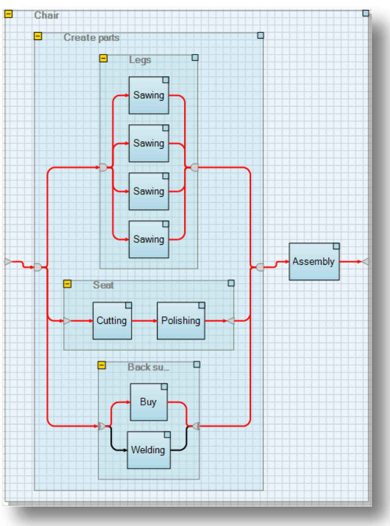
Modelling and Verifying Recursive Workflow Models using Attribute Grammars

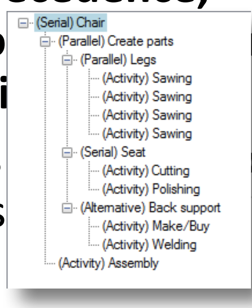
Roman Barták
Charles University in Prague, Czech Republic

Background on Workflows

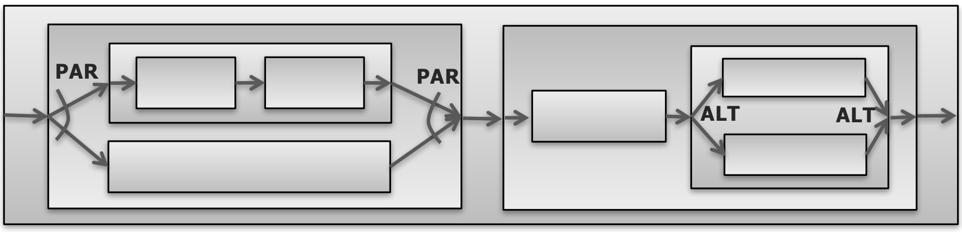
Workflow is a description of a (manufacturing, business, …) process



– **tasks** and **relations** between them
– we use a specific **nested structure** (obtained by task decompositions)
– extra **precedence, synchro**… **d causal constrai**… **dded**
– **process** … f tasks that satisfies … nts



Nested Workflows



workflow is obtained by task decomposition

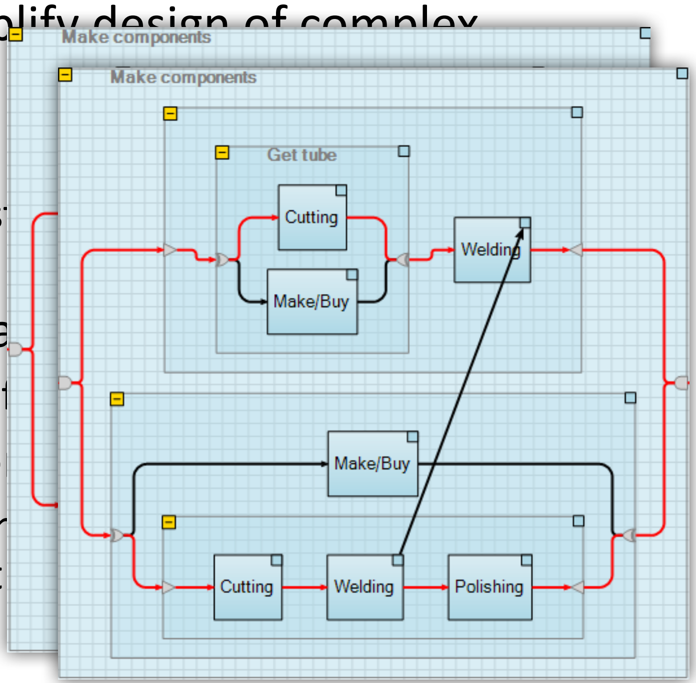
The problem of selecting a valid process containing given tasks is tractable.

However, if we add extra constraints then the problem becomes NP-complete.

