Verification by abstraction and specialisation of constraint logic programs

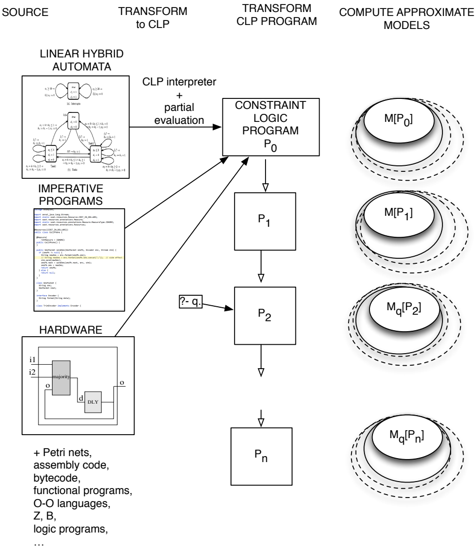
John Gallagher[1,2]

[1] Roskilde University [2] IMDEA Software Institute, Madrid

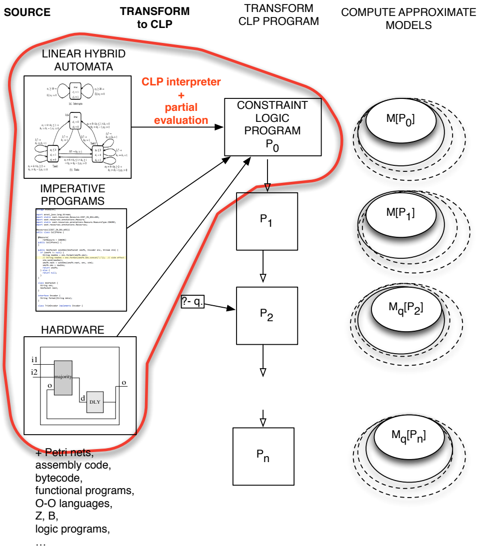
Rich Model Toolkit COST Action Meeting
Malta

Acknowledgements
EU FP7 ENTRA Project
Danish Natural Science Council NUSA Project

Map of the Talk



From Semantics to CLP



From Semantics to CLP interpreters

- Judgement

$$\frac{\alpha_1, \ldots, \alpha_n}{\alpha} \qquad \text{where } b$$

- CLP

$$\alpha \text{ :- } \alpha_1, \ldots, \alpha_n, b.$$

Note that the definitions of α_i, b can be “programmed" in CLP (cf. Manuel’s talk).

From Semantics to CLP interpreters: example

Current work: modelling the semantics of XC.

- Judgement

$$\frac{\langle S_1 \sigma \rangle \xrightarrow{L} \langle S_1' \sigma' \rangle}{\langle (S_1 \parallel S_2) \sigma \rangle \xrightarrow{L} \langle (S_1' \parallel S_2) \sigma' \rangle}$$

- Coq representation

```
ex_par_1_step : forall s1 s1' s2 st st' l r,
    exec s1 st l s1' st' r
-> exec (PAR s1 s2) st l (PAR s1' s2) st' r
```

- CLP representation

```
% ex_par_1_step
exec(par(S1, S2), St, L, par(S11, S2), St1,R) :-
    exec(S1, St, L, S11, St1,R).
```

CLP interpreters: defining a run

- Multi-step computation

$$\frac{}{\langle \text{skip } \sigma\rangle \rightarrow^* \langle \text{skip } \sigma\rangle}$$

$$\frac{\langle S_0\ \sigma_0\rangle \xrightarrow{\epsilon} \langle S_1\ \sigma_1\rangle \quad \langle S_1\ \sigma_1\rangle \rightarrow^* \langle S_2\ \sigma_2\rangle}{\langle S_0\ \sigma_0\rangle \rightarrow^* \langle S_2\ \sigma_2\rangle}$$

- CLP

```
run(skip,St,skip,St,0).
run(S,St,S2,St2,R) :-
    exec(S,St,emptyl,S1,St1,R1),
    run(S1,St1,S2,St2,R2),
```

