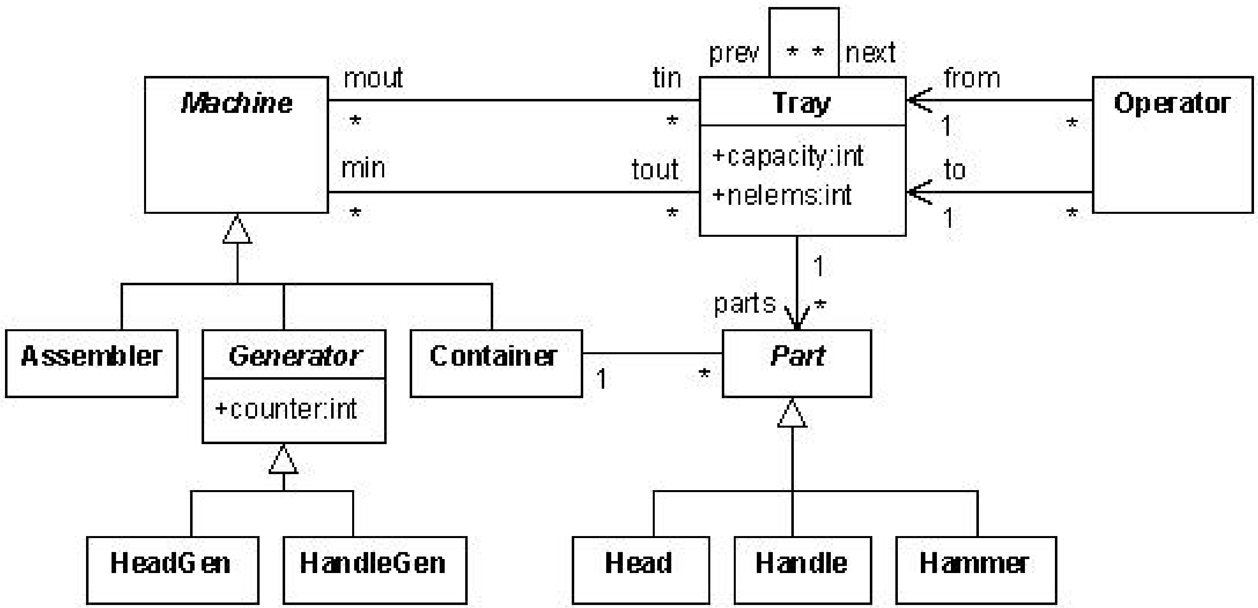
Analyzing Rule-Based Behavioural Semantics of Visual Modeling Languages with Maude

José Eduardo Rivera — Universidad de Málaga
Esther Guerra — Universidad Carlos III de Madrid
Juan de Lara — Universidad Autónoma de Madrid
Antonio Vallecillo — Universidad de Málaga

A motivating example

Meta-model of DSVL for production systems



context Operator
inv: self.from.next->includes(self.to)

context Tray
inv: self.capacity > 0

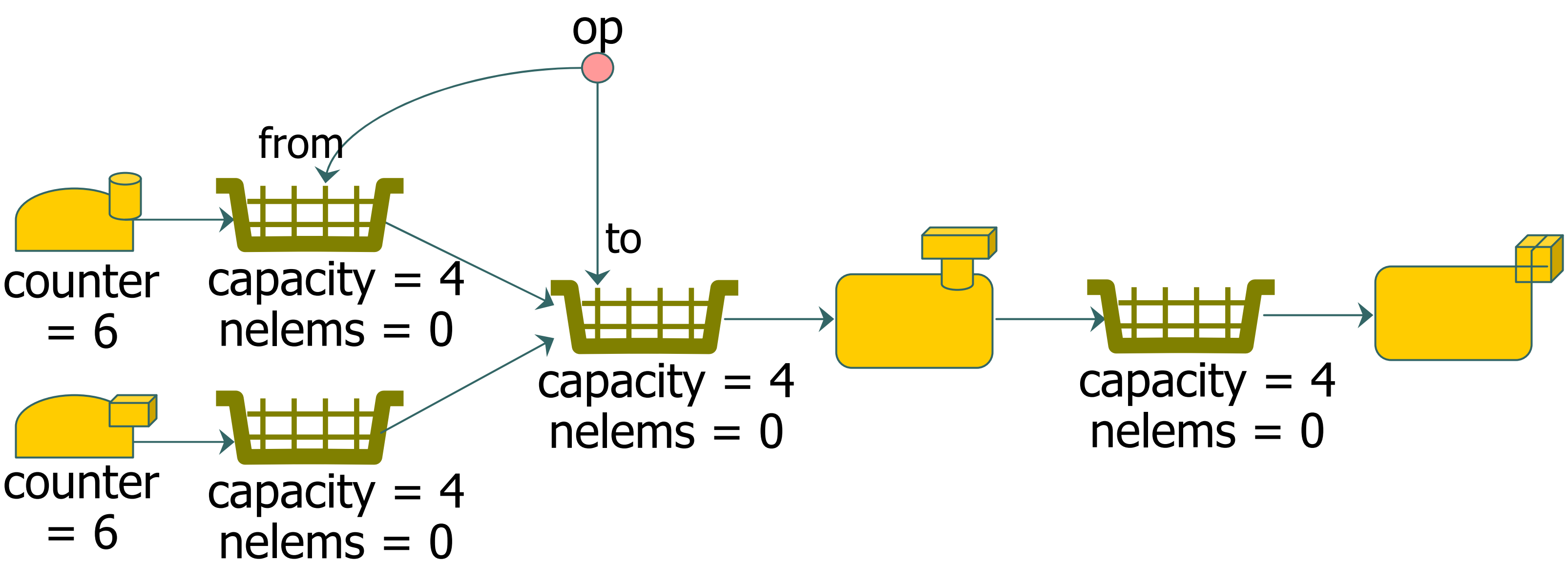
context Tray
inv: self.part->size() = self.nelems

context Tray
inv: self.nelems <= self.capacity

- Different kinds of machines connected through trays
- Trays contain parts and can be interconnected
- Operators transfer parts between connected trays