Extra Constraints in Nested Workflows

Extra constraints simplify design of complex workflows.

- **Causal** constraints
  - define relations res process
- **Precedence** constra
  - defining ordering of
- **Synchronization** co
  - define temporal syn example starting at



Motivation

Can we represent nested workflows with extra constraints in some "standard" framework?

- to exploit techniques (such as verification) for that framework
- to unify various workflow modeling approaches

Can we represent easily recursion in the workflow (task decomposition contains the top task itself)**?**

- to model planning problems, where the number of actions is unknown in advance

Background on Attribute Grammars

Attribute grammar is a context-free grammar

$S \rightarrow A.B.C$

$A \rightarrow a$

$A \rightarrow a.A$

$B \rightarrow b$

$B \rightarrow b.B$

$C \rightarrow c$

$C \rightarrow c.C$

$a^+b^+c^+$

Background on Attribute Grammars

Attribute grammar is a context-free grammar where:

- extra attributes are added to symbols

$S(n) \rightarrow A(k).B(l).C(m)$

$A(n) \rightarrow a$

$A(n) \rightarrow a.A(m)$

$B(n) \rightarrow b$

$B(n) \rightarrow b.B(m)$

$C(n) \rightarrow c$

$C(n) \rightarrow c.C(m)$

$a^+b^+c^+$

Background on Attribute Grammars

Attribute grammar is a context-free grammar where:

- extra attributes are added to symbols
- constraints connect these attributes

$S(n) \rightarrow A(k).B(l).C(m)$ [n=k=l=m]

$A(n) \rightarrow a$ [n=1]

$A(n) \rightarrow a.A(m)$ [n=m+1]

$B(n) \rightarrow b$ [n=1]

$B(n) \rightarrow b.B(m)$ [n=m+1]

$C(n) \rightarrow c$ [n=1]

$C(n) \rightarrow c.C(m)$ [n=m+1]

$a^nb^nc^n$

Translating Nested Workflows to Attribute Grammars

How to get an attribute grammar equivalent to the nested workflow with extra constraints?

– equivalence: process ~ word

Core ideas:

- nested structure → context-free structure
- task relations → attributes and constraints

Translating the Nested Structure

Using **start and end time attributes** for tasks

Parallel decomposition (a single rule)

$T_i(S_i,E_i) \rightarrow T_{i1}(S_{i1},E_{i1})\dots T_{ik}(S_{ik},E_{ik})$
$[S_i = \min\{S_{i1},\dots,S_{ik}\},\ E_i = \max\{E_{i1},\dots,E_{ik}\}]$

Serial decomposition (a single rule)

a special form of parallel decomposition with extra precedence constraints $[E_i \leq S_{i+1}]$

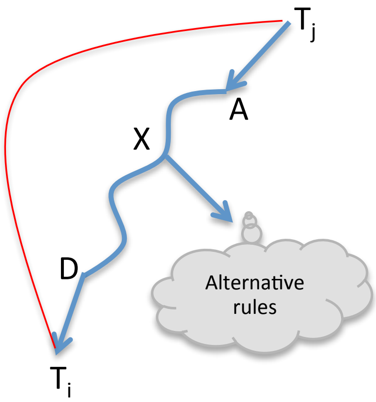
Alternative decomposition (a set of rules)

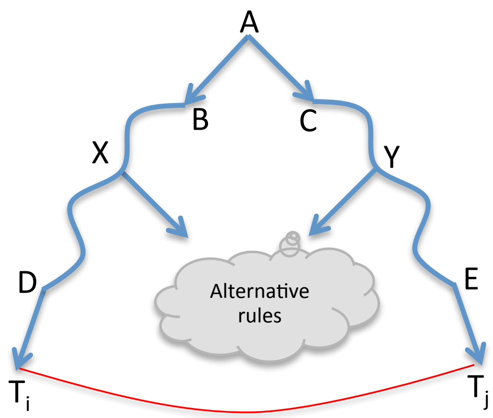
$T_i(S_i,E_i) \rightarrow T_{ij}(S_{ij},E_{ij})\quad [S_i = S_{ij}, E_i = E_{ij}]$

