Formal Specification and Analysis of Robust Adaptive Distributed Cyber-Physical Systems

Carolyn Talcott
(and the Soft Agents Team)

SFM
Quantitative Evaluation of Collective Adaptive Systems
June 2016

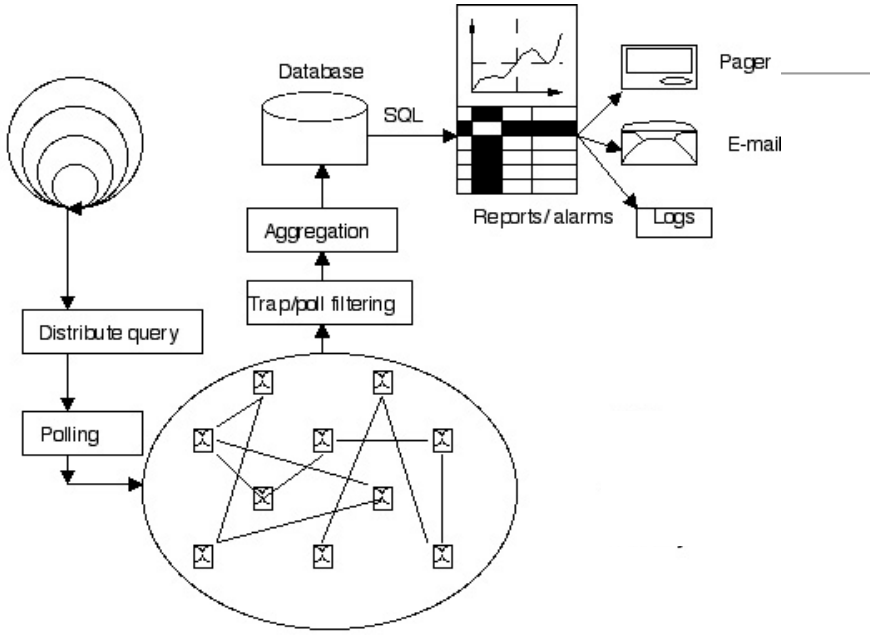
Vision

- Multiple agents with diverse computational and physical capabilities collaborate to solve problems
  - they act using only local knowledge (sensed and heard)
  - they operate in open, unpredictable environments
  - communication may be disrupted and/or delayed
- Robustness and fault tolerance achieved by diversity, redundancy, adaptability, interchangeability
- How do we gain confidence in designs before going to the effort of building and testing in the field?
- Executable formal models to the rescue!

Formal Modeling Methodology

Model builder

data

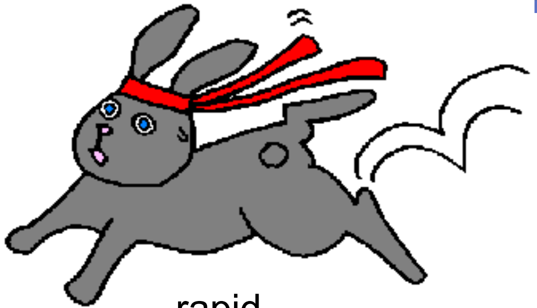


model

asking questions

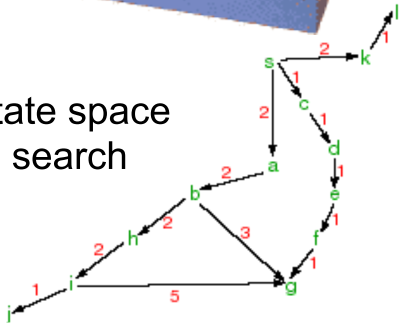
$S \models \Phi$

model checking



rapid prototyping

state space search



Plan

- Part I
    - Introduction/Overview
    - Modeling
- Part II
    - Rewriting Logic and Maude
    - The soft agent framework I
- Part III
    - The soft agent framework II
    - Patrol bot case study
    - Surveillance drone case study

Part I

Introduction/Overview

Examples I

- Google's self driving cars (see them in Mt. View!)

- Amazon's warehouse automation, last mile delivery drones (to appear).

- PINC solutions PINC Air is an aerial sensor platform that operates outdoors and indoors to efficiently inventory hard-to-reach assets using an array of sensors that include GPS, RFID, OCR and Barcode readers.

Examples II

- The Knightscope security robots attended a Stanford drones swarm event, roaming the busy plaza without bumping into people or other objects. Their normal job is surveillance of limited areas, looking for problems.

- Liquid Robotics wave glider is a surfboard size robot that is powered by the ocean, capable of multiple modes of communication and of carrying diverse sensors. Wave gliders are able to autonomously and safely navigate from the US to Australia, and able to call home when pirates try bot-napping gliders, in addition to collecting data.

Are we done?

- Maybe industry has already achieved the vision ?
- There are autonomous agents carrying out non-trivial tasks.
- But
  - single agents working alone
  - special purpose
  - years of research and testing
- What are the design principles, tools?

Examples III

- Autonomous flying quadrotor robots (Vijay Kumar Lab, U. Penn)
  - sensors: inertial measurement units, cameras, laser range scanners, altimeters and/or GPS sensors.
  - capabilities: navigation in 3-dimensional space, sensing other entities, and forming ad hoc teams.
  - potential applications: construction, search and rescue, first response, and precision farming.