A model of a production system

Concrete syntax

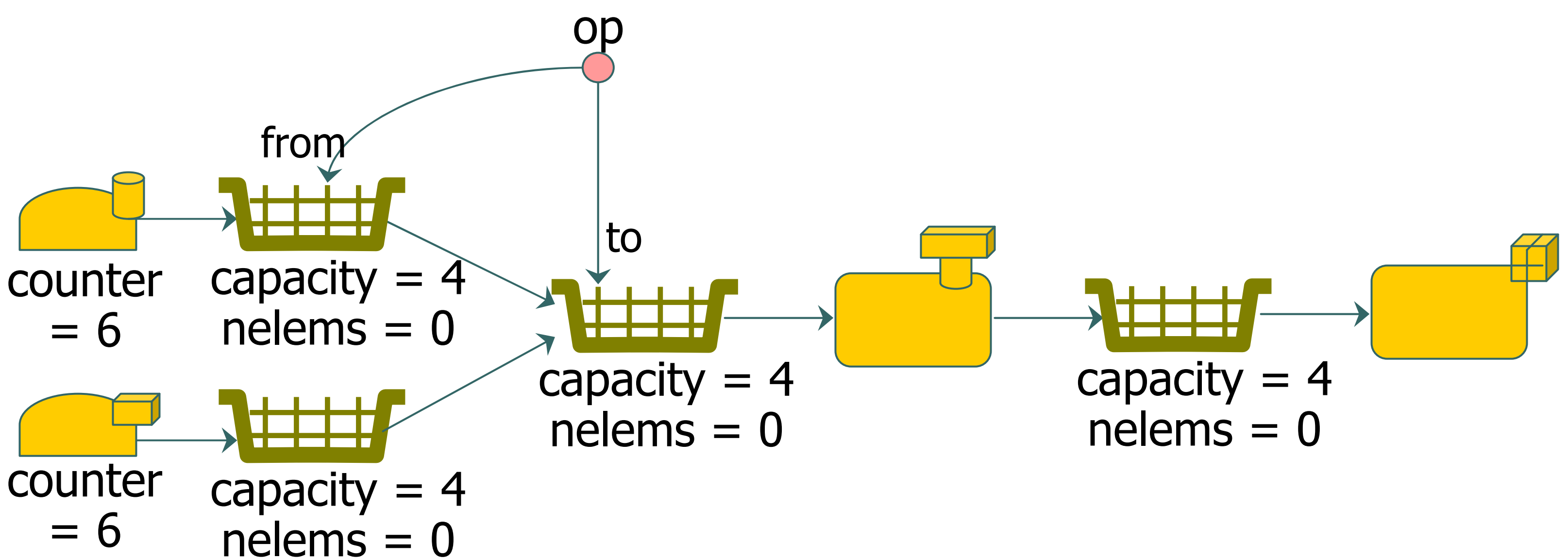


- Nice picture!
- …But how does the system actually works?
- …How can I check that it does work well?

MDE is more than Conceptual Modeling!!!

- Current DSLs
  - Unanimated (mostly static)
  - Limited analysis capabilities
- Almost inexistent Tool Support
  - Simulation
  - Analysis
  - Estimation
  - Quality evaluation and control
  - …
- Almost inexistent proven methodologies
  - For neither development nor modernization

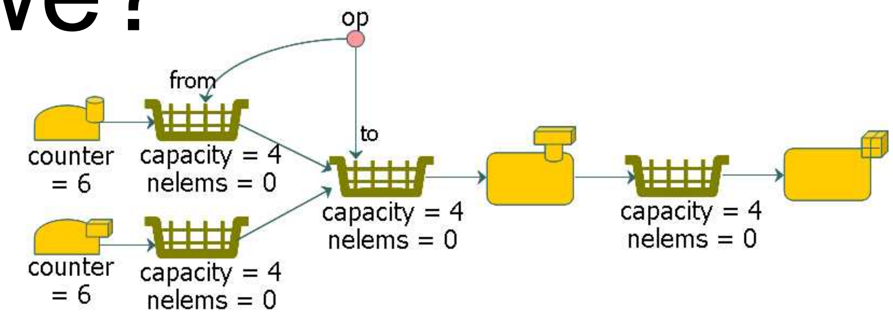
A model of a production system



Key issues:

1. How to specify the behavioral semantics of Visual DSLs in a **precise**, **intuitive**, yet **formal** way
2. How to analyze the behavior of a given system?

Why should we?