Partial evaluation

- 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.
```

Structure filtering

“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')	$\leftarrow$	c_1(X,X').
transition(X,X')	$\leftarrow$	c_2(X,X').
$\cdots$	$\leftarrow$	$\cdots$
initState(X)	$\leftarrow$	c_{init}(X).
reach(X)	$\leftarrow$	initState(X).
reach(X')	$\leftarrow$	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

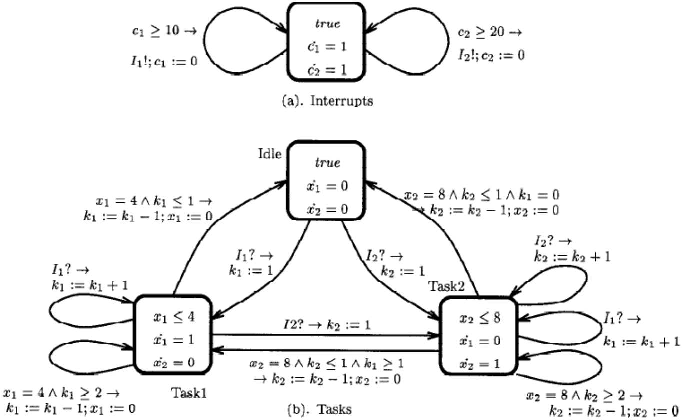
$$\frac{\sigma \not\models p}{\langle \text{assert } p\ \sigma \rangle \rightarrow \langle \text{error} \rangle}$$

$$\frac{\text{initState}\langle S\ St \rangle \quad \langle S\ \sigma \rangle \rightarrow^{*} \langle \text{error} \rangle}{\mathit{false}}$$

- CLP representation

```
exec(assert(P),St, error) :- ¬P.
false :- init(S,St), exec(S,St,error).
```

Example: A task scheduler [Halbwachs et al. 94]



Example Transition for Scheduler

Sample transition of Scheduler.

```
transition((J, L, N, P, R, S, G),(A, B, C, D, E, F, 0)) :-
    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>=20, A=I, B=0,
    C=M, D=O, E=Q,
    F=1.
```

Semantics for termination

- 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)

```
bin(rev([X|Xs],Zs),Q) :-
  bin(rev(Xs,Ys),Q).
bin(rev([X|Xs],Zs),Q) :-
  rev(Xs,Ys), bin(app(Ys,[X],Zs),Q).
bin(app([X|Xs],Ys,[X|Zs]),Q) :-
  bin(app(Xs,Ys,Zs),Q).
bin(rev(X,Y),rev(X,Y)) :- true.
bin(app(X,Y,Z),app(X,Y,Z)) :- true.
```

Good partial evaluations

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?

Big-step vs. small-step semantics

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 vs. small-step semantics - cont'd

big-step

