Software Tools for Technology Transfer (STTT) manuscript No.
(will be inserted by the editor)

Experimental Assessment of Combining Pattern Matching Strategies with VIATRA2

Ákos Horváth · Gábor Bergmann · István Ráth · Dániel Varró

Received: date / Revised version: date

Abstract As recent tool contests demonstrated, graph transformation tools scale up to handle very large models for model transformations thanks to recent advances in graph pattern matching techniques. In this paper, we assess the performance and capabilities of the VIATRA2 model transformation framework by implementing the AntWorld case study of the GraBats 2008 graph transformation tool contest. First, we extend initial measurements carried out in [1] to assess the effects of combining local-search based and incremental pattern matching strategies. Moreover, we also assess the performance characteristics of various language features of VIATRA2 as well as the cost of certain model manipulation operations. We observe by experimentation how VIATRA2 can scale up to large iteratively growing model sizes and focus on execution time and memory consumption. We believe that the results obtained from the benchmark example can set the course for further performance enhancement of VIATRA2 and other future model transformation frameworks.

1 Introduction

Automated model transformations play an important role in modern model-driven system engineering in order to query, derive and manipulate large, industrial models. Since such transformations are frequently integrated into design environments, they need to provide short reaction time to support software engineers.

This work was partially supported by EU projects SENSORIA (IST-3-016004) and SecureChange (ICT-FET-231101).

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary

Graph transformation (GT) [2] based tools have been frequently used for specifying and executing complex model transformations. In GT tools, *graph patterns* capture structural conditions and type constraints in a compact visual way. At execution time, these conditions need to be evaluated by *graph pattern matching*, which aims to retrieve one or all matches of a given pattern to execute a transformation rule.

Benchmark measurements conducted at recent tool contests [3, 4] demonstrated that GT tools scale up for transforming very large models, thanks to sophisticated, *local-search based graph pattern matching* algorithms proposed in transformation tools such as GrGEN.NET [5], FUJABA [6], or VIATRA2 [7]. As a commonality in all these approaches, pattern matching is driven by a search plan, which provides an optimal (or sufficiently good) ordering for traversing and matching the nodes and edges of a graph pattern.

As an alternative, incremental pattern matching approaches (INC) [8–12] have recently become a hot research topic in the model transformation community. The basic idea is to improve the execution time of the time-consuming pattern matching phase by imposing additional memory consumption. Essentially, the (partial or full) matches of graph patterns are stored explicitly, and these match sets are updated incrementally upon elementary model changes. While model manipulation becomes slightly more complex, all matches of a graph pattern can be retrieved in constant time, thus eliminating the need for recomputing existing matches.

The VIATRA2 model transformation framework [7] supports both pattern matching strategies, which can be selected separately for each graph pattern / transformation rule. While initial measurements [13] implied that in many scenarios the incremental pattern matching approach (of VIATRA2) significantly outperforms

the local-search based approach (of VIATRA2), recent applications [14] revealed that available memory can be insufficient for caching match sets for the incremental approach, especially, on an average desktop computer.

A primary goal for the current paper is to investigate if there are benefits in combining incremental and local-search based pattern matching strategies using the AntWorld benchmark [15]. This extends our initial investigations for the problem [1] by carrying out a more systematic evaluation and fine tuning for selecting the right strategy for the AntWorld case study.

In addition, we also investigate the efficiency of (the implementation of) certain features of the VIATRA2 language [16] to support the transformation designer in using the appropriate language constructs and to trigger further development efforts. Then, we reason about the performance of core model manipulation operations of the VIATRA2 model space, and apply these results to futher optimization. Finally, we also give some estimations on the complexity class of the case study itself and suggest some improvements for future cases for tool comparison.

The rest of the paper is structured as follows. Section 2 briefly introduces graph patterns, graph transformation rules, and control structures as available in the VIATRA2 transformation language. Then Section 3 introduces our solution to the case study, while Section 4 focuses on the different pattern matcher strategies provided by the VIATRA2 framework. We present our comparative benchmark results and analysis along with our suggestions for improvement of VIATRA2 and the case study in Section 5. Finally, Section 6 concludes the paper.

2 Background

In order to understand the concepts of VIATRA2 graph transformation environment, we give a brief overview of the metamodeling foundations and transformation language of this framework.

2.1 Metamodeling foundations

The VIATRA2 framework is based on the VPM (Visual and Precise Metamodeling) [17] metamodeling approach, which can support different metamodeling paradigms by supporting (i) multi-level metamodeling with explicit and generalized instance-of relations and (ii) dynamic typing of elements.

The VPM language consists of two basic elements: the entity (a generalization of MOF package, class, or object) and the relation (a generalization of MOF association end, attribute, link end, slot). *Entities* represent basic concepts of a (modeling) domain, while *relations* represent the relationships between other model elements. Furthermore, entities may also have an associated value which is a string that contains application-specific data and generalization is supported by the use of explicit *supertypeOf* relations (similar to the concept of UML).

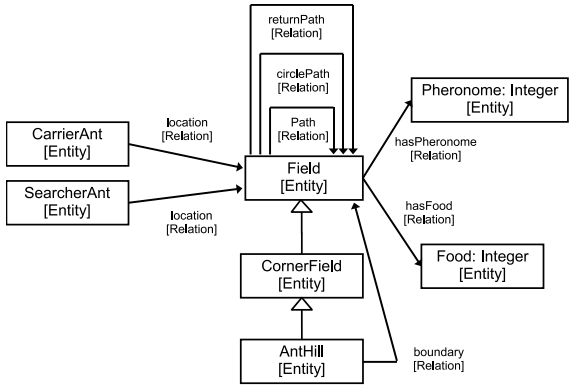


Fig. 1 The AntWorld metamodel

In traditional graph transformation terms, entities can be interpreted as nodes while relations are edges. Entities in a metamodel define node types while entities in models are simply referred to as nodes. In the paper, we use the VIATRA2 terminology for models to avoid the overloading of terms "node" and "edge".

A simplified metamodel of the AntWorld case study used in our experiments, represented in VIATRA2, is shown in Fig. 1. The entity `Field` represents a field of AntWorld (*grid node* in the original specification, but we use the term *field* to avoid confusion); `CornerField` entities are fields that are located on the axis, and the `AntHill` is the central field. Fields are connected by paths. Each circular path formed by `circlePath` relations connects the set of fields that were created in a single round. Except for the anthill, each field has a single outgoing `returnPath` relation pointing towards a field in the previous circle; most fields have a single incoming `returnPath` as well, but corner fields have three, and the anthill has four.

Fields may be associated with an integer number of food items or pheromones associated with them. Finally, a field may contain two kinds of ants: `SearcherAnt` entities represent ants that do not carry food but are in search of a food bundle, while `CarrierAnt` entities represent ants that carry a food item and are on their way to return to the anthill.

2.2 Graph patterns

The transformation language of VIATRA2 (Viatra Textual Command Language – VTCL [16]) consists of several constructs that together form an expressive language for developing both model to model transformations and code generators. Graph patterns (GP) define constraints and conditions on models, graph transformation (GT) [2] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [18] rules can be used for the description of control structures.

Graph patterns are the atomic units of model transformations. They represent conditions (or constraints) that have to be fulfilled by a part of the model space in order to execute some manipulation steps on the model. The basic pattern body contains model element and relationship definitions.

In VTCL, *patterns may call other patterns* using the *find* keyword. This feature enables the reuse of existing patterns as a part of a new (more complex) one. The semantics of this reference is similar to that of Prolog clauses: the caller pattern can be fulfilled only if their local constructs can be matched, and if the called (or referenced) pattern is also fulfilled. For more complex pattern specification the VTCL language also allows to define *alternate* (OR) *pattern bodies* for a pattern, with a meaning that the pattern is fulfilled if at least one of its bodies is fulfilled. A *negative application condition* (NAC, defined by a negative subpattern following the *neg* keyword) prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match. Negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations), where the expressiveness of such patterns converges to first order logic [19].

As an example, the ants that are searching for food, but are not attracted by a pheromone trace, use the `anyNeighborButHome` *graph pattern* to determine which field to move to. This pattern, used to match neighboring fields (excluding the *AntHill*) is shown in Fig. 2.

This pattern uses alternate pattern bodies to represent moving in the forward or reverse direction of a path relation between `Field1` and `Field2`. It also reuses the `home` pattern as its NAC to put the `not AntHill` type constraint on `Field2`.

2.3 Graph transformation rules

Graph transformation (GT) [2] provides a high-level rule and pattern-based manipulation language for graph models. In VTCL, graph transformation rules may be specified by using a *precondition* (or left-hand side – LHS) pattern determining the applicability of the rule, and a *postcondition* pattern (or right-hand side – RHS) which declaratively specifies the result model after rule application. Elements that are present only in (the image of) the LHS are deleted, elements that are present only in the RHS are created, and other model elements remain unchanged. Further actions can be initiated by calling any ASM instructions within the `action` part of a GT rule, e.g. to report debug information or to generate code.

For instance, the GT rule `return` defines how the ants that are carrying food take a step towards the hill, as shown in Fig. 3. The mechanism of leaving pheromones is omitted here for the sake of brevity.

```
pattern home(Field2) =
 {AntHill(Field2);}

pattern
anyNeighborButHome(Field1, Field2) =
{
 field(Field1);
 field(Field2);
 field.path(P, Field1, Field2);
 neg find home(Field2);
} or {
 field(Field1);
 field(Field2);
 field.path(P, Field2, Field1);
 neg find home(Field2);
}
```

Listing 1 VIATRA2 source code for the anyNeighborButHome pattern

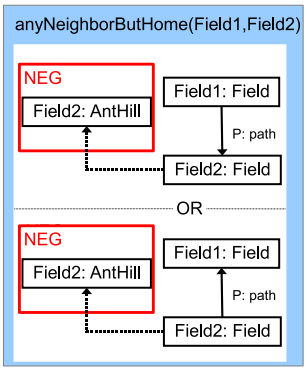


Fig. 2 The AnyNeighborButHome graph pattern

2.4 Model manipulation

The ASM language of VIATRA2 also includes constructs to directly manipulate models from ASM rules. It is important to point out that in our solution to the case study, we have opted to perform the model simulation entirely programmatically, using ASM sequences (instead of declarative GT rules, as shown above), but still relying on graph patterns for preconditions.

```
// Ant returns along returnpath.
gtrule return(in Ant) =
{
 precondition pattern lhs(Ant,
  InnerNeighbor, OuterNeighbor,Loc) = {
   field(InnerNeighbor);
   field(OuterNeighbor);
   field.returnPath(RP,OuterNeighbor,InnerNeighbor);
   carrierAnt(Ant);
   carrierAnt.location(Loc,Ant,OuterNeighbor)
  }
 //Deletes Loc and creates NewLoc
 postcondition pattern rhs(Ant,
  InnerNeighbor, OuterNeighbor,Loc) = {
   field(InnerNeighbor);
   field(OuterNeighbor);
   field.returnPath(RP,OuterNeighbor,InnerNeighbor);
   carrierAnt(Ant);
   carrierAnt.location(NewLoc,Ant,InnerNeighbor);
  }
}
```