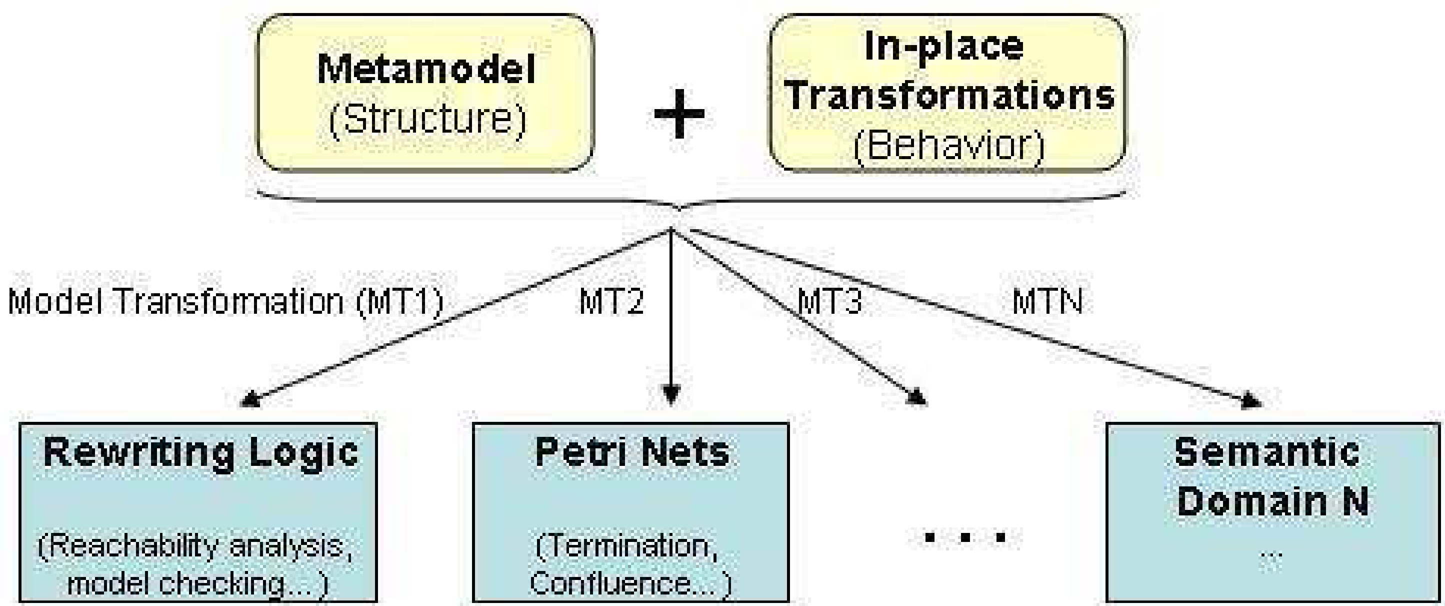


- Animate models
  - Define the behavioral semantics of DSLs
    - so that models can be understood, manipulated and maintained by both users and machines (i.e. **Tools**!)
  - Conduct simulations
- Analyze models
  - Define different semantics to a DSL (depending on our focus: performance, deadlock-freedom, simulation,…)
  - Make effective use of specific Analysis Tools

How do we do that?

- Option#1: Use a single language/notation/…
  - We've tried that for years… ☹
- Option#2: Use different DSLs and define "semantic bridges" between them
  - Each DSL is more apt for expressing some concerns
  - Each DSL has a precise semantics and set of (specific and very efficient) associated tools
  - Bridges provide "semantic mappings" semantic domains (and analysis tools)

How do we do that?



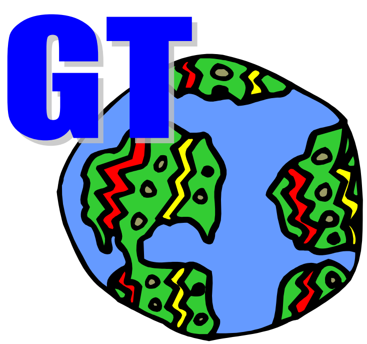
In this paper

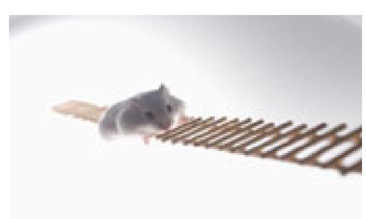
- GT is used to specify behavioral semantics
  - GT semantics are then translated (encoded) into Maude specs
  - Maude specs can be analyzed using the Maude tool-kit

- Benefits
  - Additional analysis techniques to GT specs
  - Intuitive representation of Maude specs

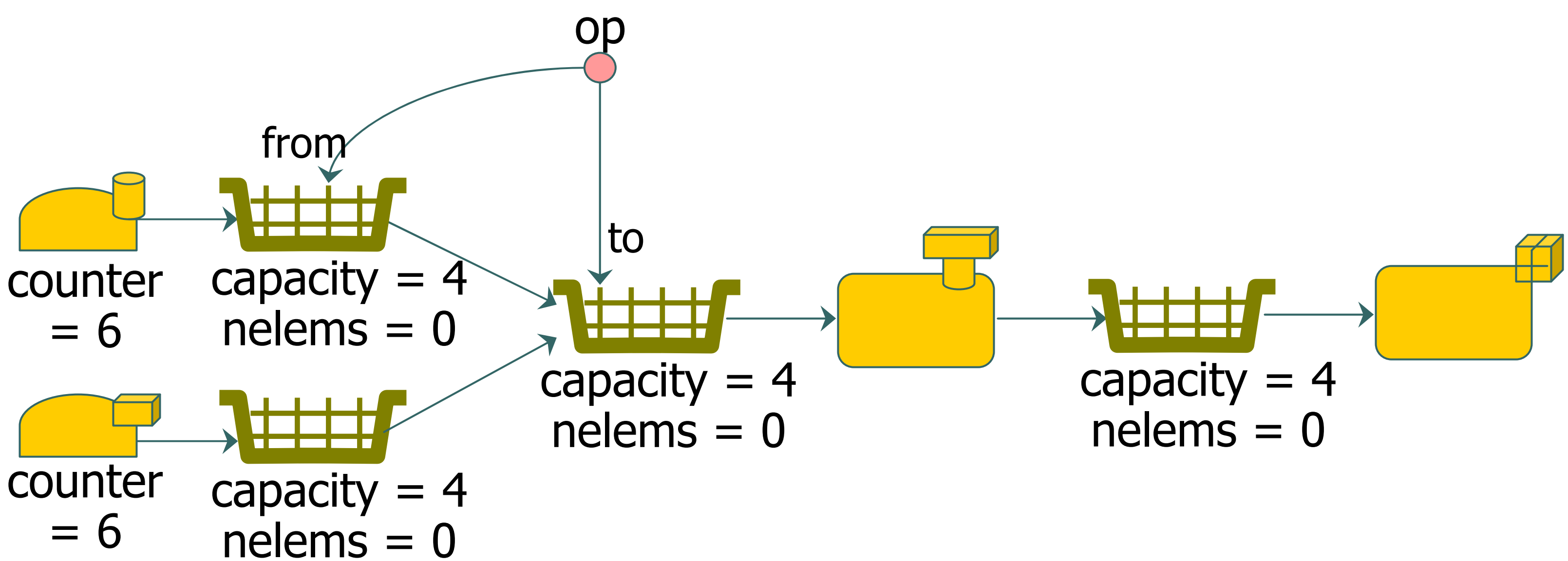
Why?