Translating Extra Constraints

All extra constraints are binary (between T_i and T_j)

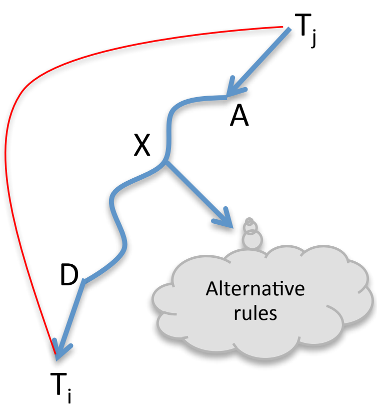
Two possible situations:





Add attribute (M) to each symbol on the path between T_i and T_j.

Translating Extra Constraints (1)



Assume constraint $T_i \Leftrightarrow T_j$

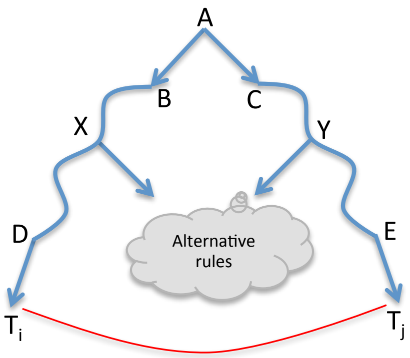
Grammar rules:

- $T_j(\ldots,M) \rightarrow \ldots,A(\ldots,M),\ldots$ [M=1]
- $X(\ldots,M) \rightarrow \ldots$ [M=0]
- $D(\ldots,M) \rightarrow \ldots,T_i,\ldots$ [M=1]

	$T_j \Rightarrow T_i$	T_i mutex T_j	$T_i \rightarrow T_j$	T_i ss T_j	T_i se T_j
$T_j(\ldots,M) \rightarrow \ldots,A(\ldots,M),\ldots$	M=1	M=1	$M \leq S_j$	$M = S_j$	$M = E_j$
$X(\ldots,M) \rightarrow \ldots$	M=0	M=1	---	---	---
$D(\ldots,M) \rightarrow \ldots,T_i,\ldots$	M=1	M=0	$E_i \leq M$	$M = S_i$	$M = S_i$

Translating Extra Constraints (2)

Assume constraint
$T_i \Leftrightarrow T_j$



Grammar rules:

Parallel decomposition of A:

A → ...,B(...,M),...,C(...,M),... []

Alternative decomposition of A:

A → B(...,M)	[M=0]
A → C(...,M)	[M=0]
X(...,M) → ...	[M=0]
Y(...,M) → ...	[M=0]
D(...,M) → ...,T_i,...	[M=1]
E(...,M) → ...,T_j,...	[M=1]

Next steps

We can represent nested workflows with extra constraints using attribute grammars.

Can we verify the workflow/planning domain model represented as an attribute grammar?

– What does it mean to verify the attribute grammar?

– How can we realize the verification algorithm in the case of recursive grammars?

Attribute Grammar Verification Problem

Where is the bug in the following grammar?

S(NS) → A(NA).B(NB)	[NS = NA,NS = NB]
A(N) → a	[N = 1]
A(N) → a.a	[N = 2]
B(N) → b.b	[N = 2]

The attribute grammar verification problem consists of detecting non-terminals and rules that cannot be used in any successful derivation.