Listing 2 VIATRA source code for graph transformation rules

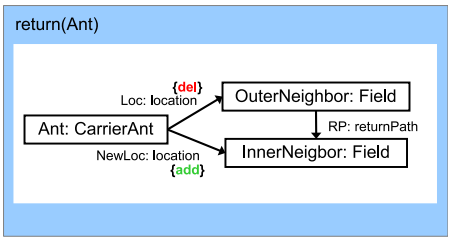


Fig. 3 Graph Transformation rule for ants returning towards the hill

The example code shown in Lst. 3 demonstrates how the sequence of grabbing a food item is defined using ASM model manipulation constructs. First, the remaining amount of food is calculated. If this is positive, the new value of the food bundle is set accordingly; otherwise, the food bundle is exhausted and must be deleted (also deleting implicitly all edges pointing to-or-from the deleted element which in this case is the `hasFood` edge connecting it to the field). Then instead of deleting (and recreating) the ant it is dynamically retyped from a `searcherAnt` to a `carrierAnt`; instantiation relationships can be manipulated much the same way as ordinary model elements. Please note that the `location` edge needs retyping too, since we opted to use different types of location relations for the two ant types, instead of lifting this Relation to the common supertype `ant`.

```
rule grab(in Ant, in LocationEdge, in Food) =
 let Rest = toInteger(value(Food)) -1  in seq{
 if (Rest>0) setValue(Food, Rest); else delete(Food);
 delete(instanceOf(Ant,ants.metamodel.searcherAnt));
 delete(instanceOf(LocationEdge,
  ants.metamodel.searcherAnt.location));
 new(instanceOf(Ant,ants.metamodel.carrierAnt));
 new(instanceOf(LocationEdge,
   ants.metamodel.carrierAnt.location));}
```

Listing 3 VIATRA2 source code for food grabbing

2.5 Control structure

To control the execution order and mode of transformations, *abstract state machines* [18] are used. ASMs provide complex model transformations with all the necessary control structures including the sequencing operator (`seq`), ASM rule invocation (`call`), variable declarations and updates (`let` and `update` constructs), `if-then-else` structures, non-deterministically selecting (`random`) constructs, iterative execution (applying a rule as long as possible (ALAP) `iterate`), the simultaneous rule application at all possible matches (locations) (`forall`) and single rule application on a single match (`choose`).

The example code shown in Lst. 4 demonstrates how typical control structure combinations are used in VIATRA2.

The first `choose` rule tries to find a single match for the `Ant`, `LocationField`, `Food` and `Field` variables (defined in its head), which satisfy the `canGrab` graph pattern, and then executes the `grab` ASM rule. If more variable substitutions satisfy the pattern, then one is chosen non-deterministically and if there are no such substitutions then the `choose` rule *fails*.

Using the `iterate` rule in the example allows to apply its `choose` rule as-long-as-possible (ALAP), i.e. as long as a match for the `canGrab` pattern can be found. In other terms, the `choose` rule is applied on a single non-deterministically selected match followed by its `grab` ASM rule invocation and repeated as long as the `canGrab` pattern can be matched.

As for the following `forall` rule, it finds *all* substitutions (matches) for variables defined in its head (`Ant`, `LocationField`), which satisfy the `hasCarrierAnt` pattern, and then executes the `deposit` ASM rule for each substitution separately. If no variable substitutions satisfy the pattern, then the `forall` rule is still successful and does not fail. In contrast to the `iterate` rule, it first collects all available matches and then applies its ASM rule for each in a single step. Note that the `Hill` variable will have to be defined prior to the execution of the `forall` rule as it is assumed as an input parameter for the `hasCarrierAnt` pattern and is not defined in its head along with the `Ant` and `LocationEdge` variables.

```
iterate choose Ant, LocationEdge, Food, Field with
 find canGrab(Ant, LocationEdge, Food, Field) do
  call grab(Ant,LocationEdge,Food); //grab food
forall Ant, LocationEdge with //desposit on the Hill
 find hasCarrierAnt(LocationEdge, Hill, Ant) do
  call deposit(Hill,Ant,LocationEdge);
```

Listing 4 Example control structure combinations

3 Description of the solution

The AntWorld case study [15] is a model transformation benchmark featured at GraBaTs 2008 [4]. AntWorld, probably inspired by Ant Colony Optimization [20], simulates the life of a simple ant colony searching for food to spawn more ants on a dynamically growing rectangular world. The ant collective forms a swarm intelligence, as ants discovering food sources leave a pheromone trail on their way back so that the food will be easily found again by other ants.

The sequence shown in Lst. 5 defines how one iteration of the AntWorld case study is managed in our implementation. An iteration is divided into seven different phases; four for the ant simulation and three for the world management. All phases are captured by a combination of `forall` and `choose` structures guarded by graph patterns (see in Fig. 4) filtering the input model parameters of their invoked ASM rule. How each phase manages its task is described in the following Sec. 3.1 and Sec. 3.2 for the ant and world simulation phases, respectively.

```
rule doRound()=let Hill =ants.model.hill in seq {
//Ant actions
iterate choose Ant, LocationEdge, Food, Field with
  find canGrab(Ant, LocationEdge, Food, Field)
    do call grab(Ant,LocationEdge,Food); //grab food
forall Ant, LocationEdge with //desposit on the Hill
  find hasCarrierAnt(LocationEdge, Hill, Ant)
    do call deposit(Hill,Ant,LocationEdge);
forall Ant, FromField, LocationEdge with // return
 find hasCarrierAnt(LocationEdge, FromField, Ant) do
  choose NewField with
   find alongReturnPath(FromField, NewField) do seq {
    call moveAnt(HA1, NewField);
    call leavePheromone(FromField);
 }
forall Ant with find searcher(Ant) //search for food
  do call search(Ant); // both kinds of search
//World management
forall Pheromone with find pheromone(Pheromone)
  do call evaporate(Pheromone); //evaporate pheromone
iterate //create new ants
  if(toInteger(value(Hill)) > 0) call consume(Hill);
  else fail; // until all deposited food is consumed
//only searchers can breach the boundary!
if (find boundaryBreachedBySearcher())
  call growGrid(); //grow the game map
}
```

Listing 5 VIATRA2 source code for an iteration

3.1 Ant simulation

Grab phase: First, the food gathering is managed by an ALAP execution of the `grab` rule guarded by the `canGrab` (see in Fig. 4(a)) pattern that iterates over all searcher ants standing on a food bundle. This way for each ant standing on a food pile the `grab` ASM rule calculates the remaining amount of food. If it is positive, the new value of the food bundle is set accordingly otherwise, the food bundle is exhausted and deleted from the model. The actual model manipulation operations are detailed in Sec. 2.4.

Deposit phase: The `hasCarrierAnt` pattern (depicted in Fig. 4(d)) with the `Field` parameter bound to the anthill in a `forall` construct identifies all carrier ants that have successfully delivered a food bundle to the hill. The `deposit` rule increases the integer value of the hill representing the actual number of food bundles on the hill and dynamically changes the type of its `Ant` input parameter from `CarrierAnt` to `SearcherAnt` along with its `LocationEdge` parameter in the inverse way as it is done in the `grab` rule.

Return phase: In this phase all carrier ants that did not reach the hill yet, will step one field closer to home along the `returnPath` relation. Their next position `NewField` is determined by the `alongReturnPath` pattern (depicted in4(c)) used with a `choose` structure while all the carrier ants are iterated over by the outer `forall` with the `hasCarrierAnt` (see in Fig. 4(d) pattern. The concrete model manipulations are carried out by the `moveAnt` and the `leavePheromone` rule. The `moveAnt` rule is only a single operation that sets the target of the `OldLocation` edge to its `NewField` input parameter. While the `leavePheromone` leaves a pheromone on its `Field` input parameter. To do so it first checks that the input `Field` already has a `Pheromone` using a `try-else` control structure combined with a `choose` invoking the `hasPheromone` pattern. If it has, then simply add 1024 to its integer value, otherwise the `else` branch executes attaching a newly created `Pheromone` with 1024 as its value to the `Field`. Note that, because the *deposit phase* was already executed in the current iteration there are no carrier ants standing on the anthill for which the `hasCarrierAnt` pattern could be reused.

Search phase: Finally, searcher ants looking for a food source are actuated by the `search` rule (handling both the unguided and pheromone-guided cases) executed in a `forall` construct using the `searcher` (see in Fig. 4(f)) pattern. The `search` rule retrieves the `Field1` field on which the input `Ant` is standing. Additionally, it checks if there is any pheromone infested field neighboring `Field1` using the `attractingOuterNeighbor` pattern in a `try-else` construct invoked by a choose rule. If there is, then it steps to that, otherwise to one of the neighboring fields (except the anthill) selected by the `anyNeighborButHome` pattern (detailed in Sec. 2.2). Both patterns are matched in a true pseudo-random fashion defined by the `@Random` annotations.

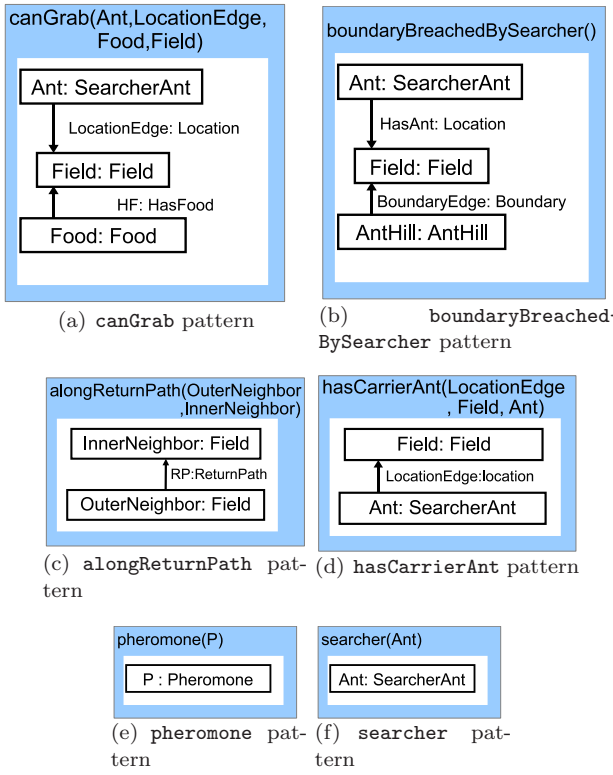
3.2 World management

Evaporate Pheromone phase: To volatilize the `Pheromones` in the model the simple `pheromone` pattern (see in Fig. 4(e)) in a `forall` construct invoking the `evaporate` rule is used. In the `evaporate` rule first the remaining amount of pheromone is calculated. If this is positive, the new value of the pheromone is set accordingly; otherwise the pheromone is exhausted and deleted from the model. The calculation is kept in the integer domain using the JAVA built in rounding mechanism on the division operator.

Create Ants: As the number of food bundles on the anthill is managed by the integer value of the hill itself, the creation of the ants does not involve pattern matching. It is handled using an `iterate` construct which executes the `consume` rule as-long-as the integer value of the hill is higher than zero and terminates the loop with the `exit` command. At every invocation of the `consume` rule it decrements the integer value of the hill by one and creates a new `SearcherAnt` with its `location` pointing to the anthill.