- Graph transformation
  - Benefits: visual, declarative, rule-based way to specify behavior, very close to the domain expert
  - Drawbacks: limited analysis capabilities in some cases (e.g., if dealing with attributes)
- Maude
  - Benefits: many formal analysis methods and tools
  - Drawbacks: specialized knowledge and expertise





GT for Behavioral Specifications

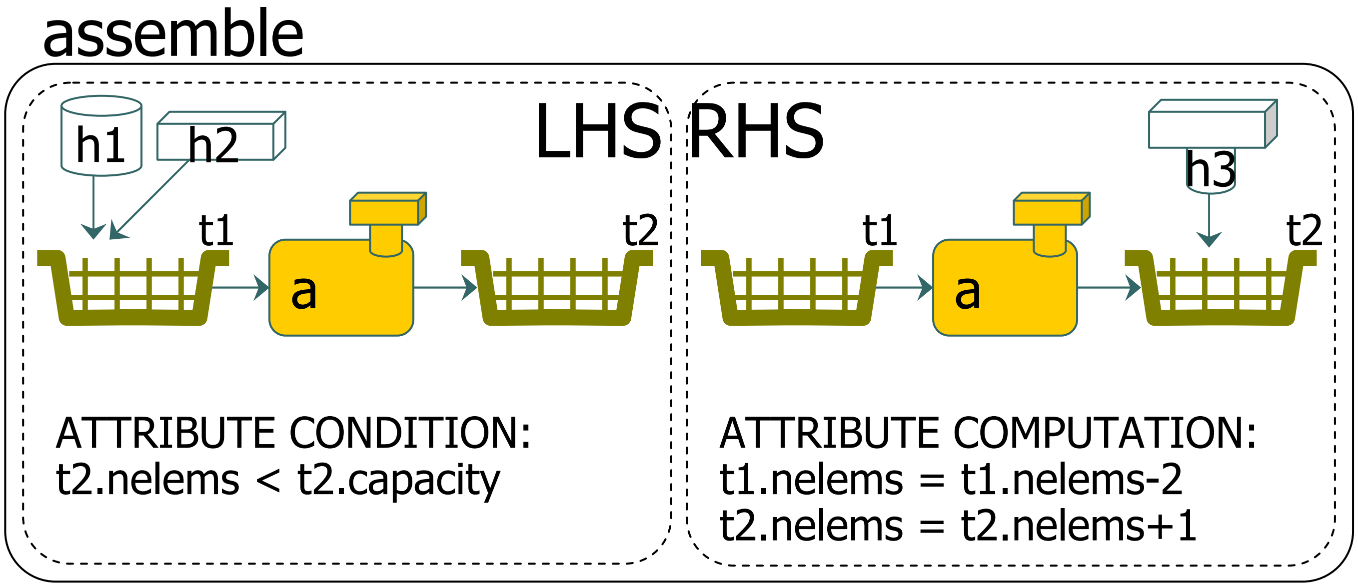


- Graph transformation rules use the concrete syntax to express how a model can evolve through time, i.e. its **behavioral semantics**

Graph transformation

Graph transformation rules

l:[NAC] x $LHS \rightarrow RHS$



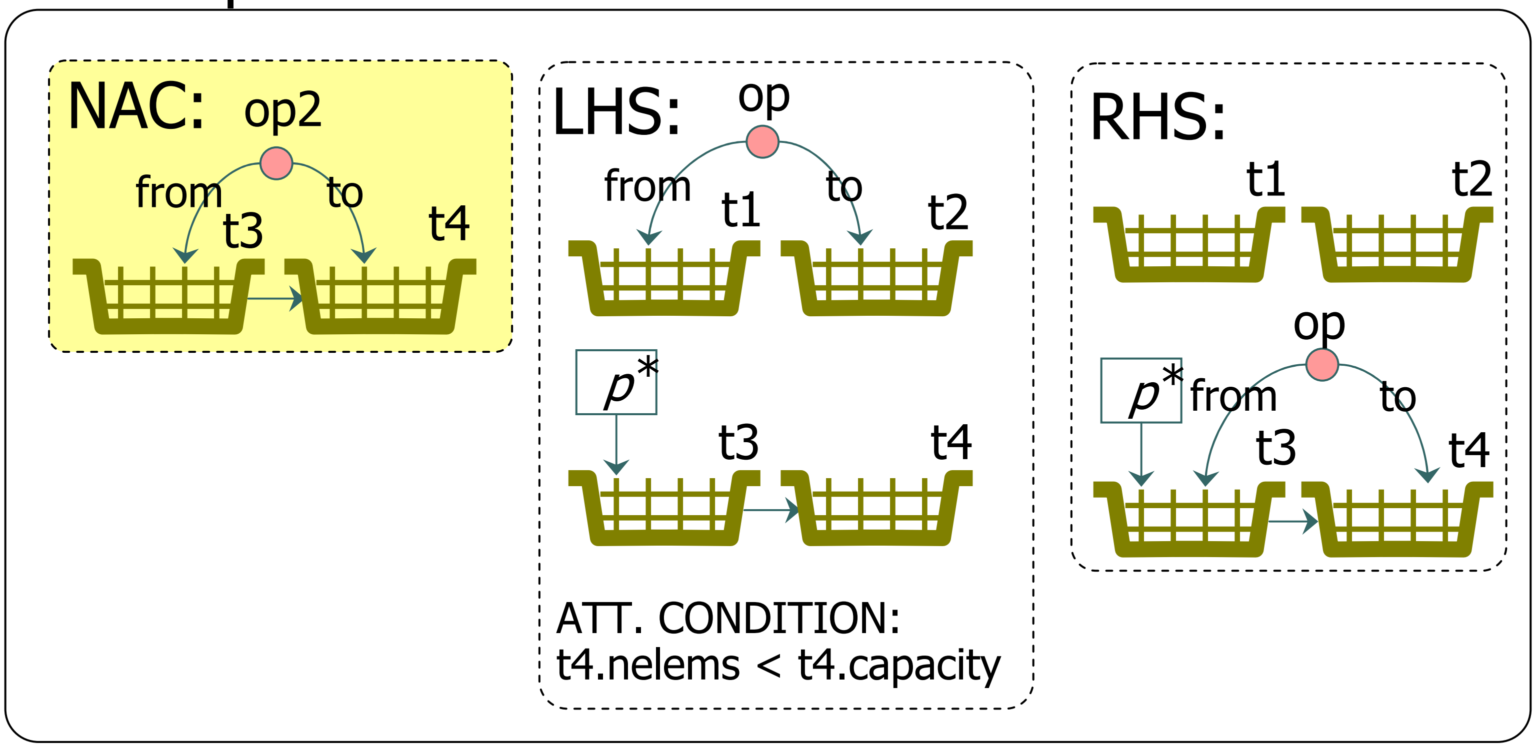
- LHS: pre-conditions (including attribute conditions)
- RHS: post-conditions (including attribute computacions)
- NAC: additional negative application condition

