Verification by abstraction and specialisation of constraint logic programs

John Gallagher$^{1,2}$

$^1$Roskilde University  $^2$IMDEA Software Institute, Madrid

Rich Model Toolkit COST Action Meeting
Malta

Acknowledgements
EU FP7 ENTRA Project
Danish Natural Science Council NUSA Project
Map of the Talk

SOURCE

LINEAR HYBRID AUTOMATA

CLP interpreter

CLP PROGRAM

CONSTRAINT LOGIC PROGRAM

P_0

M[P_0]

TRANSFORM to CLP

+ Petri nets, assembly code, bytecode, functional programs, O-O languages, Z, B, logic programs, ...

IMPERATIVE PROGRAMS

P_1

M[P_1]

HARDWARE

P_2

M_q[P_2]

TRANSFORM CLP PROGRAM

M_q[P_n]

COMPUTE APPROXIMATE MODELS

?- q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.

SOURCE

TRANSFORM

to CLP

CLP PROGRAM

P_0

M[P_0]

TRANSFORM CLP PROGRAM

M[P_1]

COMPUTE APPROXIMATE MODELS

M_q[P_2]

M_q[P_n]

? - q.
From Semantics to CLP

- Linear Hybrid Automata
- Imperative Programs
- Hardware

CLP interpreter + partial evaluation

P₁

P₂

Pₙ

M[P₀]

M[P₁]

M[q[P₂]]

M[q[Pₙ]]

John Gallagher

CLP analysis for verification 3/45
Judgement

\[ \frac{\alpha_1, \ldots, \alpha_n}{\alpha} \text{ where } b \]

CLP

\[ \alpha :\!-\! \alpha_1, \ldots, \alpha_n, b. \]

Note that the definitions of \( \alpha_j, b \) can be “programmed” in CLP (cf. Manuel’s talk).
Current work: modelling the semantics of XC.

- Judgement

\[
\begin{align*}
\langle S_1 \sigma \rangle & \xrightarrow{L} \langle S'_1 \sigma' \rangle \\
\langle (S_1 \parallel S_2) \sigma \rangle & \xrightarrow{L} \langle (S'_1 \parallel S_2) \sigma' \rangle
\end{align*}
\]

- Coq representation

\[
\text{ex_par_1_step : forall s1 s1' s2 st st' l r, exec s1 st l s1' st' r} \\
\rightarrow \text{exec (PAR s1 s2) st l (PAR s1' s2) st' r}
\]

- CLP representation

\[
\text{% ex_par_1_step} \\
\text{exec(par(S1, S2), St, L, par(S11, S2), St1,R) :-} \\
\text{exec(S1, St, L, S11, St1,R).}
\]
Multi-step computation

\[
\langle \text{skip } \sigma \rangle \rightarrow^* \langle \text{skip } \sigma \rangle
\]

\[
\langle S_0 \sigma_0 \rangle \overset{\epsilon}{\rightarrow} \langle S_1 \sigma_1 \rangle \quad \langle S_1 \sigma_1 \rangle \rightarrow^* \langle S_2 \sigma_2 \rangle
\]

CLP

\[
run(\text{skip}, St, \text{skip}, St, 0).
\]

\[
run(S, St, S2, St2, R) :-
exec(S, St, emptyl, S1, St1, R1),
run(S1, St1, S2, St2, R2),
\]

John Gallagher

CLP analysis for verification 6/45
Experiments with offline partial evaluator LOGEN (M. Leuschel)

The CLP interpreter is annotated to indicate
- which calls are unfolded
- a “filter” for each argument controlling generalisation and removal of static structures

Essentially, everything is unfolded except
- the recursive calls to “run"
- the computations on dynamic values

Program’s syntactic structure is removed
Example. XC program semantics instrumented with resource usage (e.g. energy) as final argument.