Boundary Breached phase: Finally, in order to check that a searcher ant has reached the boundary of the actual world an `if` construct is used to check that the `boundrayBreachedBySearcher` (see in Fig. 4(b)) pattern matches to the actual model. If it matches the `growGrid` rule is invoked to handle the expansion of the world. The algorithm used is a circular based traversal of the boundary fields along the `circularPath` starting from a randomly selected border field. During the traversal for each boundary field a new outer neighbor is created connected to its neighboring fields along with the update of the `boundary` relation from the hill. The only exemption from this rule are the `CornerFields` where three new (a `CornerField` and two simple `Field` fields along with their relations between them and the `boundary` relation to the hill) fields are generated to create the new corner of the actually constructed boundary. This way the distribution of the newly created food bundles are also arranged during the traversal and on every tenth newly created boundary field an additional `Food` bundle is added.

The complete source code of the case study is available in Appendix A.



(a) `canGrab` pattern (b) `boundaryBreachedBySearcher` pattern (c) `alongReturnPath` pattern (d) `hasCarrierAnt` pattern (e) `pheromone` pattern (f) `searcher` pattern

Fig. 4 Patterns used in the `doRound` rule

4 Pattern matching strategies in VIATRA2

4.1 Pattern matching strategies in the VIATRA2 framework

Pattern matching plays a key role in the efficient execution of all model transformation engines. In case of graph transformation based approaches, the goal is to find the occurrences of a graph pattern, which contains structural as well as type constraints on model elements. During pattern matching, each variable of a graph pattern is bound to a node in the model such that this matching (binding) is consistent with edge labels, and source and target nodes of the model.

Most graph transformation approaches (e.g. [6,21, 22,5] and many more) usually rely on a *local search based pattern matching* (LS) that starts the matching process from a single node and extends it step-by-step by neighboring nodes and edges.

As an alternatative approach, *incremental pattern matching* (INC) [8,9,11,12] relies on a *cache* in which the matches of a pattern are stored explicitly. Its match set is readily available from the cache at any time without searching, and the cache is incrementally updated whenever changes are made to the model.

As an important language feature, the ASM machine defining the entire transformation, as well as individual patterns, can be annotated with special information on how they should be treated. For example, patterns marked with `@Random` will select a match in a true pseudo-random fashion when used in a `choose` rule, as required by the AntWorld specification. Furthermore, our solution relies on explicitly specifying the desired pattern matching strategies (see Sec. 5.1.1), which is selected by annotating the machine with `@localsearch` or `@incremental`; the decision can be locally overridden on a per-pattern basis with the same annotations.

How these two fundamentally different approaches are implemented in the VIATRA2 framework is briefly introduced in Sec. 4.2 and Sec. 4.3, respectively. However there are cases where the use of neither the incremental nor the local search based pattern matching approach is significantly more efficient than the other. We argue that many transformations could benefit even more from combining these two approaches, by using different pattern matcher engines for different graph patterns. How the combination of these different pattern matching strategies (referred to as *hybrid pattern matching*) within a transformation is possible in VIATRA2 is briefly introduced through the AntWorld case study in Sec. 4.4.

4.2 Local Search based Pattern Matching in VIATRA2

The generation of search plans [23, 24] is a frequently used and efficient strategy to drive the execution of LS pattern matching algorithms. Informally, a *search plan* defines the order in which pattern nodes are bound to objects of the instance model during pattern matching. In addition to simply specifying the binding order of pattern nodes, it often also includes an order of elementary operations that have to be executed to drive pattern matching.

The LS graph pattern matcher of VIATRA2 follows the same approach. Without going into technical details, our approach consists of the following steps (see in Fig. 5):

First, we separate compile time parts from run-time parts, where each part consists of the following steps:

Compile time At compile time each step is calculated once for each pattern description.

- First, for each pattern description a *call tree* is generated capturing how patterns call other patterns. A call tree is a directed bipartite tree describing the structural dependencies of a given pattern by encapsulating the alternative pattern bodies and pattern invocations.

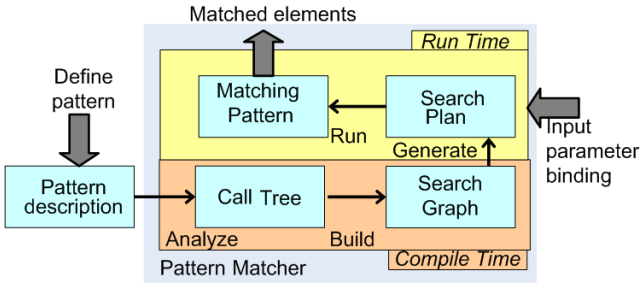


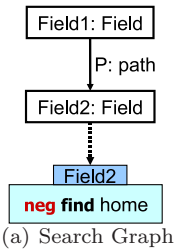
Fig. 5 The Overview of the VIATRA2 LS pattern matcher

- Then for each *call tree* a corresponding *search graph* is generated. A search graph is a joint representation of pattern graph elements and operation constraints that drives the pattern matching process. In our interpretation a search graph is a hypergraph [24] representing a constraint net, where graph nodes reflect variables, and hyperedges express constraints (predicates) between the variables. In order to yield better search plans, the operation scope of the optimizer module is increased by flattening the call tree and by merging pattern bodies and pattern invocations into a common *search graph*. This allows the use of our optimization techniques on a global scale rather than on isolated pattern bodies.

Run time After initializing the data structures at compile time, run time steps have to be calculated for each separate pattern invocation.

- A *search plan* is generated from the *search graph* based on the input parameter binding and the cost of search operations to drive the pattern matching process. A search plan is a totally ordered list of search operations (one possible traversal of the search graph), where search operations represent the atomic units of pattern matching (a single step in the matching process). It is either an extend operation which extends the matching by a new element (e.g match the target node along an edge), or a check operation used for checking constraints between pattern elements (e.g., whether an edge runs between two nodes).
- Finally, after executing the search plan matches relevant to the input parameter binding are passed out.

As an illustration, Fig. 6(a) shows the *search graph* built for the first pattern body of the `anyNeighborButHome` pattern depicted in Fig. 2. It is a very simple search graph containing only two nodes `Field1` and `Field2` connected by the `P` relation with the `home` NAC invocation with `Field2` as its input parameter.



(a) Search Graph

operation	type
1: Field1 instance of Field	check
2: source of P is Field1	extend
3: P instance of Path	check
4: target of P is Field2	extend
5: Field2 instance of Field	check
6: nac home invocation	check

(b) Search Plan for bound Field2

Fig. 6 Search Graph and Plan for the first pattern body of the anyNeighborButHome pattern

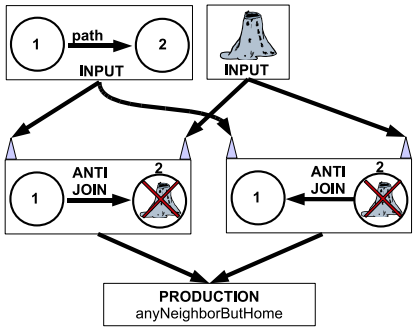


Fig. 7 Simple RETE matcher

Fig. 6(b) shows a possible *search plan* generated from the search graph with `Field1` considered as an input parameter. The search plan extends the already bound `Field1` to `Field2` through the `P` relation and checks that all newly matched element has their appropriate type and finally invokes the `home` pattern as a NAC. A more detailed description how the LS pattern matcher of the VIATRA2 framework works is given in [24].

Overall, the VIATRA2 LS strategy can produce reasonable performance with a relatively small memory footprint, although adaptive graph pattern matching using run-time model sensitive search optimization [25] are not yet supported.

4.3 RETE-based incremental graph pattern matching in VIATRA2

Incremental pattern matching [9] offers an entirely different execution model compared to local search-based implementations. The match sets for all patterns involved in the graph transformation are computed in an initialization phase prior to execution (e.g. when the model itself is loaded into memory), and as the transformation progresses, this match set cache is incrementally updated as the model graph changes (update phases). Thus, model search phases are reduced to fast read-from-cache operations, in exchange for the overhead imposed by cache update phases which occur synchronously with model manipulation operations. Benchmarking [13] has shown that in certain scenarios, this approach leads to several orders-of-magnitude increases in speed.

The incremental graph pattern matcher of the VIATRA2 framework adapts [9] the RETE algorithm [26], which is a well-known technique in the field of rule-based systems.

RETE net for graph pattern matching. RETE-based pattern matching relies on a network of nodes storing *partial matches* of a graph pattern. A partial match enumerates those tuples of model elements which satisfy a subset of the constraints described by the graph pattern. In a relational database analogy, each node stores a *view*. Matches of a pattern are readily available at any time, and they will be incrementally updated whenever model changes occur.

Input nodes serve as the underlying knowledge base representing a model. There is a separate input node for each entity type (class), containing a view representing all the instances that conform to the type. Similarly, there is an input node for each relation type, containing a view consisting of tuples with source and target in addition to the identifier of the edge instance.

At each *intermediate node*, *set operations* (e.g. filtering, projection, join, etc.) can be executed on the match sets stored at input nodes to compute the match set which is stored at the intermediate node. The match set for the entire pattern can be retrieved from the final *production node*. One kind of intermediate node is the *join node*, which performs a natural join on its parent nodes in terms of relational algebra; whereas an *anti-join node* contains the set of tuples stored at the primary input which do *not* match any tuple from the secondary input (which corresponds to anti-joins in relational databases).

As an illustration, Fig. 7 shows a RETE network matcher built for the `anyNeighborButHome` (see Fig. 2) pattern illustrating the use of anti-join nodes for NAC. By anti-joining two input nodes (the top-most nodes on Fig. 7), this sample RETE net enforces a relation type constraint (`path` relation type connecting two fields, see left input node) and the non-satisfiability of an entity constraint (anthill type, see right input node). To ensure that the directed path edges can be traversed in both directions, two opposite directions of the `path` edge are checked in two separate pattern bodies; the final production node contains a union of the two cases.

Updates after model changes. Input nodes receive notifications about each elementary model change (e.g. when a new model element is created or deleted) and release an update token on each of their outgoing edges. Such an update token represents changes in the partial

matches stored by the RETE node. Positive update tokens reflect newly added tuples, and negative updates indicate tuples being removed from the set. Upon receiving an update token, a RETE node determines how the set of stored tuples will change, and release update tokens of its own to signal these changes to its child nodes. This way, the effects of an update will propagate through the network, eventually influencing the result sets stored in production nodes.

The match set can be retrieved from the network instantly without re-computation, which makes pattern matching very efficient. As a trade-off, there is increased memory consumption, and update operations become more complex.