Graph transformation

Graph transformation rules

$l:[NAC]\ \mathrm{x}\ LHS \longrightarrow RHS$

moveOperator



Graph transformation

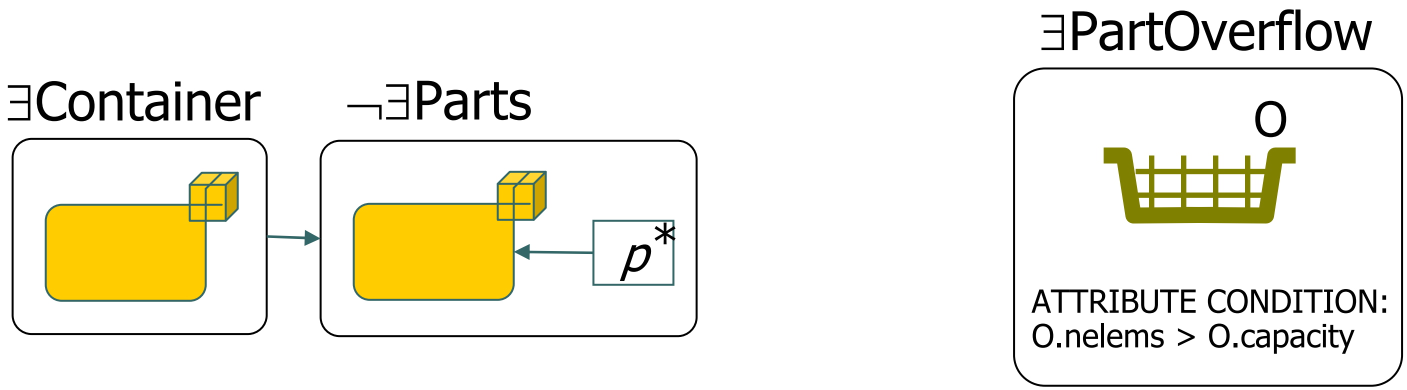
Derivation

While some rule is applicable do:

1. Find a morphism from the LHS to the host graph
    1. NACs and attribute conditions must be satisfied as well
2. Substitute the match by the RHS
    1. Elements in the LHS and not in the RHS are deleted
    2. Elements in the RHS and not in the LHS are created
3. Calculate attribute computations

- There are two main algebraic formalizations of GT: DPO (double pushout) and SPO (single pushout)
- The chosen semantics will affect the Maude equivalent representation

Graph constraints



- A graph constraint is made of a set of graphs related through morphisms
- It demands the existence or absence of a certain graph structure in a model
- We use graph constraints to express **model properties** to be analyzed in an intuitive way

Introduction to Maude

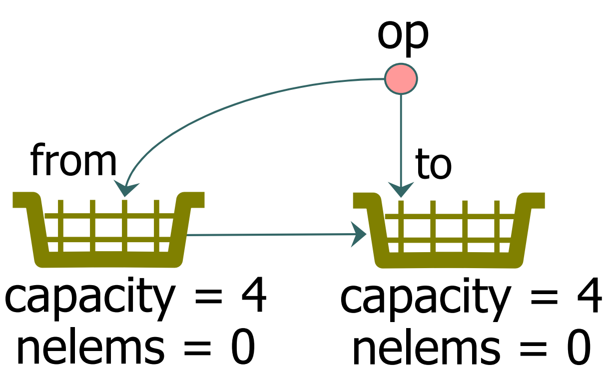
- It support equational logic and rewriting logic specification and programming of systems
- A system is axiomatized by an equational theory describing its states and a collection of rewrite rules
- Rule syntax:

  crl [l] : t => t' if Cond

```
mod BANK is
   class Account | balance : Int .
   class Deposit | account : Oid, amount : Int .
   vars N M : nat . vars A D : Oid .
   crl [deposit] :
      < A : Account | balance : N >
      < D : Deposit | account : A, amount : M >
      => < A : Account | balance : N + M  >
   if (M > 0)
endm
```