% factorial

function factnonrec(int n)
    int m=1;
    while(n > 0)
        m = m * n
        n = n - 1
    return m

test7_0(A,B,C) :-
    runeval__2(B,A,1,1,D),
    C is 1+D.
    runeval__2(A,B,C,D,E) :-
        B>0,
        F is B*C,
        G is B-D,
        runeval__2(A,G,F,D,H),
        E is 9+H.
    runeval__2(cns(nat(A)),0,A,B,C) :-
        C is 4.
“Flattening" transformation – removes redundant structure and retains only the dynamic values. This is important to enable analysis of the partially evaluated program.

/*
 runeval(stm(let(n,cns(nat(C))),let(m,cns(nat(D))),seq(ifnz(var(n), seq(seq(asg(m,mul(var(n),var(m))),asg(n,sub(var(n),cns(nat(E)))))), while(var(n),seq(asg(m,mul(var(n),var(m))), asg(n,sub(var(n),cns(nat(F))))),skip),ret(var(m))))),[],A,[],B) :-
 runeval__3(A,F,C,D,E,B). */

runeval__3(A,F,C,D,E,B) :-
    runeval__4(A,F,C,D,C,E,G), B is 1+G.
Transition systems: CLP program encoding reachable states

transition(X,X') ← $c_1(X, X')$.
transition(X,X') ← $c_2(X, X')$.
... ← ...
initState(X) ← $c_{init}(X)$.
reach(X) ← initState(X).
reach(X') ← reach(X), transition(X,X').

The transition relation for a given system can be unfolded in the reach clauses.
$c_i(X, X')$ are constraints over some domain.
Generating assertions from semantics

- Judgement

\[
\sigma \not\models p \\
\langle \text{assert } p \sigma \rangle \rightarrow \langle \text{error} \rangle \\
\text{initState}\langle S \ St \rangle \quad \langle S \ \sigma \rangle \rightarrow^* \langle \text{error} \rangle \\
\text{false}
\]

- CLP representation

```prolog
exec(assert(P),St, error) :- ¬P.
false :- init(S,St), exec(S,St,error).
```
Example: A task scheduler [Halbwachs et al. 94]

(a). Interrupts

\[ c_1 \geq 10 \rightarrow \]
\[ I_1!; c_1 := 0 \]
\[ c_2 \geq 20 \rightarrow \]
\[ I_2!; c_2 := 0 \]

(b). Tasks

\[ x_1 = 4 \land k_1 \leq 1 \rightarrow \]
\[ k_1 := k_1 - 1; x_1 := 0 \]
\[ x_2 = 8 \land k_2 \leq 1 \land k_1 = 0 \rightarrow k_2 := k_2 - 1; x_2 := 0 \]
\[ I_1? \rightarrow \]
\[ k_1 := k_1 + 1 \]
\[ I_2? \rightarrow \]
\[ k_2 := k_2 + 1 \]
\[ k_2 := k_2 - 1; x_2 := 0 \]
\[ I_2? \rightarrow k_2 := k_2 + 1 \]
\[ k_1 := k_1 + 1 \]
\[ I_1? \rightarrow \]
\[ k_1 := k_1 + 1 \]
\[ x_1 = 1 \]
\[ x_2 = 0 \]
\[ x_2 = 8 \land k_2 \leq 1 \land k_1 \geq 1 \]
\[ x_1 = 4 \land k_1 \geq 2 \rightarrow \]
\[ k_1 := k_1 - 1; x_1 := 0 \]
\[ x_1 := 0 \]
\[ x_2 := 1 \]
\[ x_2 := 1 \]
Sample transition of Scheduler.

\[
\text{transition}((J, L, N, P, R, S, G),(A, B, C, D, E, F, 0)) :\neg G < H, \\
1*I = 1*J + 1*(H-G), \\
1*K = 1*L + 1*(H-G), \\
1*M = 1*N + 0*(H-G), \\
1*O = 1*P + 0*(H-G), \\
1*Q = 1*R + 0*(H-G), \\
1* = 1*S + 0*(H-G), \\
K \geq 20, A = I, B = 0, \\
C = M, D = O, E = Q, \\
F = 1.
\]
Binary clause semantics (Codish et al., 1999, derived from a more general “resolvent” semantics for logic programs).