```
proc :-
  stmt1,
  stmt2,
  .  .  .
  stmtn.
```

small-step

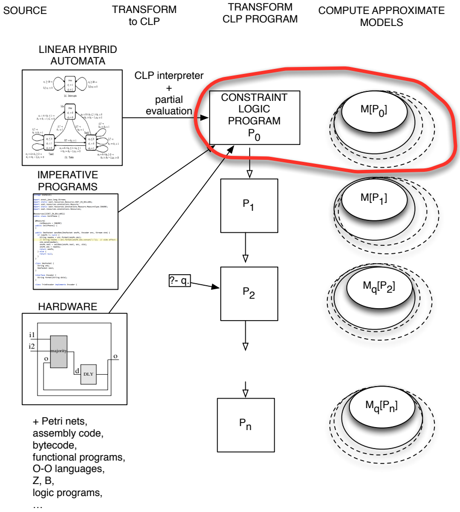
```
proc :-
  stmt1.
stm1 :-
  stmt2.
  .  .  .
stmtn-1 :-
  stmtn.
stmtn :-
  .  .  .
```

Semantics to CLP - Summary

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



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).

$$T_P^{\mathcal{C}}(I) = \left\{ A \leftarrow \mathcal{C} \;\middle|\; \begin{array}{l} A \leftarrow B_1, \ldots, B_n, D \in P \\ \{A_1 \leftarrow \mathcal{C}_1, \ldots, A_n \leftarrow \mathcal{C}_n\} \in I \\ \exists\, \theta \text{ such that} \\ mgu((B_1, \ldots, B_n), (A_1, \ldots, A_n)) = \theta \\ \mathcal{C}' = \bigcup_{i=1,\ldots,n} \{\mathcal{C}_i\theta\} \cup D \\ \mathrm{SAT}(C') \\ \mathcal{C} = \mathrm{proj}_{Var(A)}(\mathcal{C}') \end{array} \right\}$$

$$M^{\mathcal{C}}[\![P]\!] = \mathrm{lfp}(T_P^{\mathcal{C}})$$

Checking program properties

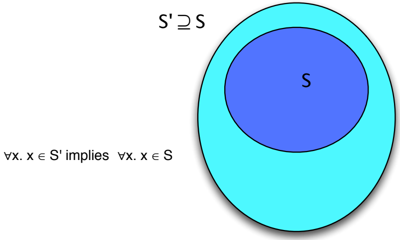
- The minimal model is equivalent to the set of derivable facts of the program.
- So we can check whether $P \models A$ either
  - by checking whether $A \in M[\![P]\!]$
  - or, by running A as a query to P (using a complete proof rule such as tabling (cf. Manuel's talk).
- Other semantics (e.g. greatest fixpoints) are also relevant to other problems (see later in talk).

Computing Fixpoints

- The minimal model is computed as the least fixed point of the immediate consequences function $T_P^{\mathcal{C}}$.
- This is the limit of the Kleene sequence $\emptyset, T_P^{\mathcal{C}}(\emptyset), T_P^{\mathcal{C}}(T_P^{\mathcal{C}}(\emptyset)), \ldots$.
- 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).

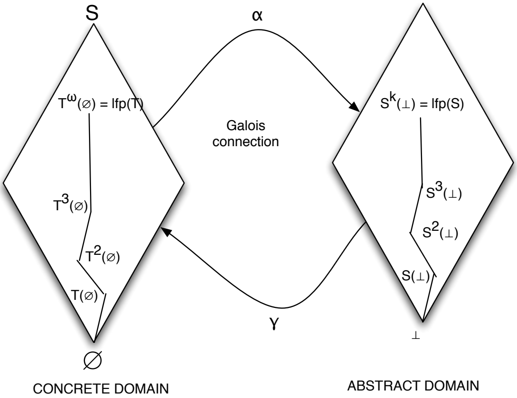
Proofs by Approximation

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.



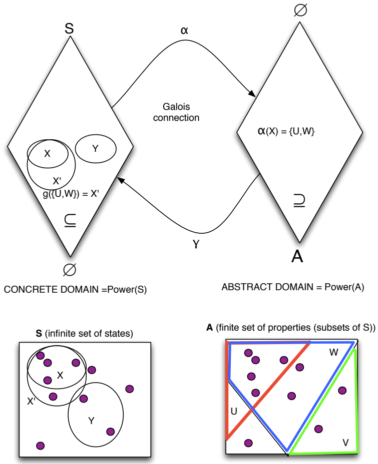
cf. Manuel’s talk and references for a full account of verification by analysis.

Abstract interpretation of CLP in one picture.



Safety condition: $T \circ \gamma \subseteq \gamma \circ S$

A property-based abstraction is an abstract interpretation.



Tree automata abstractions

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

```
gen([0,1]).
gen([0 | X]) ← gen(X).
trans(X,Y) ← trans1(X,Y).
trans([1 |X],[0|Y]) ← trans2(X,Y).
trans1([0,1|T],[1,0 |T]).
trans1([H|T],[H|T1]) ← trans1(T,T1).
trans2([0],[1]).
trans2([H|T],[H|T1]) ← trans2(T,T1).
reachable(X) ← gen(X).
reachable(X) ← reachable(Y), trans(Y,X).
```

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