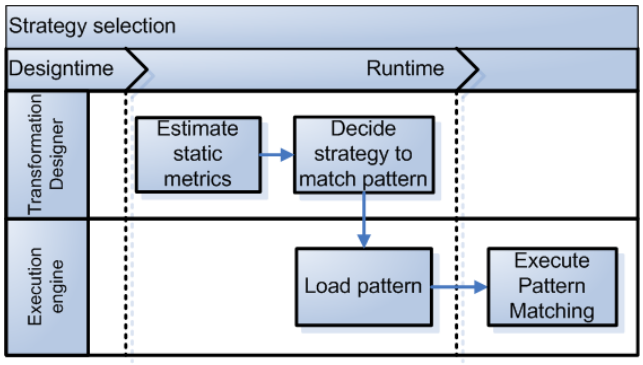


Fig. 8 Selecting pattern matching strategies

4.4 Hybrid pattern matching strategy

Recent benchmarks evaluations [13] and tool contests [4] in the graph transformation community have shown that INC can be order(s) of magnitude faster than LS approaches for certain problem classes. There are also other cases where the use of local search based pattern matching approach is significantly more efficient on memory consumption than any other. We believe that many transformations could benefit even more from combining these two approaches to use the most suiting pattern matcher engine for each graph patterns.

In the VIATRA2 framework, a transformation designer can fine-tune the performance or memory consumption of graph pattern matching by prefixing it with `@localsearch` or `@incremental` annotations to select the designated pattern matching strategy. This way the interpreter automatically uses the defined pattern matcher during the transformation execution. This feature also holds for composite patterns which allows the definition of different matching strategies for certain parts of the pattern. This way the search plan generated for these composite patterns are optimized to favor (already) incrementally matched patterns traversal in the early steps of the matching process to bind elements for the later LS matched part. The same algorithm as for LS is used to generate these search plans. It differs only in two parts: (i) the flattening process is not invoked on the incrementally matched patterns and (ii) during execution the incrementally matched pattern invocations are transformed into one search operation that bound its interface symbolic parameters from its cache. The high-level workflow of this technique is illustrated in Fig. 8.

To illustrate how hybrid pattern matching is performed in VIATRA2, Listing 6 shows the `attractingOuterNeighbor` pattern composed of the `attractingField` and `alongReturnPath` patterns defined to be matched by INC and LS, respectively. The pattern is used to match any attracting (has pheromone) neighboring field of `Field1` that is leading to a food source (away from the hill). The idea behind using hybrid approach in this case comes from following considerations: (i) the `alongReturnPath` pattern matches to all neighboring fields that consume a large amount of memory if incrementally cached, (ii) a pure LS approach would have to go through all neighboring fields and check if they hold pheromone leading to a relatively low performance and (iii) as `Field1` is normally an input parameter of the `attractingOuterNeighbor` pattern and `Field2` is already incrementally cached the `alongReturnPath` pattern needs only to check if there is a `returnPath` edge between its input parameters. The search plan generated for the `attractingOuterNeighbor` with `Field1` considered as an input parameter is depicted in Fig. 9.

```
@localsearch //hybrid pattern matching
pattern attractingOuterNeighbor(Field1, Field2) = {
 find alongReturnPath(Field2, Field1); // LS
 find attractingField(Field2); // INC
}
@incremental
pattern attractingField(Field) = {
 field(Field);
 field.hasPheromone(HF, Field, Pheromone);
 pheromone(Pheromone);
 check(toInteger(value(Pheromone)) > 9);
}
@localsearch
pattern alongReturnPath(OuterNeighbor,InnerNeighbor)=
{
 field(InnerNeighbor);
 field(OuterNeighbor);
 field.returnPath(RP, OuterNeighbor, InnerNeighbor);
}
```

Listing 6 VIATRA2 source code for the attractingOuterNeighbor composite pattern

Based on our previous experience [13,14], we identified the following factors to be important in general for transformation designers to choose between LS and INC strategies:

operation	type
1: Field1 instance of field	check
2: attractingField invocation (incremental)	extend
3: Field2 instance of field	check
4: source of RP is Field2	extend
5: RP instance of returnPath	check
6: target of RP is Field1	check

Fig. 9 Search Plan of the `attractingOuterNeighbor` pattern

Static attributes of graph patterns: One of the most important factor, in the sense of memory consumption, is the *number of graph patterns* in a transformation program. The cache size of a pattern increases overall memory consumption when matched by INC strategy. However, in practical applications, we experienced that the number of matches gradually decrease as the pattern to be matched becomes more and more complex (having more and more elements). This contradicts the intuition that larger patterns will have more matches due to more combinatorial possibilities. Although this combinatorial increase may hold for smaller patterns, it is overwhelmed by the scarcity due to restrictiveness of larger patterns in many practical scenarios. As a result, large patterns should be preferably matched by INC and in case of large number of patterns smaller ones should be by LS.

Control structure How patterns are used and invoked in a transformation program has a huge impact on overall performance and can greatly influence the cost of pattern matching. *Usage frequency* of patterns is relevant, since the more often a pattern is used, the more advantage INC has. Frequently used patterns can be identified by static analysis of the transformation code, e.g. by marking patterns that are used from within a loop. Another significant factor can be *parameter passing*, i.e. to reuse the result of other rules or patterns as an input. This technique increases efficiency in LS as search operations are much more efficient if one or more pattern variables are bound, i.e. their values are known at time of the query. INC performance is not affected.

Model-specific graph characteristics Ultimately the underlying model determines all performance characteristics. In order to indicate its effect on each pattern we defined a simple scalar metric called *node type complexity*. It is a rough upper bound on the number of potential matches can be obtained as the product of the cardinalities (number of model instances) of the types of each node in the graph pattern. This estimate is, of course, accurate as there are also edges in the pattern to constrain the possible combinations of nodes. However, high complexity may result in high memory consumption for INC, and long search operations for LS.

A more detailed investigation how relevant factors influence pattern matcher selection is available in [1]. How we specified our hybrid implementation is discussed in Sec. 5.1.

In overall a well defined hybrid approach can usually largely reduce memory consumption within reasonable run-time performance degradation.

5 Benchmarking

In this section, we present our experiments to assess VIATRA2's performance on the AntWorld case study. Our main goal with benchmarking is two-fold: (i) to demonstrate how the performance of VIATRA2 evolved with the incremental pattern matching approach (Sec. 4.3), and (ii) to present some useful design-time optimizations and fine-tune options in Sec. 5.1 which can have significant impact on performance.

5.1 Fine-tuning

Based on the involved segment of VIATRA2 we categorized our optimizations into three categories: (i) pattern matching strategy selection (see Sec. 5.1.1), (ii) advanced model management application(see Sec. 5.1.2) and (iii) language specific consideration (see Sec. 5.1.3).

5.1.1 Pattern matching strategy

We designed our implementation to effectively support a hybrid pattern matching approach (see Sec. 4.1) that trades runtime performance for memory consumption compared to the pure incremental solution. This hybrid solution was based on the following considerations:

- Considerable memory can be saved by ensuring that the map (fields and path relations) is not contained in the RETE net, as these are the types with the highest number of instances. Patterns concerning these model features should be assigned to the local search based matcher, to keep the RETE net small. As these patterns happen to establish simple local relationships of low complexity, they are efficiently matched using the local search based engine.
- To achieve high performance through avoiding expensive repeated searching, the incremental pattern matcher was selected to deal with the `hasFood`, `location`, `hasPheromone`, `boundary` relations. This allowed useful collections such as ants stumbling upon

food, ants reaching the boundary, or pheromones that are still strong enough to attract ants to be incrementally maintained.
– Some patterns contain model features of both kinds. On certain occasions, a subpattern was extracted for the incremental pattern matcher, and the local search based matcher utilized this cache in a true hybrid fashion. See Lst. 6 as an example.
– The design of patterns and the metamodel had to support efficient division of pattern matching tasks between the two matching strategies. Fig. 1 shows that the VIATRA2 solution uses a relation type `boundary` connecting the anthill to exactly those fields that are on the boundary of the currently explored grid. This relation identifies boundary fields, which is useful when determining whether the grid needs to be expanded by another circle, and also during said expansion. The motivation for introducing it into the solution was to enable these parts of the transformation to be efficiently executed using the local search based pattern matcher, effectively reducing the size of the incremental pattern matcher, making the hybrid solution a viable compromise. It is important to point out that had we relied entirely on the incremental pattern matcher, using these `boundary` relations would have become unnecessary, as path relations would have been admissible in the RETE net, and boundary fields would have been efficiently expressible as a pattern with a NAC (see Lst. 7).

```
@incremental
pattern boundary(Field) = {
 field(Field);
 neg pattern nonBoundary(Field) = {
  field(Field);
  field(OuterNeigbor);
  field.returnPath(RP, OuterNeigbor, Field);
 }
}
```

Listing 7 Identifying boundary fields without relying on an explicit marking

5.1.2 Model management issues

VIATRA2 is an interpreted model transformation engine, and has a generic reflective model representation. Among many other features, model elements are allowed to have multiple types at once, instantiation is expressed as an explicit relationship between type and instance model elements, and this type information can be manipulated at runtime. In order to facilitate multi-level metamodeling, every model element is allowed to act as a type. Moreover, both relations and attributes are first-class model elements that can be the sources and targets of relations and have fully qualified names in the hierarchical namespace scheme of VIATRA2.

Although providing great flexibility, these features of the model representation make model elements relatively heavy-weight. Therefore this approach has a negative effect on performance, that should be taken into consideration when designing the transformation. As a simple example, model elements of special significance (e.g. the anthill) can be looked up at the beginning of the transformation, and later retreived from a cache whenever needed. This avoids the cost associated with accessing a specific model element identified by its fully qualified name.

Moving an ant involves pointing its `location` relation to its new position. Instead of deleting and recreating `location` relations (which involves the deletion and creation of `instanceOf` relationships and other administrative data, described in Sec. 2.4), we merely change the target end of the relation. This simplification helps to reduce the amount of model manipulation. When growing the grid and expanding its boundary, `boundary` relations are reused in a similar fashion.

As seen in Fig. 1, the type of ant (searcher / carrier) is not represented by graph elements or attributes, but by using two disjoint entity types. This choice was made to reduce the number of model elements, as no further attributes or connected model elements are required to express the type of an ant, which is an entity with a high number of instances. Changing ants from one type to another is achieved by dynamic retyping (see Lst. 3).

5.1.3 Language-specific considerations

As described in Sec. 2.3, transformation semantics can be specified using the well-known graph transformation formalism. Our solution takes a slightly different approach: the precondition (LHS) patterns of the GT rules are kept intact, but instead of specifying the action declaratively by a RHS, model manipulation is given as an imperative sequence, using the ASM language of VIATRA2. The foremost benefit of this choice is that the transformation is able to take advantage of some more advanced model manipulation operations, such as the ones needed by the methods described in Sec. 5.1.2. Additionally we expected that this imperative language usage itself has a noticeable performance advantage, because the declarative GT rule specification may imply some expensively checked type constraints that may be unnecessary and can be omitted in an imperative rule definition.

5.2 Measurements

We conducted benchmark measurements on our test system with a quad-core Intel Xeon CPU clocked at 2.00 GHz and 12 GBs of system memory. We used the OpenJDK 64-bit Server VM (IcedTea6 1.3.1 build 12) on Linux 2.6.18 with 10GBs of memory allocated to the JVM.

5.2.1 Variants

We divided our experiments into two groups: the first group was performed to demonstrate the difference between various pattern matching strategies (Sec. 5.1.1), while the second was aimed at illustrating the effects of fine tuning described in Sec. 5.1.2 and Sec. 5.1.3.

