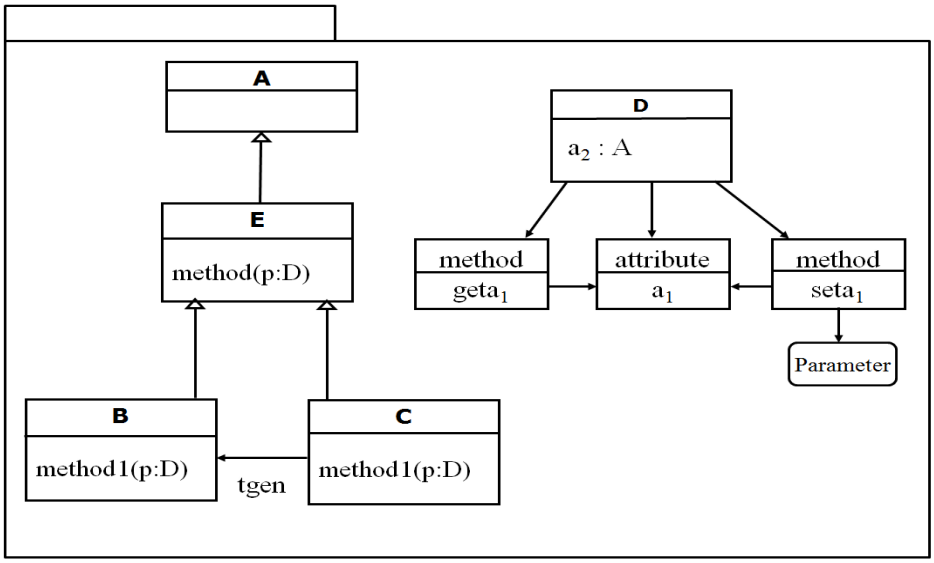

Figure 8: Final Class level diagram

4.3 Experimental Results

Given the problem representation as in Section 4, we consider the given set of refactoring steps as the set of components C. The search space will be defined by the associated set of transitions and each transition will be assigned an identification number for reference. We can employ any (finite) number of ants, depending on the size of the problem and computing resources available. In our example we use five ants, each to start with an empty state ($|x| = 0$). They select enabled transitions to move in the search space and which will enlarge their states and enable more transitions until all remaining transitions are in conflict. To guide the behaviour of future ants based on preliminary success, each ant will assign an improvement to the edge traversed when adding component c to path x. The objective function $O(S)$ will evaluate the best resulting design by assigning a probe vector.

The best path computed by the algorithm, representing an optimal sequence of refactorings is given in Table 1. It represents a sequence of steps of the set in Section 2 for the initial class model in Figure 2. The resulting class model is visualised in Figure 8.

5 Conclusion

Our approach involves a combination of graph transformation theory and the ACO meta heuristic, aiming to improve search-based refactoring. Rather than representing the search space of designs and refactorings explicitly we use the unfolding as a more scalable representation where designs (states) are given by sets of transformations closed under causal dependencies. We can thus reconstruct states when needed, for example in order

to evaluate the objective function, but will deal with the more compact representation when navigating the search space. As a further tribute to scalability, we are using the approximated unfolding. Algorithmically, we are following a hybrid approach [GD00] where the ACO meta heuristic is augmented with local search to improve its performance. Hybrid ACO has been shown to be effective in situations of large and rugged search spaces with complex constraints on solutions. In particular, the implicit representation of states (by a sets of transformations closed under causality and without conflicts) should allow us to scale the search to larger problems, avoiding state-space explosion.

Traceability will be evaluated through experiments with smaller models, assessing the effort it takes a human developer to understand the changes proposed by the search-based approach. The use of dependency information between transformations allows us to remove steps that are unrelated to the intended change, making each change relevant and therefore easier to interpret.

We have implemented the approach up to a point where it remains to check that sequences produced in the approximated model are also executable in the full model. If the sequence does not exists in the real model then a refinement of the abstraction will be required [KK06], which will lead to a more accurate unfolding and another round of optimisation.

Acknowledgements: Fawad Qayum is financed by the Higher Education Commission of Pakistan under Overseas Faculty Development Programme University of Malakand, for a PhD studentship at University of Leicester.

References

[BCK01] P. Baldan, A. Corradini, B. König. A Static Analysis Technique for Graph Transformation Systems. In *Proc. of CONCUR '01*. Pp. 381–395. Springer-Verlag, 2001. LNCS 2154.

[BCM99] P. Baldan, A. Corradini, U. Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In *FoSSaCS '99: Held as Part of the European Joint Conf: on the Theory and Practice of Software, ETAPS'99*. Pp. 73–89. Springer-Verlag, London, UK, 1999.

[Chi] U. Chirico. A Java Framework for Ant Colony Systems.

[DKSR04] F. L. Dotti, B. König, O. M. dos Santos, L. Ribeiro. A case study: Verifying a mutual exclusion protocol with process creation using graph transformation systems. Technical report, 2004.

[DMG97] M. Dorigo, S. Member, L. M. Gambardella. Ant Colony System: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1997.

[dor05] Review of "Ant Colony Optimization" by M.Dorigo, T.Stützle, MIT Press, Cambridge, MA, 2004. *Artif. Intell.* 165(2):261–264, 2005. Reviewer-Christian

Blum.
doi:http://dx.doi.org/10.1016/j.artint.2005.03.003

[Fow99] M. Fowler. *Refactoring: Improving The Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[GD00] L. M. Gambardella, M. Dorigo. An Ant Colony System Hybridized with a New Local Search for the Sequential Ordering Problem. *INFORMS J. on Computing* 12(3):237–255, 2000.
doi:http://dx.doi.org/10.1287/ijoc.12.3.237.12636

[HPJ01] M. Harman, U. Ph, B. F. Jones. Search-Based Software Engineering. *Information and Software Technology* 43:833–839, 2001.

[KK06] B. König, V. Kozioura. Counterexample-guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In *Proc. of TACAS '06*. Pp. 197–211. Springer, 2006. LNCS 3920.

[KK08] B. König, V. Kozioura. Augur 2—A New Version of a Tool for the Analysis of Graph Transformation Systems. In *Proc. of GT-VMT '06 (Workshop on Graph Transformation and Visual Modeling Techniques)*. ENTCS 211, pp. 201–210. Elsevier, 2008.

[MTR07] T. Mens, G. Taentzer, O. Runge. Analysing Refactoring Dependencies Using Graph Transformation. *Software and Systems Modeling (SoSyM)*, pp. 269–285, September 2007.

[OC08] M. O'Keeffe, M. O. Cinnéide. Search-based refactoring: an empirical study. *J. Softw. Maint. Evol.* 20(5):345–364, 2008.
doi:http://dx.doi.org/10.1002/smr.v20:5

[OM02] O. C. M. O'Keeffe M. A stochastic Approach To Automated Design Improvement. In *Proc. of the 2nd Inter: Conf: on the Principles and Practice of Programming in Java.*. Pp. 56–62. Comp.Sc Press, Trinity College Dublin: Ireland, ACM SIGAPP, 2002.

[SSB06] O. Seng, J. Stammel, D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO 2006*. Pp. 1909–1916. ACM, New York, NY, USA, 2006.
doi:http://doi.acm.org/10.1145/1143997.1144315

[EPS73] H. Ehrig, M. Pfender, H. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*. Pp. 167–180. IEEE, 1973.

[Tae01] G. Taentzer. Towards Common Exchange Formats for Graphs and Graph Transformation Systems. *Electr. Notes Theor. Comput. Sci.* 44(4), 2001.