```
gen([0,1]).
gen([0,0,1]) .
gen([0,0,0,...,1]).
....
```

Intended reachable states
reachable([0,0,...,1,...0,0])
(lists with exactly one 1)

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).

Abstraction by Intervals, polyhedra, $\cdots$

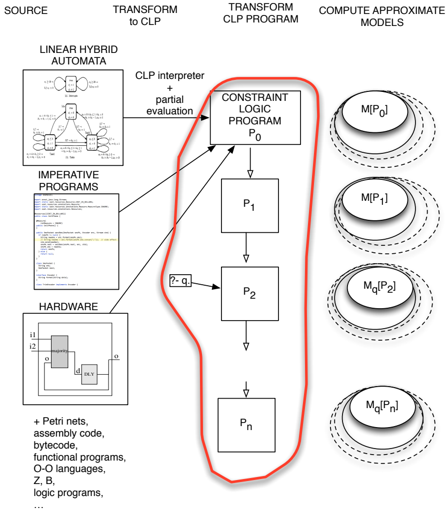
- Example.
  ```
  applen(X,Y,Z) :- X=0, Y=Z, Y>=0.
  applen(X,Y,Z) :- applen(X1,Y,Z1), X = X1+1, Z = Z1+1.
  revlen(X,Y) :- X=0,Y=0.
  revlen(X,Y) :- revlen(X1,Z),applen(Z,U,Y),X=X1+1, U=1.
  false :- revlen(X,Y), X>Y.
  false :- revlen(X,Y), X<Y.
  ```
- Approximation by convex hulls gives:
  ```
  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.

Summary - computing approximate models

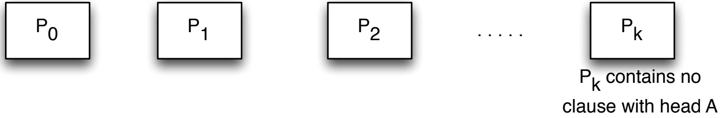
- Abstract interpretation provides a systematic framework for generating sound approximations of the models of CLP programs.
- A great variety of useful abstract domains has been developed.
- Abstraction can be combined with refinement heuristics to improve the precision of abstractions.
- Generic optimisation of fixpoint computation has been studied (program SCCs, worklists, semi-naive evaluations,...).

Proof by CLP transformation

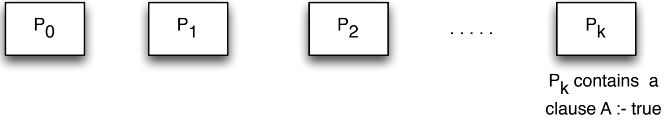


Proof by CLP transformation (overall idea)

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 (Leuschel)

- 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.

Query-answer transformations

- A generalisation of “magic set" transformations for Datalog
- For each predicate p, define two predicates p_{ans} and p_{query}.
- Given a program P and query Q, derive a program P_Q.
- $P \models Q$ iff $P_Q \models Q_{ans}$.

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

Proof for transformation - Summary

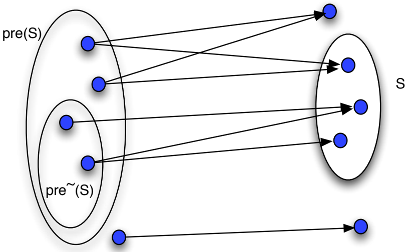
- Model-preserving transformations are applied
- Proof is obtained when the required property becomes explicit in the transformed program.
- Specialisation wrt a query is a very useful form of transformation – achieved by query-answer transforms, or by various specialisation algorithms.

Final topic: Abstract model checking of CLP transition programs

- We start with a CLP representation of a transition system.
- Each transition is a clause of form $\mathit{transition}(X, X') :- c(X, X')$, also represented as $\bar{X} \stackrel{c(\bar{X},\bar{X}')}{\longrightarrow} \bar{X}'$.
- $c(X, X')$ is a constraint over some constraint domain.

pre and $\widetilde{pre}$ functions

From a transition relation, compute functions $pre : 2^S \to 2^S$, $\widetilde{pre} : 2^S \to 2^S$.



- $pre(Z)$: the set of possible predecessors of set of states Z.
- $\widetilde{pre}(Z)$: the set of definite predecessors of set of states Z.

pre and $\widetilde{pre}$ expressed using constraint operations

A constraint $c(\bar{X})$ stands for the set of states satisfying $c(\bar{X})$.

$$
\begin{aligned}
pre(c'(\bar{y})) &= \bigvee\{\exists \bar{y}(c'(\bar{y}) \wedge c(\bar{x},\bar{y})) \mid \bar{x} \xrightarrow{c(\bar{x},\bar{y})} \bar{y} \text{ is a transition}\}\\
\widetilde{pre}(c'(\bar{y})) &= \neg(pre(\neg c'(\bar{y})))
\end{aligned}
$$