Binary clauses can be derived from a CLP meta-program by partial evaluation (Gallagher, LOPSTR’03)

\[
\begin{align*}
\text{bin}(\text{rev}([X|Xs],Zs),Q) & : - \\
& \text{bin}(\text{rev}(Xs,Ys),Q). \\
\text{bin}(\text{rev}([X|Xs],Zs),Q) & : - \\
& \text{rev}(Xs,Ys), \text{bin}(\text{app}(Ys,[X],Zs),Q). \\
\text{bin}(\text{app}([X|Xs],Ys,[X|Zs]),Q) & : - \\
& \text{bin}(\text{app}(Xs,Ys,Zs),Q). \\
\text{bin}(\text{rev}(X,Y),\text{rev}(X,Y)) & : - \text{true}. \\
\text{bin}(\text{app}(X,Y,Z),\text{app}(X,Y,Z)) & : - \text{true}.
\end{align*}
\]
To be useful for analysis, the partially evaluated CLP program should:

- be of the same size order as the original program,
- predicates correspond (more or less) to program points,
- remove all the source program syntax.

Is this always possible?
The form of the semantic judgements determines the form of the CLP program.

- Big-step semantics generally makes it easier to obtain a “good” partial evaluation.
- Small-step semantics produces programs that are “transition systems” and are “easier to analyse”; but . . .
- For recursive programs, small-step semantics requires a stack to be represented explicitly in the CLP program.
- For big-step semantics, the stack is implicit in the CLP semantics.
- Compound data structures, heap, etc. need careful consideration in order to get an analysable CLP program.
big-step
proc :-
  stmt1,
  stmt2,
  ...
  stmtn.

small-step
proc :-
  stmt1.
  stm1 :-
    stmt2.
    ...
  stmtn-1 :-
    stmtn.
  stmtn :-
    ...

CLP programs (should be) derived systematically from semantics:

- CLP representation of semantic judgements (e.g. operational semantics, proof rules)
- Semantics possibly instrumented or enhanced with traces, etc.
- Partially evaluate semantics wrt a fixed program to get a CLP program
- Filter out the syntactic structures from the interpreter, leaving a CLP program over the domain of the program
Computing (approximate) models of CLP programs

SOURCE
TRANSFORM
to CLP
CONSTR.
LOGIC
PROGRAM
P_0
implies
TRANSFORM
CLP PROGRAM
M[P_0]
DIRECT
EVAL
CLP interpreter
+ partial
evaluation
? q.
? q.
? q.
? q.
COMPUTE APPROXIMATE
MODELS
M[P_1]
M[P_1]
M[q[P_2]]
M[q[P_n]]

LINEAR HYBRID
AUTOMATA
IMPERATIVE
PROGRAMS
HARDWARE
+ Petri nets,
assembly code,
bytecode,
functional programs,
O-O languages,
Z, B,
logic programs,
...

CLP analysis for verification 19/45
CLP model semantics

Here we focus on the model semantics (in contrast to the proof semantics). A model is a set of constrained facts. The “immediate consequence” operator for a CLP program (a generalisation of the standard $T_P$ function).

\[
T_P^C(I) = \begin{cases}
    A \leftarrow C & A \leftarrow B_1, \ldots, B_n, D \in P \\
    \{A_1 \leftarrow C_1, \ldots, A_n \leftarrow C_n\} \in I \\
    \exists \theta \text{ such that } mgu((B_1, \ldots, B_n), (A_1, \ldots, A_n)) = \theta \\
    C' = \bigcup_{i=1,\ldots,n} \{C_i\theta\} \cup D \\
    \text{SAT}(C') \\
    C = \text{proj}_{\text{Var}(A)}(C')
\end{cases}
\]

\[M^C[P] = \text{lfp}(T_P^C)\]
The minimal model is equivalent to the set of derivable facts of the program.