For the first group, we configured the transformation program (available in Appendix A) with annotations to create the following run configurations:

1. the *local search solution* made exclusive use of the traditional, local search-based pattern matcher implementation described in Sec. 4.2.
2. in contrast, the *incremental solution* relied solely on the RETE-based pattern matcher described in Sec.4.3.
3. finally, we combined the pattern matching strategies with techniques described in Sec. 4.4 and 5.1.1 to create a *hybrid solution*.

For the second group, the following run configurations were created:

1. in order to illustrate the attainable performance gain by avoiding expensive model management operations, we compared an *unoptimized* variant (which did not make use of dynamic typing and relation retargeting as described in Sec. 5.1.2) to the *optimized* variant which incorporated both.
2. finally, we designed two variations to determine the performance impact of a language-specific optimization which involves using imperative model manipulation rules instead of purely declarative graph transformation rules (Sec. 5.1.3).

5.2.2 Telemetry

To obtain numeric results, we designed the simulation transformation to generate XML output containing execution time and memory usage telemetry data.[1] Every 25 rounds, telemetry data was written to an output buffer, which was flushed to a file after the transformation has terminated.

Overall, we executed five 500-round simulation runs for each variant, with the exception of the local search solution where only 150 rounds were executed (since it is significantly slower than the other two variants). Memory consumption measurements were performed in separate execution runs to avoid a potential negative performance impact.

[1] Execution time was measured by the `System.currentTimeMillis()` Java call, while heap usage was estimated by performing garbage collection calls (`System.gc()`) and recording the result of `Runtime.totalMemory() - Runtime.freeMemory()`.

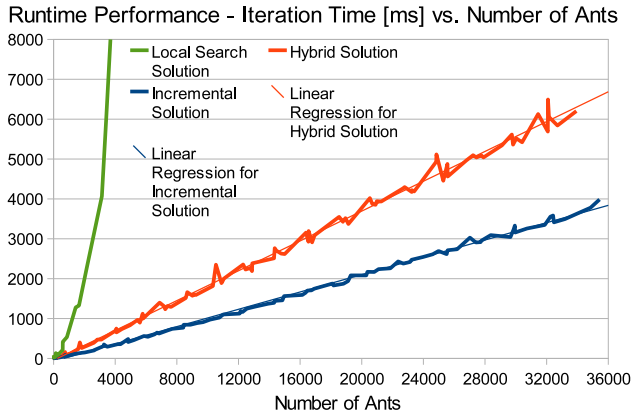


Fig. 10 Execution Time per Iteration

5.3 Analysis of the results

Results were analyzed by transforming the XML output to CSV spreadsheets which were processed in OpenOffice.Org 3.0. We combined the results from each of the five separate execution runs to create a data series consisting of 100 records for INC and HYB (30 records for LS).

5.3.1 Complexity class analysis

By looking at the data, we found that there is a very high correlation (correlation coefficient $R^2 > 0.995$) between the time needed to execute a round and the number of ants (Fig. 10). There is also a fairly high correlation ($R^2 > 0.97$) between the number of fields in the grid and the measured memory consumption (Fig. 14). The number of rounds, however, has a significantly weaker correlation ($R^2 < 0.9$ for some solutions) with both the round time and the memory footprint size. Thus, we generated charts which show cumulative and per-round execution times against the number of ants, and heap usage compared to the number of fields.

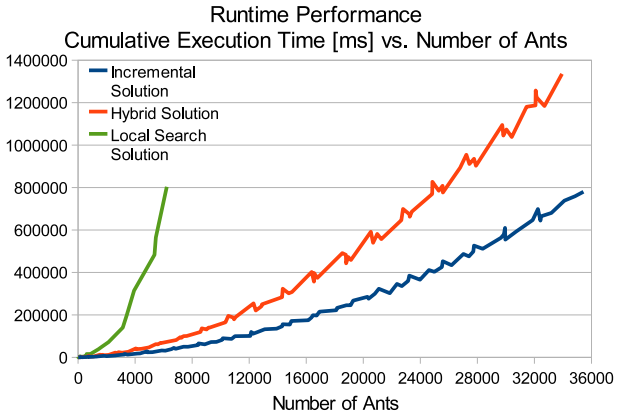


Fig. 11 Cumulative Execution Time

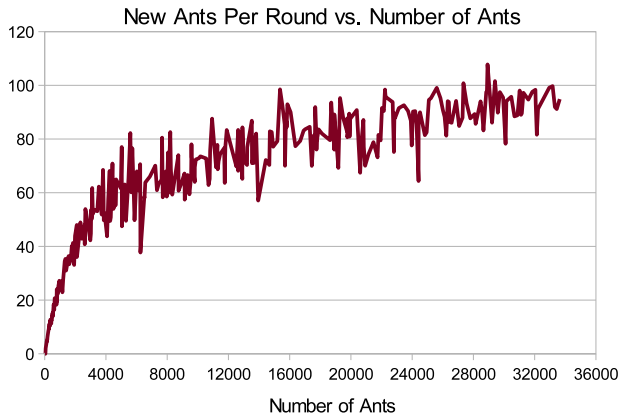


Fig. 12 New Ants Per Round

The cumulative execution time chart with linear scales is shown in Fig. 11. While the local search variant exhibits a high-order polynomial increase as the ant population is growing, both the pure incremental and hybrid variants perform significantly better, following a low-order polynomial characteristic.

In order to determine the polynomial order more precisely, we conducted the following analysis. In the followings, we follow the Landau notation [27] to describe asymptotical limiting behaviour of characteristic functions.

First, we split cumulative execution time into the cumulative time required to simulate the behaviour of the ants, the cumulative time required to grow the grid, and the cumulative time consumed by dropping and evaporating pheromone traces. The dropping of pheromone is included in the pheromone time, not in the ant management time. Formally,

$$Time = Time_{Ants} + Time_{Area} + Time_{Pheromones} \quad (1)$$

where the cumulative time spent on building the grid, $Time_{Area}$, should be intuitively proportional to the grid size with any efficient implementation:

$$Time_{Area} \sim Area \quad (2)$$

The lower bound of the time consumed by pheromone management is approximated by the total number of times pheromones were dropped. This is also an upper bound of the total pheromone management time with an appropriate constant coefficient, because if pheromone is left on a new field, its evaporation will have to be simulated once each round, and there will be a constant number of rounds before it vanishes. Even if pheromone is dropped on the same field multiple times[2], this evaporation cost will be sub-additive, as the pheromone trace containing the combined amount will still evaporate only once per round, and the exponential decay lends it a sub-additive lifespan. Consequently,

$$Time_{Pheromone} \in \Theta(PheromoneDroppings) \quad (3)$$

In our experiments, we plotted the ant population increase against the size of the population in Fig. 12. The plot shows an approximately square root-type upper bound for the increase, in harmony with the following theoretical considerations. In order to give birth to the nth ant (excluding the initial 8), the colony needed to gather n food units. As food is distributed proportionally to the grid area, it follows that the discovered area defines an upper bound to the number of ants:

$$Ants \in O(Area) \quad (4)$$

The area is a quadratic function of the radius of the map, therefore the nth food unit, giving birth to the nth ant, needs to be delivered from a distance of at least $\sqrt{n}$ with some constant multiplier (let us neglect the fact that the order of food units may actually vary). Pheromones are dropped on each step, therefore the spawning the nth ant involves at least $\sqrt{n}$ pheromone droppings; formally, this can be expressed as follows:

$$\frac{\delta PheromoneDroppings}{\delta Ants} \in \Omega(Ants^{0.5}) \quad (5)$$

By integrating with respect to $\delta Ants$, we obtain the following:

$$PheromoneDroppings \in \Omega(Ants^{1.5}) \quad (6)$$

As previously established, the birth of the nth ant requires retrieving food along a path having a length of at least $\sqrt{n}$; since at most n ants are distributed along this retrieval path, and each ant can make one step each round, the birth rate per round can be *approximated* by

[2] this phenomenon is actually very common, as several thousand ants may retrieve food along the same path; our experiments suggest that the number of individual fields with pheromone traces tends to stay relatively low

an upper bound of $\sqrt{n}$ (Fig. 12). This observation can be formalized as follows:

$$\frac{\delta Ants}{\delta Rounds} \in O(Ants^{0.5}) \quad (7)$$

Thus, we are looking for the expression for the cumulative time spent for ant management as the function of the size of the ant population. Its rate of change is expressed as follows:

$$\frac{\delta Time_{Ants}}{\delta Ants} = \frac{\delta Rounds}{\delta Ants} \times \frac{\delta Time_{Ants}}{\delta Rounds} \quad (8)$$

As moving each ant in a round takes a constant-bounded time with an efficient implementation (and potentially more with an inefficient implementation),

$$\frac{\delta Time_{Ants}}{\delta Rounds} \in \Omega(Ants) \quad (9)$$

holds and by substituting Equation 7 into Equation 8, we get Equation 10:

$$\frac{\delta Time_{Ants}}{\delta Ants} \in \Omega(Ants^{-0.5} \times Ants) = \Omega(Ants^{0.5}) \quad (10)$$

By integrating with respect to $\delta Ants$, we obtain Equation 11:

$$Time_{Ants} \in \Omega(Ants^{1.5}) \quad (11)$$

The area management component of the total time can be approximated by combining Equation 2 and Equation 4:

$$Time_{Area} \in \Omega(Ants) \quad (12)$$

The following lower-bound approximation holds for the cumulative pheromone management time, as implied by Equation 3 and Equation 6:

$$Time_{Pheromone} \in \Omega(Ants^{1.5}) \quad (13)$$

Finally, from Equation 1, Equation 11, Equation 12 and Equation 13, we have an estimation of the time complexity of AntWorld simulation with respect to the number of ants:

$$Time \in \Omega(Ants^{1.5} + Ants + Ants^{1.5}) = \Omega(Ants^{1.5}) \quad (14)$$

In reality, the ants do not follow an optimal strategy for exhaustively retrieving all food available within the discovered radius, but rather they are diverted by pheromones towards the direction of previously found food bundles, distorting the circularity of explored area, and needlessly expanding the grid. Also, a number of ant steps are wasted during the search for new food sources. Consequently, the boundary expressed in Equation 4 turns out to be weak; according to regression calculations performed on our measurements, the number of ants seem to be proportional to the area to the power of approximately 0.66. This also means that

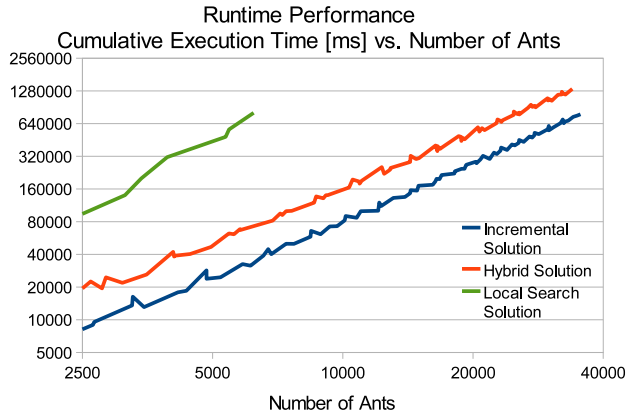


Fig. 13 Cumulative Execution Time (double logarithmic scale)