From graph transformation to Maude
Encoding models

- Nodes represented by objects
- Attributes and edges represented by object attributes

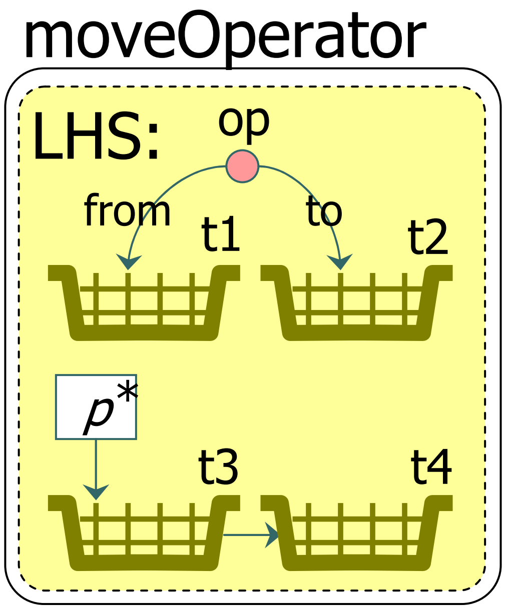



```
ProductionSystem {
  < 't1 : Tray | parts : empty,
                 next : 't3, prev : empty,
                 min : empty, mout : empty,
                 capacity : 4, nelems : 0 >
  < 't3 : Tray | parts : empty,
                 next : empty, prev : 't2,
                 min : empty, mout : empty,
                 capacity : 4, nelems : 0 >
  < 'op : Operator | from : 't1, to : 't3 >
}
```

- Meta-models → a sort for each element (e.g. @Class)

From graph transformation to Maude

Encoding LHS of rules



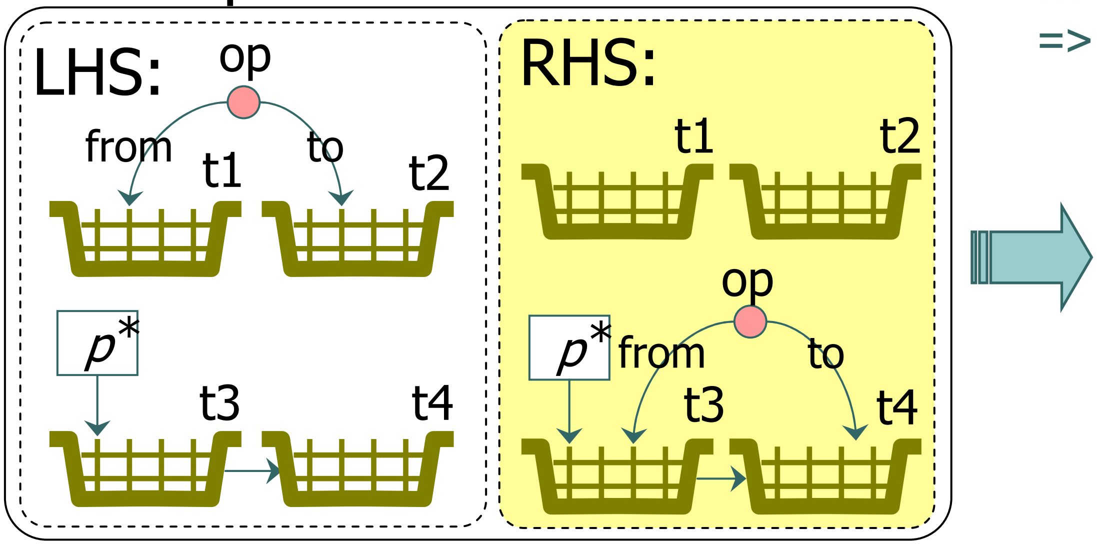

```
crl [MoveOperator] :
   ProductionSystem {
     < T1 : Tray | SFS@T1 >
     < T2 : Tray | SFS@T2 >
     < OP : Operator |
              from : T1,
              to : T2, SFS@OP >
     < T3 : Tray |
              next : (T4, NEXT@T3),
              parts : (P, PARTS@T3),
              SFS@T3 >
    < T4 : Tray |
              prev : (T3, NEXT@T4),
              capacity : CAPT@T4,
              nelems : NEL@T4,
              SFS@T4 >
     < P : X:Part | SFS@P >
   OBJSET }
```

(graph constraints expressing model properties are transformed in the same way)

From graph transformation to Maude

Encoding RHS of rules

moveOperator

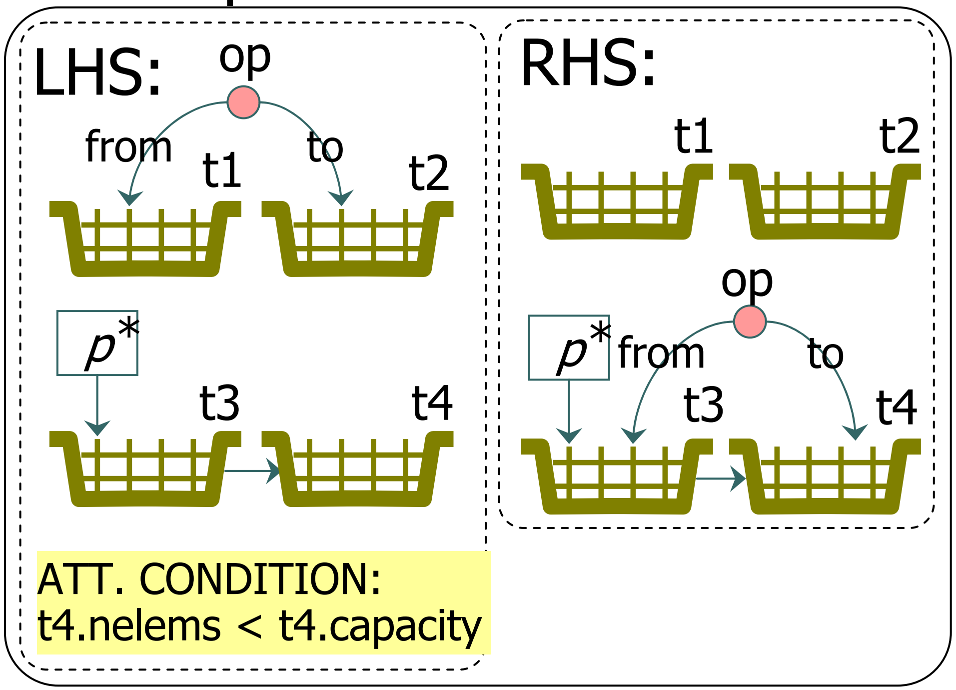



```
…
=> ProductionSystem {
   < T1 : Tray | SFS@T1 >
   < T2 : Tray | SFS@T2 >
   < OP : Operator |
         from : T3,
         to : T4, SFS@OP >
   < T3 : Tray |
         next : (T4, NEXT@T3),
         parts : (P, PARTS@T3),
         SFS@T3 >
  < T4 : Tray |
         prev : (T3, NEXT@T4),
         capacity : CAPT@T4,
         nelems : NEL@T4,
         SFS@T4 >
   < P : X:Part | SFS@P >
  OBJSET }
```