So we can check whether $P \models A$ either

- by checking whether $A \in M[P]$
- or, by running $A$ as a query to $P$ (using a complete proof rule such as tabling (cf. Manuel’s talk)).

Other semantics (e.g. greatest fixpoints) are also relevant to other problems (see later in talk).
The minimal model is computed as the least fixed point of the immediate consequences function $T^C_P$.

This is the limit of the Kleene sequence $\emptyset, T^C_P(\emptyset), T^C_P(T^C_P(\emptyset)), \ldots$.

In general this is not a finite sequence – hence approximation is required.

- either in the model computation (bottom-up) or in the computation (top-down).
The core of verification using static analysis is proof by approximation. Over-approximation gives us sufficient conditions for proving universal formulas over some infinite set.

$$S' \supseteq S \Rightarrow \forall x. x \in S' \implies \forall x. x \in S$$

cf. Manuel’s talk and references for a full account of verification by analysis.
Abstract interpretation of CLP in one picture.

Safety condition: $T \circ \gamma \subseteq \gamma \circ S$
A property-based abstraction is an abstract interpretation.
A proof of safety can be found by approximating the infinite model of this program by a tree automaton (regular type inference, cf. Manuel’s talk).

Operations on a token ring (with any number of processes) (example from Podelski & Charatonik).

```prolog
\begin{align*}
gen([0,1]). 
\text{gen}([0 \mid X]) &\leftarrow \text{gen}(X). 
\text{trans}(X,Y) &\leftarrow \text{trans1}(X,Y). 
\text{trans}([1 \mid X],[0 \mid Y]) &\leftarrow \text{trans2}(X,Y). 
\text{trans1}([0,1 \mid T],[1,0 \mid T]) &\leftarrow \text{trans1}(T,T1). 
\text{trans2}([0],[1]). 
\text{trans2}([H \mid T],[H \mid T1]) &\leftarrow \text{trans2}(T,T1). 
\text{reachable}(X) &\leftarrow \text{gen}(X). 
\text{reachable}(X) &\leftarrow \text{reachable}(Y), \text{trans}(Y,X). 
\end{align*}
```

What are the possible answers for \text{reachable}(X)? Can \(X\) be a list containing more than one '1'?

Intended reachable states
\text{reachable}([0,0,...,1,...0,0]) (lists with exactly one '1')
Example.
applen(X,Y,Z) :- X=0, Y=Z, Y>=0.
applen(X,Y,Z) :- applen(X1,Y,Z1), X = X1+1, Z = Z1+1.
revlen(X,Y) :- X=0,Y=0.
revlen(X,Y) :- revlen(X1,Z),applen(Z,U,Y),X=X1+1, U=1.
false :- revlen(X,Y), X>Y.
false :- revlen(X,Y), X<Y.

Approximation by convex hulls gives:
applen(X,Y,Z) :- X+Y=Z.
revlen(X,Y) :- X=Y

Note that *widenings* are used in these abstract domains, since they are not of finite height.
Abstract interpretation provides a systematic framework for generating sound approximations of the models of CLP programs.

A great variety of useful abstract domains has been developed.

Abstraction can be combined with refinement heuristics to improve the precision of abstractions.

Generic optimisation of fixpoint computation has been studied (program SCCs, worklists, semi-naive evaluations,...).
Proof by CLP transformation

- **SOURCE**
  - LINEAR HYBRID AUTOMATA
  - IMPERATIVE PROGRAMS
  - HARDWARE
  - Petri nets, assembly code, bytecode, functional programs, O-O languages, Z, B, logic programs, ...

- **TRANSFORM to CLP**
  - CLP interpreter
  - partial evaluation

- **TRANSFORM CLP PROGRAM**
  - CONSTRAINT LOGIC PROGRAM
    - $P_0$
    - $P_1$
    - $P_2$
    - $P_n$

- **COMPUTE APPROXIMATE MODELS**
  - $M[P_0]$
  - $M[P_1]$
  - $M[q[P_2]$
  - $M[q[P_n]$

John Gallagher | CLP analysis for verification 29/45
Proof by CLP transformation (overall idea)

Given a CLP program $P_0$, say we wish to show that some atom $A$ is not a consequence.