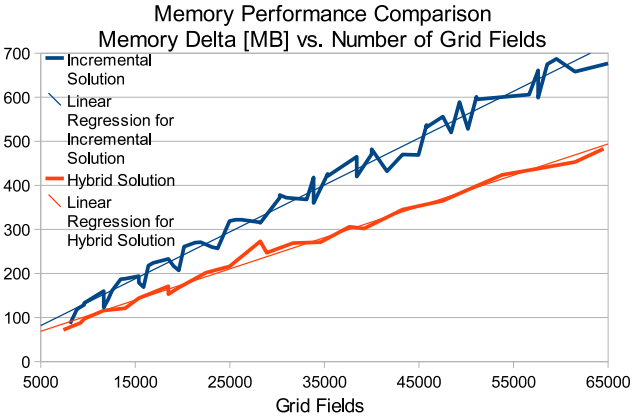


Fig. 14 Memory Delta

ants have a smaller birthrate than allowed in Equation 7, and therefore Equation 10 will not give a close approximation of the time spent on ant management. Finally, as even Equation 12 gives a weak boundary, the total time may have a complexity higher than $Ants^{1.5}$. Our experiments confirm this assumption: for rounds 100-500, regression gave the approximation of $Time \sim Ants^{1.68}$ with a correlation over 99% for our solution (the first 100 rounds appeared more random and less characteristic). Nevertheless, this is still a low-order polynomial behaviour, see Fig. 11 for the results; the complexity is visually confirmed by using logarithmic scales for both axes (Fig. 13).

5.3.2 Effects of optimizations

Hybrid pattern matching Fig. 14 shows memory consumption data comparing pure incremental pattern matching with our hybrid approach. In both cases, the overall heap consumption of the VIATRA2 engine grows linearly with the number of grid fields, however, the gradient for the hybrid run is lower (for a given number of

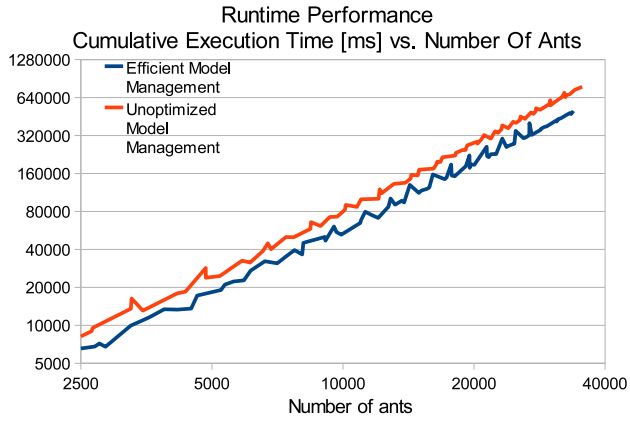


Fig. 15 Cumulative Execution Time (double logarithmic scale) and Model Management Optimizations

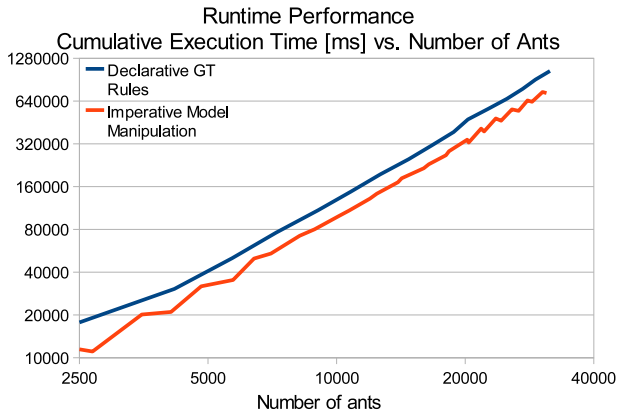


Fig. 16 Cumulative Execution Time (double logarithmic scale) and Language-Specific Optimizations

fields, the pure incremental variant consumes approximately 1.5 times more memory than the hybrid variant). Since the execution time per iteration values are also linear for the hybrid variant (Fig. 10), it can be concluded that the hybrid pattern matching approach performs in the same complexity class as the pure incremental version. In other words, for a linear decrease in memory consumption, a linear decrease in execution speed can be expected (as supported by the constant difference in the logarithmic plot in Fig. 13).

Model management-specific optimizations Fig. 15 shows the performance gain attained by avoiding expensive model management operations. We compared an *unoptimized* variant (which did not make use of dynamic typing and relation retargeting as described in Sec. 5.1.2) to the *optimized* variant which incorporated both. As the plots follow the same low-order polynomial characteristic, the difference is only a constant multiplier, yielding a performance gain of approximately 30%.

Language-specific optimizations The results for the final trial, which was designed to determine the performance impact of a language-specific optimization which involves using imperative model manipulation rules instead of purely declarative graph transformation rules (Sec. 5.1.3), are shown in Fig. 16. Similarly to the other case, we measured a constant-multiplier difference of about the same magnitude (30%).

5.3.3 Optimization summary

The summary of the results obtained from various optimization strategies is shown in Table 1.

Variant	Iterations	Model elements	Total time [s]
INC	1200	∼1.5M	4969
Hybrid	1400	∼2.0M	11907

Table 2 Statistics for the maximum possible iteration count

To see how far VIATRA2 can go with the most optimized implementation on our test hardware, we conducted a final test run which ran until the 10GB JVM heap space was exhausted. The results are shown in Table 2.

5.4 Suggestions for improvement

5.4.1 VIATRA2*-related issues*

During the analysis and profiling of our various implementations we have discovered that the performance bottleneck in our system is mainly related to how we manage our models. In almost all cases we have observed that *core model management functions* (e.g. `deleteEntity`, `getAllElementsOfType`, etc.) are consuming most of the time. Most of our optimization techniques described in Sec. 5.1.2 and Sec. 5.1.3 are aiming to decrease the use of (inefficient) model management functions either (i) by reducing their usage frequency through reducing model size, or (ii) by replacing them with less intrusive manipulation operations. Based on this consideration we believe that future optimization work should focus on the following aspects:

– The core model management component should be streamlined to support much faster operations (at least queries) and a more compact representation. Faster query operations would also significantly boost the speed of local search based pattern matching. A straightforward approach is to use

Optimization strategy	Performance	Memory footprint
LS	High-order polynomial	Constant
Switch to INC	Polynomial order reduction	Linear increase with model size
Switch to Hybrid	Linear 50% loss	50% reduction
Dynamic typing, relation retargeting	Linear 30% gain	none
Imperative rules	Linear 30% gain	none

Table 1 Optimization strategies

the incremental pattern matching technology at the core level for the administration of type-instance relationships.
- The generation of model manipulation operations from the RHS of a GT rule has significant impact on overall performance. We plan to investigate ways that allow the interpreter to semi-automatically map declarative RHS specifications into model manipulation operations incorporating the efficient techniques discussed in Sec. 5.1.2.
- Furthermore, type constraints associated with a declaratively specified GT rule should be enforced through static type analysis instead of costly rule application-time checking.

5.4.2 Case study-related issues

By analyzing the data, we have observed several factors, which, in our opinion, may negatively impact the usefulness of the AntWorld example as a basis of performance comparison.

Non-determinism. A random generator is used at critical phases of the transformation, which makes it difficult to validate the implementation. Moreover, randomness may severely impact the overall performance since (i) it is a dominant factor in ant behaviour (determines the length of food searching phases) and (ii) by using a well-crafted fake randomizer, one may force the ants not to find food, thus falsifying the results easily (a few number of ants in a large number of rounds on a small field is a lot cheaper than many ants in a small number of rounds on a large field). Also, the high degree of impact by non-determinism necessitates the recording of a large number of data which, in our case, slowed down the measurement process considerably. Finally, non-determinism prevented the establishment of sample test cases (pairs of inputs and outputs) which would have been useful for solution authors to verify the soundness of implementation and validate the correct interpretation of the specification.

Ambiguous specification. The placement of food is under-specified, and may impact the overall performance. According to the specification, a food bundle should be created on every tenth field, but there is no unambiguous definition of the order in which fields of a new grid circle are created when the grid is expanded. We believe that the intention of the benchmark authors was to have the new nodes created in the natural circular order (either clockwise or anti-clockwise), and our implementation conforms to this assumption, placing food packets evenly along the circular paths. An alternate interpretation of the specification would permit the creation of new fields in an arbitrary order (which is in harmony with conventional graph transformation practice), which would possibly result in uneven distribution of food; finding areas with very dense food would result in higher ant birth rates, while missing these concentrated areas would constrain the growth of the population. As Sec. 5.3 shows, the number of ants has a principal effect on performance.

Comparison difficulties. There is no specified way of obtaining measurement results (no guidelines as to measurement metrics, output formats, etc.), which makes it difficult to compare the performance of the various tools (especially in light of our observations regarding the correlation of execution time vs. number of rounds and number of ants in Sec. 5.3).

In order to overcome these weaknesses we believe that two simple modifications in the specification could erase the random behaviour of the case study:

Random generator specification. What kind of random generator is used has the largest impact on the overall performance, thus a *specific pseudo random generator* like the linear congruential or Lagged Fibonacci generator [28] would ease reproducibility of measurement results, especially with a concrete definition how to use the randomly received values for the selection of possible actions in all situations (e.g., ant movement).

Ant processing order. In order to obtain deterministically reproducible results, the order in which the ants are processed during iterations has to be specified precisely. This feature could easily be added using a single integer attribute for each ant representing its place in the processing sequence. Combining these two modifications would grant deterministic ant behavior on the

same map leading back to the question of food distribution that only needs some extra clarifications –as already mentioned– to obtain a similar map for each rundown.

Overall, based on our considerations about the effectiveness of the case study as a basis of evaluation, we decided to avoid investigating any detailed tool-to-tool comparison. For instance, in our benchmarking paper [13], we have included some comparisons with GrGEN.NET, but with previous acknowledgement of the GrGEN.NET team to avoid misunderstandings and false claims due to the facts that: (i) we are not experts of the transformation language of other tools, thus it is difficult to make judgements about non-VIATRA2 code; (ii) most transformation tools have radically different technological approaches for model persistence, which makes it difficult to do a "fair" comparison (since, for instance, "compiled" transformation engines are typically directed towards different use cases than "interpreted" tools).

6 Conclusion

In this paper, we focused on a detailed performance evaluation of the VIATRA2 model transformation engine with the AntWorld case study. We found the case study very useful to compare various incarnations of VIATRA2 to each other. In addition to highlighting the high-level differences between the local search-based and incremental pattern matcher implementations, we also demonstrated that their combination can form an effective hybrid approach capable of exploiting their advantages without sacrificing additional resources. Additionally, the transformation proved to be powerful enough to also demonstrate language-specific and model management-related fine-tuning possibilities.

However, it is important to mention that as our LS engine does not yet support model sensitive search plan optimization [25,29], the actual assessment of the complexity class does not necessarily hold for other advanced LS-based approaches (like GrGEN.Net).