From graph transformation to Maude

Encoding attribute conditions of rules

moveOperator

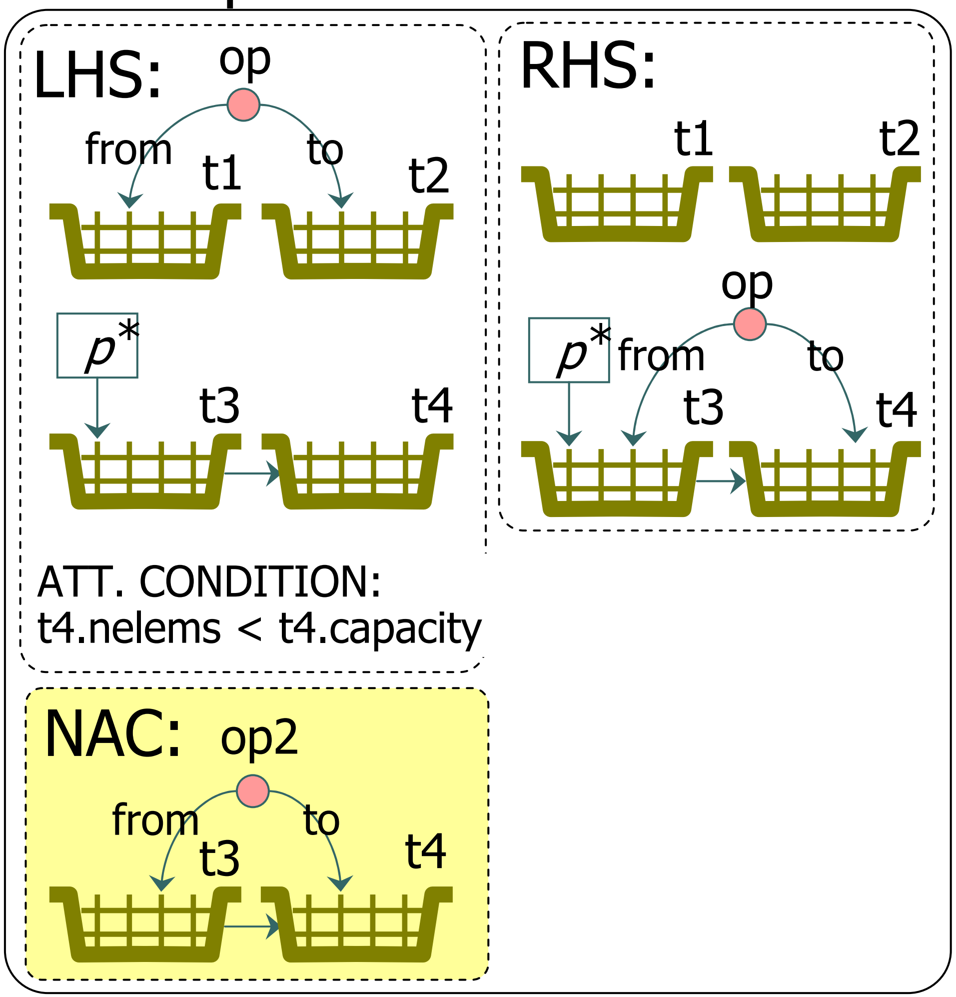


…
if (NEL@T4 < CAP@T4)

From graph transformation to Maude

Encoding negative app. conditions of rules

moveOperator




```
…
if (NEL@T4 < CAP@T4)
/ \
LHS := < T1 : Tray | SFS@T1 >
       < T2 : Tray | SFS@T2 >
       …
       < P : X:PartS | SFS@P >
/ \
MODEL :=
ProductionSystem(LHS OBJECT SET)
/ \
not
NAC@MoveOperator(LHS, MODEL)
```

Analyzing behavior with Maude

Simulation

- Maude specifications can be executed
- Maude commands:
  - `rewrite`: top-down rule-fair strategy
  - `frewrite`: depth-first position-fair strategy
- It is possible to specify upper bounds for the number of rule applications (useful for non-terminating systems)