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

Checking CTL properties

Define a function $[\![\phi]\!]$ returning the set of states where ϕ holds.
Compositional definition:

$$
\begin{array}{ll}
[\![p]\!] & = \mathsf{states}(p) \\
[\![EF\phi]\!] & = \mathrm{lfp}.\lambda Z.([\![\phi]\!] \cup pre(Z)) \\
[\![AG\phi]\!] & = \mathrm{gfp}.\lambda Z.([\![\phi]\!] \cap \widetilde{pre}(Z)) \\
\ldots &
\end{array}
$$

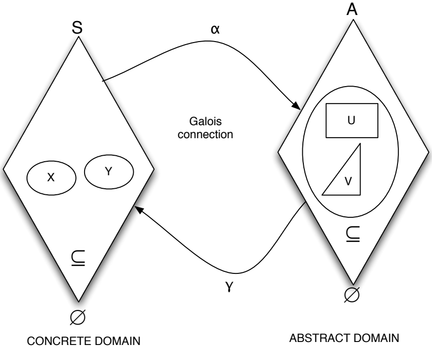
where states(p) is the set of states where proposition p holds (i.e. a constraint).

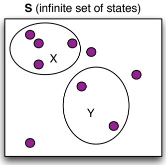
Model checking ϕ:

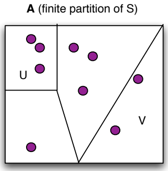
1. Evaluate $[\![\phi]\!]$.
2. Check that $I \subseteq [\![\phi]\!]$, where I is the set of initial states.

Equivalently, check that $I \cap [\![\neg\phi]\!] = \emptyset$.

Galois connection for partition abstraction







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.

$$\alpha(c) = \{d \in A \mid \mathrm{SAT}(c_d \wedge c)\}$$
$$\gamma(V) = \bigvee\{c_d \mid d \in V\}$$

- *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

$$\alpha \circ f \circ \gamma : 2^A \rightarrow 2^A.$$

Applying this construction to the function $[\![.]\!]$, obtain a function $[\![\phi]\!]^a$.

$$\begin{array}{ll} [\![p]\!]^a & = (\alpha \circ \mathsf{states})(p) \\ [\![EF\phi]\!]^a & = \mathsf{lfp}.\lambda Z.([\![\phi]\!]^a \cup (\alpha \circ pre \circ \gamma)(Z)) \\ [\![AG\phi]\!]^a & = \mathsf{gfp}.\lambda Z.([\![\phi]\!]^a \cap (\alpha \circ \widetilde{pre} \circ \gamma)(Z)) \\ \ldots & \end{array}$$

Computation of $[\![\phi]\!]^a$ terminates. It can be shown that for all ϕ,

$$[\![\phi]\!] \subseteq \gamma([\![\phi]\!]^a)$$

.

Abstract Model Checking of ϕ

1. Compute $[\![\neg\phi]\!]^a$.
2. Check that $I \cap \gamma([\![\neg\phi]\!]^a) = \emptyset$.
3. This implies that $I \cap [\![\neg\phi]\!] = \emptyset$, since $\gamma([\![\neg\phi]\!]^a) \supseteq [\![\neg\phi]\!]$.

Some Experiments on Linear Hybrid Automata

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

System	*Property*	A	Δ	*secs.*
Water Monitor	$AF(W \geq 10)$	5	4	0.02
	$AG(0 \leq W \wedge W \leq 12)$	5	4	0.01
	$AF(AG(1 \leq W \wedge W \leq 12))$	5	4	0.02
	$AG(W = 10 \rightarrow AF(W < 10 \vee W > 10))$	10	4	0.05
	$AG(AG(AG(AG(AG(0 \leq W \wedge W \leq 12)))))$	5	4	0.02
	$EF(W = 10)$	10	4	0.01
	$EU(W < 12, AU(W < 12, W \geq 12))$	7	4	0.04
Task Sched.	$EF(K2 = 1)$	18	12	0.53
	$AG(K2 > 0 \rightarrow AF(K2 = 0))$	18	12	0.30
	$AG(K2 \leq 1)$	18	12	0.04

CLP-based verification: Some directions

- 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.

THE END