As a main direction for future work, we plan to integrate the AntWorld example as a basis of functional and non-functional test case set into the standard VIATRA2 testing environment. Additionally, we intend to investigate further optimization possibilities related to multi-threaded pattern matching and parallel transformation execution. We also feel that a more in-depth analysis of how model persistence and low-level queries affect the performance of LS and INC pattern matchers is also needed, to provide effective model management queries supporting both current LS and INC and future parallel pattern matching strategies. Finally, additional assesment of different transformation scenarios is required to come up with efficient general transformation design patterns.

References

1. Bergmann, G., Horváth, A., Ráth, I., Varró, D.: Efficient model transformations by combining pattern matching strategies. In: Proc. of ICMT '09 , 2nd Intl. Conference on Model Transformation, Springer (2009) Submitted.
2. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools. World Scientific (1999)
3. The AGTIVE Tool Contest: official website (2007) `http://www.informatik.uni-marburg.de/~swt/agtive-contest`.
4. GraBaTs - Graph-Based Tools: The Contest: official website (2008) `http://www.fots.ua.ac.be/events/grabats2008/`.
5. Geiss, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.M.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: Graph Transformations - ICGT 2006. Lecture Notes in Computer Science, Springer (2006) 383 – 397 Natal, Brasil.
6. Nickel, U., Niere, J., Zündorf, A.: Tool demonstration: The FUJABA environment. In: The 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, ACM Press (2000)
7. VIATRA - VIsual Automated model TRAnsformations: The Viatra2 Homepage `http://www.eclipse.org/gmt/VIATRA2/`.
8. Varró, G., Varró, D., Schürr, A.: Incremental Graph Pattern Matching: Data Structures and Initial Experiments. In Karsai, G., Taentzer, G., eds.: Graph and Model Transformation (GraMoT 2006). Volume 4 of Electronic Communications of the EASST., EASST (2006)
9. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA transformation system. In: GRaMoT'08, 3rd International Workshop on Graph and Model Transformation, 30th International Conference on Software Engineering (2008)
10. Matzner, A., Minas, M., Schulte, A.: Efficient graph matching with application to cognitive automation. In Schürr, A., Nagl, M., Zündorf, A., eds.: Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007), Springer Verlag (2007)
11. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 321–335
12. Mészáros, T., Madari, I., Mezei, G.: VMTS AntWorld submission. GraBaTs - 4th International Workshop on Graph-Based Tools: The Contest (September 2008)
13. Bergmann, G., Horváth, A., Ráth, I., Varró, D.: A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In: ICGT2008, The 4th International Conference on Graph Transformation. (2008)
14. Kovács, M., Lollini, P., Majzik, I., Bondavalli, A.: An Integrated Framework for the Dependability Evaluation of Distributed Mobile Applications. In: Proc. Int. Workshop on Software Engineering for Resilient Systems (SERENE 2008), Newcastle upon Tyne, UK, November 17-19. (2008) 29–38
15. Albert Zündorf: Antworld benchmark specification, grabats 2008 (2008) `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2008pe%rformancecase.pdf`.

16. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**(3) (2007) 214–234
17. Varró, D., Pataricza, A.: VPM: A visual, precise and multi-level metamodeling framework for describing mathematical domains and UML. Journal of Software and Systems Modeling **2**(3) (October 2003) 187–210
18. Börger, E., Stärk, R.: Abstract State Machines. A method for High-Level System Design and Analysis. Springer-Verlag (2003)
19. Rensink, A.: Representing first-order logic using graphs. In Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G., eds.: Proc. 2nd International Conference on Graph Transformation (ICGT 2004), Rome, Italy. Volume 3256 of LNCS., Springer (2004) 319–335
20. Dorigo, M., Maniezzo, V., Colorni, A.: The ant system: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics-Part B **26** (1996) 29–41
21. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES Approach: Language and Environment. In: In [2]. World Scientific (1999) 487–550
22. ATLAS Group: The ATLAS Transformation Language. Available from `http://www.eclipse.org/gmt`.
23. Zündorf, A.: Graph Pattern Matching in PROGRES. In: Selected papers from the 5th International Workshop on Graph Gramars and Their Application to Computer Science, London, UK, Springer-Verlag (1996) 454–468
24. Ákos Horváth, Gergely Varró and Dániel Varró: Generic search plans for matching advanced graph patterns. In: Proc. of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007), Braga, Portugal, Electornic Communications of the EASST (March 31- Apr. 1 2007) 57–68
25. Varró, G., Varró, D., Friedl, K.: Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In Karsai, G., Taentzer, G., eds.: Proc. of Int. Workshop on Graph and Model Transformation (GraMoT'05). Volume 152 of ENTCS., Tallinn, Estonia, Elsevier (September 2005) 191–205
26. Forgy, C.L.: Rete: A fast algorithm for the many pattern/-many object pattern match problem. Artificial Intelligence **19**(1) (September 1982) 17–37
27. Donald Knuth: The Art of Computer Programming. Third edn. Volume 1. Addison-Wesley (1997) Section 1.2.11: Asymptotic Representations.
28. Luby, M.G.: Pseudorandomness and Cryptographic Applications. Princeton University Press, Princeton, NJ, USA (1994)
29. Batz, G.V., Kroll, M., Geiss, R.: A first experimental evaluation of search plan driven graph pattern matching. In Schürr, A., Nagl, M., Zündorf, A., eds.: Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07). Volume 5088 of LNCS., Springer (2008)

A Complete VTCL source of the Antworld case study

The following source code listing corresponds to our hybrid solution. The pure incremental or local search based solutions differ only in annotations, namely they only have a single pattern matcher selecting annotation, situated at the machine declaration.

We included it in the reviewers' version of the paper for the sake of completeness (and easy access), it will be relocated to the Eclipse site of VIATRA2.