Suppose we wish to prove that $A$ is a consequence.

Transformation rules preserve the model (wrt to some specified predicates).
The MAP system (Pettorossi, Proietti et al.)

- The MAP system is an automatic program transformation system that automatically proves properties of CLP programs.
- Compares favourably with ARMC, HSF(C) and TRACER (see De Angelis et al. PEPM 2013)

Abstraction techniques related to abstract interpretation are used during the transformations.
The Pro-B system is an automatic program specialisation system that automatically proves properties of CLP programs.

It is now being commercialised.

The main proof technique is program specialisation - again aiming to make program properties explicit.
A generalisation of “magic set” transformations for Datalog

For each predicate $p$, define two predicates $p_{\text{ans}}$ and $p_{\text{query}}$.

Given a program $P$ and query $Q$, derive a program $P_Q$.

$P \models Q$ iff $P_Q \models Q_{\text{ans}}$.

Query-answer transformation allows computation tree semantics to be simulated by model semantics. (The $p_{\text{query}}$ predicates represent calls in the computation tree).
Proof for transformation - Summary

- Model-preserving transformations are applied.
- Proof is obtained when the required property becomes explicit in the transformed program.
- Specialisation wrt a query is a very useful form of transformation – achieved by query-answer transforms, or by various specialisation algorithms.
We start with a CLP representation of a transition system. Each transition is a clause of form 
\[ \text{transition}(X, X') : -c(X, X'), \]
also represented as 
\[ \bar{X} \xrightarrow{c(\bar{X}, \bar{X}')} \bar{X}' . \]
\( c(X, X') \) is a constraint over some constraint domain.
From a transition relation, compute functions \( \text{pre} : 2^S \rightarrow 2^S \), \( \tilde{\text{pre}} : 2^S \rightarrow 2^S \).

- \( \text{pre}(Z) \): the set of possible predecessors of set of states \( Z \).
- \( \tilde{\text{pre}}(Z) \): the set of definite predecessors of set of states \( Z \).
A constraint $c(\bar{X})$ stands for the set of states satisfying $c(\bar{X})$.

\[
\text{pre}(c'(\bar{y})) = \bigvee \{ \exists \bar{y} \ (c'(\bar{y}) \land c(\bar{x}, \bar{y})) \mid \bar{x} \xrightarrow{c(\bar{x}, \bar{y})} \bar{y} \text{ is a transition} \}
\]

\[
\text{pre}(c'(\bar{y})) = \neg (\text{pre}(\neg c'(\bar{y})))
\]

We assume that the constraint solver has a projection ($\exists$-elimination) operation and is closed under boolean operations.
Checking CTL properties

Define a function $\llbracket \phi \rrbracket$ returning the set of states where $\phi$ holds. Compositional definition:

$$
\llbracket p \rrbracket = \text{states}(p)
$$
$$
\llbracket EF\phi \rrbracket = \text{lfp.}\lambda Z. (\llbracket \phi \rrbracket \cup \text{pre}(Z))
$$
$$
\llbracket AG\phi \rrbracket = \text{gfp.}\lambda Z. (\llbracket \phi \rrbracket \cap \text{pre}(Z))
$$

where $\text{states}(p)$ is the set of states where proposition $p$ holds (i.e. a constraint).

Model checking $\phi$:

1. Evaluate $\llbracket \phi \rrbracket$.
2. Check that $I \subseteq \llbracket \phi \rrbracket$, where $I$ is the set of initial states. Equivalently, check that $I \cap \llbracket \neg \phi \rrbracket = \emptyset$. 

