Incremental pattern matching in the VIATRA model transformation system

Gábor Bergmann
András Ökrös
István Ráth (rath@mit.bme.hu)
Dániel Varró
Gergely Varró

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Overview

• Introduction

• Concepts

• Performance analysis

• Future work

• Summary
Introduction
Incremental model model transformations

- Key usage scenarios for MT:
  - Mapping between languages
  - Intra-domain model manipulation
    - Model execution
    - Validity checking (constraint evaluation)
- They work with **evolving** models.
  - Users are constantly changing/modifying them.
  - Users usually work with **large** models.
- Problem: transformations are slow
  - To execute... (large models)
  - and to re-execute again and again (always starting from scratch).
- Solution: incrementality
  - Take the source model, and its mapped counterpart;
  - Use the information about how the source model was changed;
  - Map and apply the changes (but ONLY the changes) to the target model.
Towards incrementality

• How to achieve incrementality?
  ▫ Incremental updates: avoid re-generation.
    • Don’t recreate what is already there.
    • → Use reference (correspondence) models.
  ▫ Incremental execution: avoid re-computation.
    • Don’t recalculate what was already computed.
    • How?
Incremental graph pattern matching

- Graph transformations require pattern matching
- Goal: retrieve the *matching set* quickly
- How?
  - Store (cache) matchings
  - Update them as the model changes
    - Update precisely (incrementality)
- *Expected results*: good, if...
  - There is enough memory (*)
  - Queries are dominant
  - Model changes are relatively sparse (**)
Operational overview

- XForm interpreter model
- Incremental pattern matcher
- VIATRA Model space
- Pattern matching
- Event notification
- Model manipulation
- Updates
Architecture

VIATRA2 Framework

- XML serializer
- Model parser
- XForm parser
- Native importer & loader interface
- XForm interpreter
- Core interfaces
- Incremental pattern matcher
- LS pattern matcher
- VIATRA Model space
- Program model store
Concepts
Core idea: use RETE nets

- RETE network
  - node: (partial) matches of a (sub)pattern
  - edge: update propagation
- Demonstrating the principle
  - input: Petri net
  - pattern: fireable transition
  - Model change: new transition (t3)
RETE network construction

- **Key: pattern decomposition**
  - Pattern = set of constraints (defined over pattern variables)
  - Types of constraints: type, topology (source/target), hierarchy (containment), attribute value, generics (instanceOf/supertypeOf), *injectivity*, [negative] pattern calls, ...

- **Construction algorithm (roughly)**
  - 1. Decompose the pattern into elementary constraints (*)
  - 2. Process the elementary constraints and connect them with appropriate intermediate nodes (JOIN, MINUS-JOIN, UNION, ...)
  - 3. Create terminator production node
Key RETE components

- JOIN node
  - ~relational algebra: natural join

- MINUS-JOIN
  - Negative existence
Other VIATRA features

• Pattern calls
  ▫ Simply connect the production nodes
  ▫ Pattern recursion is fully supported
• OR-patterns
  ▫ UNION intermediate nodes
• Check conditions
  ▫ check (value(X) % 5 == 3)
  ▫ check (length(name(X)) < 4)
  ▫ check (myFunction(name(X))!="myException")
  ▫ Filter and term evaluator nodes
• Result: full VIATRA transformation language support; any pattern can be matched incrementally.
Updates

• Needed when the model space changes
• VIATRA notification mechanism (EMF is also possible)
  ▫ Transparent: user modification, model imports, results of a transformation, external modification, ... → RETE is always updated!
• Input nodes receive elementary modifications and release an update token
  ▫ Represents a change in the partial matching (+/-)
• Nodes process updates and propagate them if needed
  ▫ PRECISE update mechanism
Performance analysis
Performance

• In theory...
  ▫ Building phase is slow ("warm-up")
    ◆ How slow?
  ▫ Once the network is built, pattern matching is an "instantaneous" operation.
    ◆ Excluding the linear cost of reading the result set.
  ▫ But... there is a performance penalty on model manipulation.
    ◆ How much?

• Dependencies?
  ▫ Pattern size
  ▫ Matching set size
  ▫ Model size
  ▫ ...?
Benchmarking

- **Example transformation: Petri net simulation**
  - One complex pattern for the enabledness condition
  - Two graph transformation rules for firing
  - As-long-as-possible (ALAP) style execution (“fire at will”)
  - Model graphs:
    - A “large” Petri net actually used in a research project (~60 places, ~70 transitions, ~300 arcs)
    - Scaling up: automatic generation preserving liveness (up to 100000 places, 100000 transitions, 500000 arcs)

- **Analysis**
  - Measure execution time (average multiple runs)
  - Take “warm-up” runs into consideration

- **Profiling**
  - Measure overhead, network construction time
  - “Normalize” results
Profiling results

- Model manipulation overhead: ~15% (of overall CPU time)
  - Depends largely on the transformation!
- Memory overhead
  - Petri nets (with RETE networks) up to ~100000 fit into 1-1.5GB RAM (VIATRA model space limitations)
  - Grows linearly with model size (as expected)
  - Nature of growth is pattern-dependent
- Network construction overhead
  - Similar to memory; pattern-dependent.
  - PN: In the same order as VIATRA’s LS heuristics initialization.
Execution times

Sparse Petri net benchmark

Three orders of magnitude and growing...

Matches/outperforms GrGEN.NET for large models and high iteration counts.
Benchmarking summary

• Predictable near-linear growth
  ▫ As long as there is enough memory
  ▫ Certain problem classes: constant execution time 😊
  ▫ A ga
Improving performance

• Strategies
  ▫ **Improve the construction algorithm**
    • Memory efficiency (node sharing)
    • Heuristics-driven constraint enumeration (based on pattern [and model space] content)
  ▫ **Parallelism**
    • Update the RETE network in parallel with the transformation
    • Parallel network construction
Future work
More benchmarking...

- Ongoing research
  - Extending the Varro benchmark
    - Mutex STS/LTS
    - ORM
  - Extended benchmarking use cases
    - Simulation (model execution)
    - Synchronization
    - Constraint evaluation
  - Parallel transformations
Event-driven live transformations

• Problem: MT is mostly batch-like
  ▫ But models are constantly evolving → Frequent re-transformations are needed for
    • mapping
    • synchronization
    • constraint checking
    • ...
  ▪ An incremental PM can solve the performance problem, but a formalism is needed
    ▫ to specify when to (re)act
    ▫ and how.
  ▪ Ideally, the formalism should be MT-like.
Event-driven live transformations (cont’d)

- An idea: represent events as model elements.
- Our take: represent events as changes in the matching set of a pattern.
  - ~generalization
- Live transformations
  - maintain the context (variable values, global variables, ...);
  - run as a “daemon”, react whenever necessary;
  - as the models change, the system can react instantly, since everything needed is there in the RETE network: no re-computation is necessary.
- Paper accepted at ICMT2008.
Summary

• Incremental pattern matching support integrated into VIATRA2 R3
  ▫ Based on the RETE algorithm
  ▫ Provides full support for the pattern language
  ▫ High performance in certain problem classes

• Future
  ▫ Performance will be further improved
  ▫ New applications in live transformations