The ASM machine expects 3 input parameters: the number of rounds to execute, a name identifying this experiment (will be used in the output tab name), and whether to enable memory usage measurement (0 for no, 1 for yes; the latter impacts runtime performance heavily).

```
import ants.metamodel;

@incremental
machine antMachine_sleek_hybrid{

// cache
asmfunction model/0;
asmfunction antHill/0;

// statistics
asmfunction pheromones/0;
asmfunction foodCounter/0;
asmfunction foodTotal/0;
asmfunction circlesTotal/0;
asmfunction antsTotal/0;
asmfunction roundCounter/0;

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
// GRID GROWING
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// matches fields marked as boundary fields
pattern boundary(BoundaryField, BoundaryEdge, Hill)={
 antHill(Hill);
 field(BoundaryField);
 antHill.boundary(BoundaryEdge, Hill, BoundaryField);
}

// true if a searcher ant has reached the boundary
pattern boundaryBreachedBySearcher() = {
 field(Field);
  searcherAnt.location(HasAnt, Ant, Field);
 searcherAnt(Ant);
 find boundary(Field, BoundaryEdge, Hill);
}
// wraps the returnPath relation type
@localsearch
pattern alongReturnPath(OuterNeighbor,InnerNeighbor)=
{
 field(InnerNeighbor);
 field(OuterNeighbor);
 field.returnPath(RP, OuterNeighbor, InnerNeighbor);
}
// wraps the circlePath relation type
@localsearch
pattern circled(Field1, Field2) =
{
 field(Field1);
 field(Field2);
 field.circlePath(CP, Field1, Field2);
}
// identifies the next boundary field in
//the boundary circle provided that it is still
//at the boundary (has not been expanded)
@localsearch
pattern nextBoundaryField(BoundaryField,
 NextBoundaryField, NextBoundaryEdge) = {
  find circled(BoundaryField, NextBoundaryField);
  find boundary(NextBoundaryField,
                NextBoundaryEdge, Hill);
 }
// wraps the cornerField entity type
```

```
@localsearch
pattern corner(CornerField) = {
 cornerField(CornerField);
}

// expands the grid at the given boundary field
// BoundaryField determines the number of new fields
// (3 for corners, 1 otherwise)
// the first of them is returned in Back,
// the last one in front
rule expandBoundary(in BoundaryField, out Back,
         out Front, in OldBoundaryEdge, in Hill) =
 let BRP = undef, Model = model() in seq {
  new(field(Back) in Model);
  new(field.returnPath(BRP, Back, BoundaryField));
  setTo(OldBoundaryEdge, Back);
  call newField(Back);
  if (find corner(BoundaryField))
   let BE1=undef, BE2=undef, CRP=undef, FRP=undef,
   CP1=undef, CP2=undef, ExpandedCorner = undef in
   seq {
    new(cornerField(ExpandedCorner) in Model);
    new (antHill.boundary(BE1,Hill, ExpandedCorner));
    new(field.returnPath(CRP, ExpandedCorner,
                         BoundaryField));
    new(field.circlePath(CP1,Back, ExpandedCorner));
    call newField(ExpandedCorner);

    new(field(Front) in Model);
    new(antHill.boundary(BE2, Hill, Front));
    new(field.returnPath(FRP, Front, BoundaryField));
    new(field.circlePath(CP2, ExpandedCorner,Front));
    call newField(Front);
   }
  else update Front = Back;
 }
// creates food on every tenth field
rule newField(in Field) = seq {
 if (foodCounter() < 9)
  update foodCounter() = foodCounter() + 1;
 else let Food = undef, HF=undef in seq {
  update foodCounter() = 0;
  update foodTotal() = foodTotal() + 1;
  new (food(Food) in Field);
  new (field.hasFood(HF, Field, Food));
  setValue(Food, 100);
 }
}

// expands the grid with a new circle
rule growGrid() = let Hill=antHill(), CP=undef,
 FirstExpanded = undef, PreviousExpanded = undef,
 PreviousBoundaryField = undef in
 choose FirstBoundaryField, FirstBoundaryEdge with
  find boundary(FirstBoundaryField,
                FirstBoundaryEdge, Hill) do
  seq {
   update PreviousBoundaryField = FirstBoundaryField;
   call expandBoundary(FirstBoundaryField,
                    FirstExpanded, PreviousExpanded,
                    FirstBoundaryEdge, Hill);
   iterate choose NextBoundaryField, NextBoundaryEdge
    with find nextBoundaryField(PreviousBoundaryField
     , NextBoundaryField, NextBoundaryEdge) do
    let BackExpanded = undef,
        FrontExpanded = undef in seq {
     update PreviousBoundaryField
              = NextBoundaryField;
     call expandBoundary(NextBoundaryField,
                         BackExpanded,FrontExpanded,
                         NextBoundaryEdge, Hill);
     new(field.circlePath(CP,
                     PreviousExpanded, BackExpanded));
     update PreviousExpanded = FrontExpanded;
    }
   new(field.circlePath(CP,
                     PreviousExpanded, FirstExpanded));
   update circlesTotal() = circlesTotal() + 1;
  }

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
// ANT ACTIONS

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// wraps the carrierAnt entity type
pattern carrier(Ant) = {
 carrierAnt(Ant);
}
// wraps the searcherAnt entity type
pattern searcher(Ant) = {
 searcherAnt(Ant);
}
// wraps the searcherAnt.location relation type
pattern hasSearcherAnt(LocationEdge, Field, Ant) = {
 searcherAnt(Ant);
 searcherAnt.location(LocationEdge, Ant, Field);
 field(Field);
}
// wraps the carrierAnt.location relation type
pattern hasCarrierAnt(LocationEdge, Field, Ant) = {
 carrierAnt(Ant);
 carrierAnt.location(LocationEdge, Ant, Field);
 field(Field);
}
// wraps the hasFood relation type
pattern foodAvailable(Field, Food) = {
 field(Field);
 field.hasFood(HF, Field, Food);
 food(Food);
}

// identifies a searcher ant that stands on food
pattern canGrab(Ant, LocationEdge, Food, Field) = {
 find hasSearcherAnt(LocationEdge, Field, Ant);
 find foodAvailable(Field, Food);
}

// takes the unit of food and
// transforms the ant into a carrierAnt
rule grab(in Ant, in LocationEdge, in Food) =
 let Rest = toInteger(value(Food)) -1  in seq{
 if (Rest > 0) setValue(Food, Rest);
  else delete(Food);
 delete(instanceOf(Ant,ants.metamodel.searcherAnt));
 delete(instanceOf(LocationEdge,
        ants.metamodel.searcherAnt.location));
 new(instanceOf(Ant,ants.metamodel.carrierAnt));
 new(instanceOf(LocationEdge,
     ants.metamodel.carrierAnt.location));
 }
// deposits the food at the anthill
//  and resumes the searching
rule deposit(in Hill,
             inout Ant, in LocationEdge) = seq{
 setValue(Hill, toString( toInteger(value(Hill))+1));
 delete(instanceOf(Ant,ants.metamodel.carrierAnt));
 delete(instanceOf(LocationEdge,
        ants.metamodel.carrierAnt.location));
 new(instanceOf(Ant,ants.metamodel.searcherAnt));
 new(instanceOf(LocationEdge,
     ants.metamodel.searcherAnt.location));
}
// moves an ant by retargeting its location relation
rule moveAnt(in OldLocation, in NewField) =
 setTo(OldLocation, NewField);

// wraps the hasPheromone relation type
pattern hasPheromone(Field, Pheromone) = {
 field(Field);
 field.hasPheromone(HF, Field, Pheromone);
 pheromone(Pheromone);
}
// leaves a pheromone trace at the field
rule leavePheromone(in Field) =
 try choose Pheromone with
  find hasPheromone(Field, Pheromone) do
   setValue(Pheromone,
            1024 + toInteger(value(Pheromone)));
 else let Pheromone = undef, HF = undef in seq {
  new (pheromone(Pheromone) in Field);
  new (field.hasPheromone(HF, Field, Pheromone));
  setValue (Pheromone, 1024);
  update pheromones() = pheromones()+1;
 }
```

```
// identifies fields with an attracting
// amount of pheromones
pattern attractingField(Field) = {
 find hasPheromone(Field, Pheromone);
 check(toInteger(value(Pheromone)) > 9);
}

// a field away from the hill that attracts
// with pheromones, randomly selected if there
// are multiple (from a corner)
@Random
@localsearch
pattern attractingOuterNeighbor(Field1, Field2) = {
  find alongReturnPath(Field2, Field1);
  find attractingField(Field2); // true hybrid!
 }

// wraps the antHill entity type
pattern home(Field) = {
 antHill(Field);
}

// a random neighboring field except the anthill
// traverses path relations in both directions
@Random
@localsearch
pattern anyNeighborButHome(Field1, Field2) = {
  field(Field1);
  field(Field2);
  field.path(P, Field1, Field2);
  neg find home(Field2); // true hybrid!
 } or {
  field(Field1);
  field(Field2);
  field.path(P, Field2, Field1); // reverse direction
  neg find home(Field2); // true hybrid!
 }

// performs a food searching step
// moves towards attracting pheromone, if any
// moves to random neigbor field otherwise
rule search(in Ant) =
 choose Field1, HA1 with
  find hasSearcherAnt(HA1, Field1, Ant) do
   try choose /*random*/ Field2 with
    find attractingOuterNeighbor(Field1, Field2) do
     call moveAnt(HA1, Field2);
   else choose /*random*/ Field2 with
    find anyNeighborButHome(Field1, Field2) do
     call moveAnt(HA1, Field2);

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
// WORLD MANAGEMENT
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// wraps the pheromone entity type
pattern pheromone(P) = {
 pheromone(P);
}
// evaporates 5% of the pheromone patch
// deletes the patch if it becomes empty
rule evaporate(in Pheromone) =
 let Rest = (19*toInteger(value(Pheromone)))/20 in
 if (Rest > 0) setValue(Pheromone, Rest); else seq {
  delete(Pheromone);
  update pheromones() = pheromones() - 1;
 }

// consumes a deposited food item to create a new ant
rule consume(in Hill) =
 let Ant = undef, HA = undef in seq{
  setValue(Hill,toString(toInteger(value(Hill))-1));
  new(searcherAnt(Ant) in Hill);
 new(searcherAnt.location(HA, Ant, Hill));
 update antsTotal() = antsTotal() + 1;
 }

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
// MAIN
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

// executes a round
rule doRound() = let Hill = antHill() in seq {
  //Ant actions
iterate choose Ant, LocationEdge, Food, Field with
  find canGrab(Ant, LocationEdge, Food, Field) do
   call grab(Ant,LocationEdge,Food); //grab food
forall Ant, LocationEdge with //desposit on the Hill
  find hasCarrierAnt(LocationEdge, Hill, Ant) do
   call deposit(Hill,Ant,LocationEdge);
forall Ant, FromField, HA1 with // return
 find hasCarrierAnt(HA1, FromField, Ant) do // return
  choose NewField with
   find alongReturnPath(FromField, NewField) do seq {
    call moveAnt(HA1, NewField);
    call leavePheromone(FromField);
   }
forall Ant with find searcher(Ant)
 do call search(Ant); // both kinds of search
// only searchers can breach the boundary!

//World management
forall Pheromone with find pheromone(Pheromone)
 do call evaporate(Pheromone);
iterate //create new ants
 if(toInteger(value(Hill)) > 0) call consume(Hill);
 else fail; // until all deposited food is consumed
if (find boundaryBreachedBySearcher())
 call growGrid(); //grow the game map
}

// what its name says
rule printStatistics(in Buf, in MemTelemetry,
 in RoundCounter, in Rounds, in BlockSize,
 in AntAccumulator, in StartTime,
 inout LastTime) = let CurrentTime = systime() in seq
 {
 println(Buf,"\t<round-block finished=\""
  + RoundCounter + "\" goal=\"" + Rounds
  + "\" block-size=\"" + BlockSize + "\">");
 println(Buf, "\t\t<elapsed-time>");
 println(Buf, "\t\t\t<per-block> "
  + (CurrentTime-LastTime) + " </per-block>");
 println(Buf, "\t\t\t<per-round> "
  + (CurrentTime-LastTime)/BlockSize
  + " </per-round>");
 println(Buf, "\t\t\t<per-round-per-1000ants> "
  + 1000*(CurrentTime-LastTime) / AntAccumulator
  + " </per-round-per-1000ants>");
 println(Buf, "\t\t\t<total> "
  + (CurrentTime-StartTime) + " </total>");
 println(Buf, "\t\t</elapsed-time>");
 println(Buf, "\t\t<circles> "
  + circlesTotal() + " </circles>");
 println(Buf, "\t\t<grid-fields> "
  + circlesTotal() * circlesTotal() * 4
  + " </grid-fields><!-- excluding the anthill -->");
 println(Buf, "\t\t<food-bundles-created> "
  + foodTotal() + " </food-bundles-created>");
 println(Buf, "\t\t<pheromone-traces> "
  + pheromones() + " </pheromone-traces>");
 println(Buf, "\t\t<ants> "
  + antsTotal() + " </ants>");
 if (MemTelemetry == 1) println(Buf, "\t\t<memory> "
  + measureMemoryFootprint(6) + " </memory>");
 else println(Buf, "\t\t<memory> NA </memory>");
 println(Buf, "\t</round-block>");
 update LastTime = CurrentTime;
}

// entry point
rule main(in Rounds, in Variant, in MemTelemetry) =
 let StartTime = systime(), BlockSize = 25,
 Buf = getBuffer("core://"+Variant)
in seq {
// initialize asmfunctions
 update model() = ref("ants.model");
 update antHill() = ref("ants.model.hill");

 update foodCounter()  =
  toInteger(value(
            ref("ants.statistics.foodCounter")));
 update foodTotal()    =
```

```
  toInteger(value(
            ref("ants.statistics.foodTotal")));
 update circlesTotal() =
  toInteger(value(
            ref("ants.statistics.circlesTotal")));
 update antsTotal()    =
  toInteger(value(
            ref("ants.statistics.antsTotal")));
 update pheromones()   =
  toInteger(value(
            ref("ants.statistics.pheromones")));
 update roundCounter() =
  toInteger(value(
            ref("ants.statistics.roundCounter")));

 println(Buf, "<anthill-simulation rounds=\""
  + Rounds + "\" up-to=\""
  + (Rounds + roundCounter()) + "\">");

 // execute rounds and print satistics after
 // blocks of 25
 let BlockCounter = 0, AntAccumulator = 0,
  RoundMax = Rounds + roundCounter(),
  LastTime=StartTime in
 iterate seq {
  if (roundCounter() >= RoundMax) fail;
  update roundCounter() = roundCounter() + 1;

  call doRound(); //one round execution

  update BlockCounter = BlockCounter + 1;
  update AntAccumulator = AntAccumulator+antsTotal();
  if (BlockCounter >= BlockSize) seq {
   call printStatistics(Buf, MemTelemetry,
    roundCounter(), RoundMax, BlockSize,
    AntAccumulator, StartTime, LastTime);
   update BlockCounter = 0;
   update AntAccumulator = 0;
  }
 }

// conclude output
 println(Buf, "\t<final-statistics>");
 println(Buf, "\t\t<total-elapsed-time> "
  + (systime()-StartTime)
  + " </total-elapsed-time>");
 println(Buf, "\t\t<circles> "
  + circlesTotal() + " </circles>");
 println(Buf, "\t\t<grid-fields> "
  + circlesTotal() * circlesTotal() * 4
  + " </grid-fields><!-- excluding the anthill -->");
 println(Buf, "\t\t<food-bundles-created> "
  + foodTotal() + " </food-bundles-created>");
 println(Buf, "\t\t<pheromone-traces> "
  + pheromones() + " </pheromone-traces>");
 println(Buf, "\t\t<ants> "
  + antsTotal() + " </ants>");
 println(Buf, "\t</final-statistics>");
 println(Buf, "</anthill-simulation>");

 // save statistics to the model space
 setValue(ref("ants.statistics.foodCounter"),
               foodCounter());
 setValue(ref("ants.statistics.foodTotal"),
               foodTotal());
 setValue(ref("ants.statistics.circlesTotal"),
               circlesTotal());
 setValue(ref("ants.statistics.antsTotal"),
               antsTotal());
 setValue(ref("ants.statistics.pheromones"),
               pheromones());
 setValue(ref("ants.statistics.roundCounter"),
               roundCounter());
}
}
```

Listing 8 Complete source code of the VIATRA2 solution