John Gallagher
CLP analysis for verification 38/45
Galois connection for partition abstraction

CONCRETE DOMAIN
S (infinite set of states)
A (finite partition of S)
∅ ⊆ S
A ⊆ α\( \gamma \)

ABSTRACT DOMAIN
Galois connection
X Y
U U V V

John Gallagher
CLP analysis for verification 39/45
Galois connection implemented using constraint operations

Assume that the elements of the partition are given by constraints. Let $c_d$ be the constraint defining the partition element $d$.

$$\alpha(c) = \{ d \in A \mid SAT(c_d \land c) \}$$

$$\gamma(V) = \bigvee \{ c_d \mid d \in V \}$$

- SAT can be implemented by an SMT solver. We used Yices (http://yices.csl.sri.com/) interfaced to Prolog.
Abstraction of functions

Given a function

\[ f : 2^S \rightarrow 2^S \]

on the concrete domain, the **most precise** approximation of \( f \) in the abstract domain is

\[ \alpha \circ f \circ \gamma : 2^A \rightarrow 2^A. \]
Abstract checking of CTL properties

Applying this construction to the function $\llbracket \cdot \rrbracket$, obtain a function $\llbracket \phi \rrbracket^a$.

\[
\begin{align*}
\llbracket p \rrbracket^a &= (\alpha \circ \text{states})(p) \\
\llbracket EF \phi \rrbracket^a &= \text{lfp.} \lambda Z. (\llbracket \phi \rrbracket^a \cup (\alpha \circ \text{pre} \circ \gamma)(Z)) \\
\llbracket AG \phi \rrbracket^a &= \text{gfp.} \lambda Z. (\llbracket \phi \rrbracket^a \cap (\alpha \circ \text{pre} \circ \gamma)(Z))
\end{align*}
\]

\ldots

Computation of $\llbracket \phi \rrbracket^a$ terminates. It can be shown that for all $\phi$,

\[
\llbracket \phi \rrbracket \subseteq \gamma(\llbracket \phi \rrbracket^a)
\]

Abstract Model Checking of $\phi$

1. Compute $\llbracket \neg \phi \rrbracket^a$.
2. Check that $I \cap \gamma(\llbracket \neg \phi \rrbracket^a) = \emptyset$.
3. This implies that $I \cap \llbracket \neg \phi \rrbracket = \emptyset$, since $\gamma(\llbracket \neg \phi \rrbracket^a) \supseteq \llbracket \neg \phi \rrbracket$. 
Some Experiments on Linear Hybrid Automata

Arbitrary CTL formulas can be checked (not just $A$-formulas as in standard abstract model checking).

<table>
<thead>
<tr>
<th>System</th>
<th>Property</th>
<th>$A$</th>
<th>$\Delta$</th>
<th>secs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Water Monitor</td>
<td>$AF(W \geq 10)$</td>
<td>5</td>
<td>4</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>$AG(0 \leq W \land W \leq 12)$</td>
<td>5</td>
<td>4</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>$AF(AG(1 \leq W \land W \leq 12))$</td>
<td>5</td>
<td>4</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>$AG(W = 10 \rightarrow AF(W &lt; 10 \lor W &gt; 10))$</td>
<td>10</td>
<td>4</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>$AG(AG(AG(AG(AG(0 \leq W \land W \leq 12)))))$</td>
<td>5</td>
<td>4</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>$EF(W = 10)$</td>
<td>10</td>
<td>4</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>$EU(W &lt; 12, AU(W &lt; 12, W \geq 12))$</td>
<td>7</td>
<td>4</td>
<td>0.04</td>
</tr>
<tr>
<td>Task Sched.</td>
<td>$EF(K2 = 1)$</td>
<td>18</td>
<td>12</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>$AG(K2 &gt; 0 \rightarrow AF(K2 = 0))$</td>
<td>18</td>
<td>12</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>$AG(K2 \leq 1)$</td>
<td>18</td>
<td>12</td>
<td>0.04</td>
</tr>
</tbody>
</table>
Systematic generation of CLP program from semantics
Refinement techniques for arbitrary abstract domains (not just predicate abstractions)
Widening in predicate refinement
Representation and abstraction of memory, heap, stack, etc.
Program transformation vs. abstraction - understand the connections better.