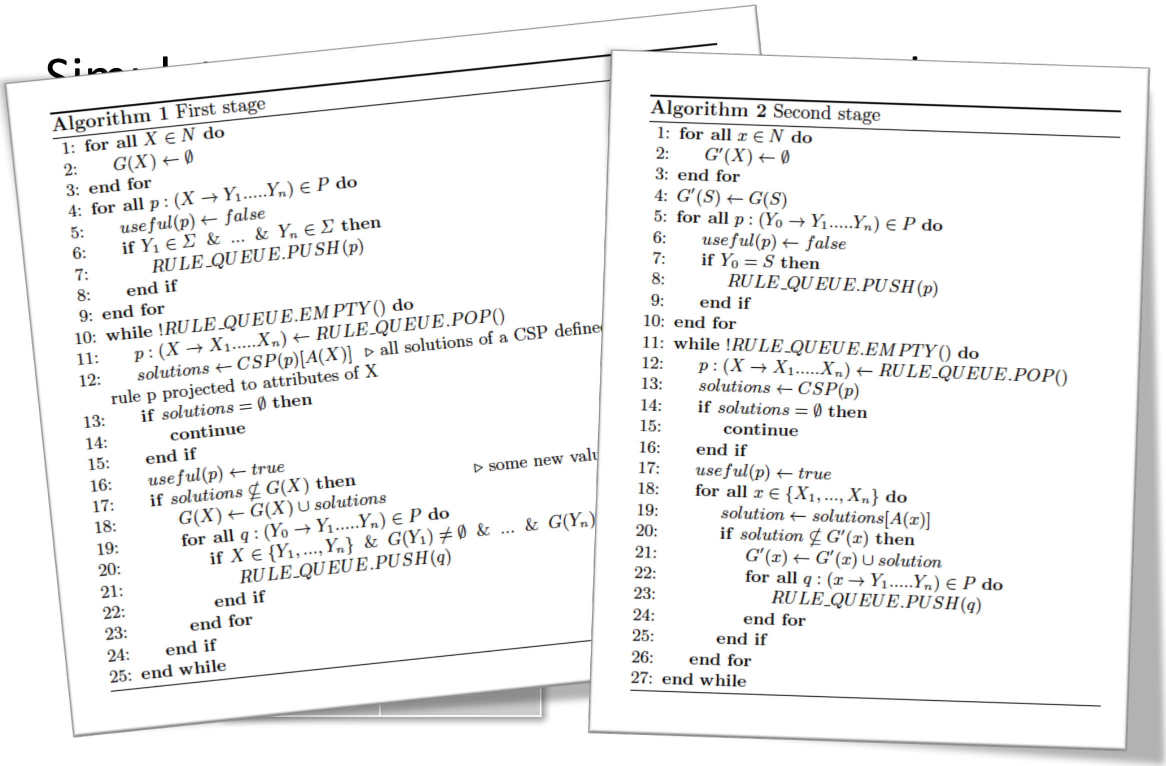
Verification via Translation to CFG

Attribute grammar verification is similar to reduction of a context-free grammar.

1. translate the attribute grammar to a CFG by grounding all attributes
2. reduce CFG

original	grounded	generating non-terminals	reachable non-terminals
S(NS) → A(NA).B(NB) [NS = NA,NS = NB] A(N) → a [N = 1] A(N) → a.a [N = 2] B(N) → b.b [N = 2]	S1 → A1.B1 S2 → A2.B2 A1 → a A2 → a.a B2 → b.b	S2 → A2.B2 A1 → a A2 → a.a B2 → b.b	S2 → A2.B2 A2 → a.a B2 → b.b