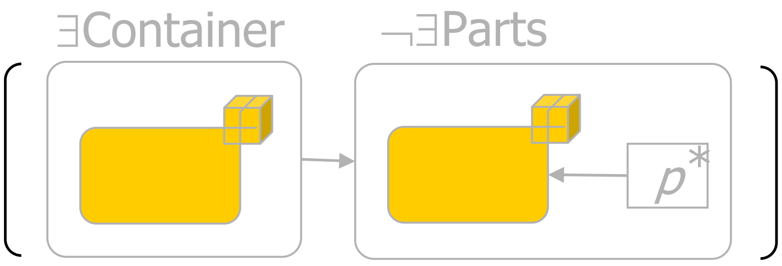
```
rewrite initModel .
```

Analyzing behavior with Maude

Reachability analysis

- We can explore the reachable state space
- Maude commands:
  - `search`: breadth-first strategy to a specified bound
    - input: model properties to be satisfied for the reachable states
    - output: reachable states satisfying the model properties
- E.g. deadlock states where there is a container without parts

```
search [10 ] initModel =>!
  ProductionSystem {
    < ‘co : Container | parts : empty, SFS >
  OBJSET } .
```



Analyzing behavior with Maude
LTL model checking

- Linear temporal logic explicit-state model checker (useful to check temporal logic properties, safety and liveness properties)
- State predicates: `exist`, `stored`, `operated`, *eventually* (`<>`), *henceforth* (`[]`)...
- E.g. check whether a given hammer is eventually stored

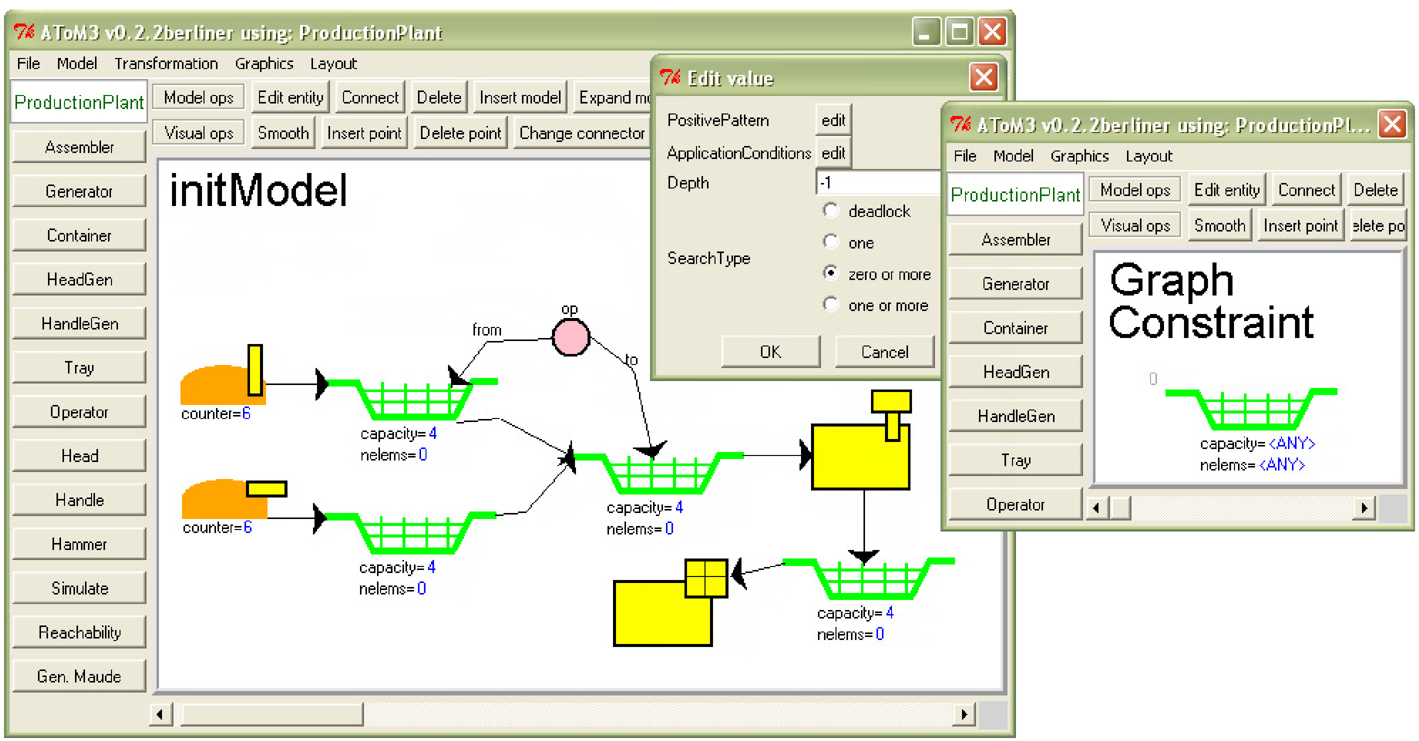
```
reduce modelCheck(initModel,
   [](exist('hammer1) -> <>stored('hammer1)) .

result Bool: true
```

Tool support

AToM^3 + Maude

- Front-end: AToM^3 for the specification of the modeling language, the GT rules and the model properties
- Back-end: Maude for the analysis



Conclusions

- Keep the best of GT and Maude:
  - Visual and intuitive specification of DSVL semantics by GT rules
- Analysis using the Maude toolkit
  - Reachability Analysis
  - Model checking
  - …
- Usable approach: Verification mechanisms are hidden
  - Transformations from GT systems to Maude (and back)

Future work/issues

- GT <-> Maude
  - Annotation of some analysis results to the original modeling language
  - Termination of a rule-based specification
  - Strategies for setting the order in which GT rules are selected and executed
  - Scalability and efficiency
- More bridges…
  - From/to GT to Petri-Nets, pre-post, etc.
  - From/to Maude to other rule-based visual notations
- Add NFP to behavioral specifications (time, probabilities,…)

Thanks!

Analyzing Rule-Based Behavioural Semantics of Visual Modeling Languages with Maude

José Eduardo Rivera — Universidad de Málaga
Esther Guerra — Universidad Carlos III de Madrid
Juan de Lara — Universidad Autónoma de Madrid
Antonio Vallecillo — Universidad de Málaga