Direct Verification via CSP



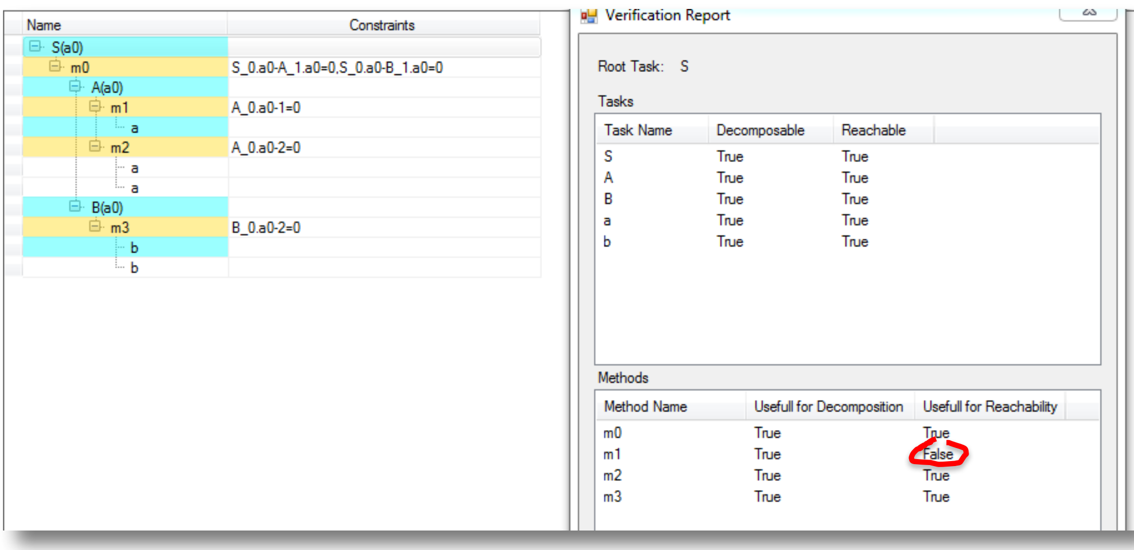
```
Algorithm 1 First stage
for all X ∈ N do
    G(X) ← ∅
end for
for all p : (X → Y1.....Yn) ∈ P do
    useful(p) ← false
    if Y1 ∈ Σ & ... & Yn ∈ Σ then
        RULE_QUEUE.PUSH(p)
    end if
end for
while !RULE_QUEUE.EMPTY() do
    p : (X → X1.....Xn) ← RULE_QUEUE.POP()
    solutions ← CSP(p)[A(X)]   ▷ all solutions of a CSP define
rule p projected to attributes of X
    if solutions = ∅ then
        continue
    end if
    useful(p) ← true
    if solutions ⊄ G(X) then                    ▷ some new valu
        G(X) ← G(X) ∪ solutions
        for all q : (Y0 → Y1.....Yn) ∈ P do
            if X ∈ {Y1, ..., Yn} & G(Y1) ≠ ∅ & ... & G(Yn)
                RULE_QUEUE.PUSH(q)
            end if
        end for
    end if
end while
```

```
Algorithm 2 Second stage
for all x ∈ N do
    G′(X) ← ∅
end for
G′(S) ← G(S)
for all p : (Y0 → Y1.....Yn) ∈ P do
    useful(p) ← false
    if Y0 = S then
        RULE_QUEUE.PUSH(p)
    end if
end for
while !RULE_QUEUE.EMPTY() do
    p : (X → X1.....Xn) ← RULE_QUEUE.POP()
    solutions ← CSP(p)
    if solutions = ∅ then
        continue
    end if
    useful(p) ← true
    for all x ∈ {X1, ..., Xn} do
        solution ← solutions[A(x)]
        if solution ⊄ G′(x) then
            G′(x) ← G′(x) ∪ solution
            for all q : (x → Y1.....Yn) ∈ P do
                RULE_QUEUE.PUSH(q)
            end for
        end if
    end for
end while
```


Implementation

Editor of attribute grammars with linear constraints.

Push-button verification with highlighting all errors in the grammar.



Summary

We can translate nested workflows with extra constraints to attribute grammars.

We can translate STRIPS planning domain models to attribute grammars.

We can fully verify the attribute grammars:

- **by translation to a CFG**
- **directly by solving underlying CSPs**

The downside:

- verification is computationally demanding

This is just the beginning…

- translation of other models (such as hierarchical task networks)
- automated learning of grammars (from example plans/schedules)
- visualization and support for interactive editing

Roman Barták
Charles University, Faculty of Mathematics and Physics
bartak@ktiml.mff.cuni.cz