Examples IV

- EU ASCENS project: robot swarms with both autonomous and collective behavior.

- SRIs NCPS project cyber-physical testbed demonstration of a surveillance team consisting of robots, quadcopters, and Android devices cooperating to travel to a site of interest, take a picture and deliver it to the interested party.

Desiderata

- Localness
  - agents must operate based on local knowledge
    - what they can observe / infer
    - what they can learn by knowledge sharing
- Safety/Liveness
  - an agent should remain safe (healthy) and do no harm
  - an agent should be able to act based on current information
  - should not require or need to rely on consensus formation
  - should be able to respond to change/threats in a timely manner
- Softness – for robustness and adaptability
  - binary satisfaction is unrealistic
  - rigid constraints are likely to fail

Questions

- Question: How accurate / comprehensive does the local knowledge need to be in order to be able to (sufficiently) satisfy a given goal?

- Question: What does an agent need to monitor? How often?

- Question: what time/space scope should be considered to (sufficiently) satisfy global goals by local actions?

Soft agents from 20k feet/meters

- An agent model with explicit Cyber and Physical aspects
- Declarative control via soft constraints
- Loosely-coupled Interaction through Sharing of Knowledge with a Partial Order (POKS)
- Formalized in Rewriting Logic/Maude

An agent model with explicit Cyber and Physical aspects

- A soft agent system is a collection of agents plus an environment object.
- CP/soft agents have the form
  - [id : class | lkb: localkb, ckb: cachedkb, evs: events, ...]
    - localkb is the agents local knowledge
    - cachedkb is knowledge to be opportunistically shared
    - events is the set of pending actions, tasks, knowledge to process
- The environment object has the form [eid | ekb]
    - envkb represents the physical environment
- The framework provides
  - rules for executing tasks and actions and for communication
  - templates for specifying agent actions using soft constraints

Declarative Control with Soft Constraints

- Behavior is driven by consideration of
  - local conditions (resource availability, environment)
  - different concerns (saving energy, getting credit, …)
  - potential contributions to global goals
- Ranking mechanisms allow different constraints to be combined
- Agents solve soft constraint problems parameterized by local knowledge to decide moves
- Supports reasoning about sufficient satisfaction
- Some challenges
  - Collaborative constraints – maybe another agent can do a task more effectively, but no task is left to others by all.
  - Inferring combined guarantees (concerns, agents not independent)
  - Deriving local constraint systems from `global/external' goals.

Loosely-coupled Interaction through Sharing of Knowledge with a Partial Order (POKS)

- Knowledge items can be sensor readings, locally computed solutions, community goals, etc.
- Knowledge items may be time-stamped
- Knowledge is shared opportunistically
  - provides delay/disruption-tolerant knowledge dissemination
  - does not require global coordination or infrastructure
  - supports entire spectrum between autonomy and cooperation

- The partial order captures
  - replacement – eliminate stale information, redundant goals
  - subsumption – logically redundant

Patrol Bot case study

- Setting:
  - a 2D grid with one or more patrol bots
  - a charging station in the center
  - possible environment effects such as wind driving bots off track
- Patrol bots move from edge to edge of a grid along a preferred strip.
- Moving takes energy
  - a bot needs to decide when to visit the charging station.
- Requirements:
  - A patrol bot should not run out of energy
  - At most one patrol bot can occupy a grid location at a give time
  - Each bot should keep patrolling (when not charging)

Surveillance drone case study

- Setting:
  - a 3D grid with some targets to be monitored and a charging/battery exchange station
  - one or more drones flying around taking photos and posting them
- Requirements:
  - A drone should not run out of energy
  - Drones should maintain separation (safety envelop)
  - There should always be a recent photo of each target
- Challenges:
  - Collision avoidance
  - Cooperation to achieve monitoring requirements
  - Imprecise flight, wind currents, obstacles

Modeling

Executable Symbolic Models

- Describe system states and rules for change
- From an initial state, derive a transition graph
    - nodes -- reachable states
    - edges -- rules connecting states
- Watch it run
    - Execution path — a set of transitions (and associated nodes) that can be fired in some order starting with the initial state (aka computation / derivation)
    - Execution strategy -- picks a path

Symbolic analysis -- answering questions

- Static analysis
- Forward/backward simulation
- Search reachability analysis
- Model checking -- do all executions satisfy ϕ, if not find counter example
- Constraint solving
- Meta analysis

Symbolic Analysis I

- Static Analysis
  - sort hierarchy / type system
  - control flow / dependencies
  - coherence/convergence checking
- Simulation from a given state (prototyping)
  - run model using a specific execution strategy
- Symbolic simulation — starting with a state pattern (with variables)
  - rules applies using unification

Symbolic Analysis II

- Forward search from a given state
  - breadth first search of transition graph
  - find ALL possible outcomes
  - find only outcomes satisfying a given property
- Backward search from a given state S
  - run a model backwards from S
  - find initial states leading to S
  - find transitions/rules that might contribute to reaching S

Symbolic Analysis III

- Model checking
  - determines if all pathways from a given state satisfy a given property, if not a counter example is returned
  - example property:
    - if at the eastern edge, then eventually at the western edge
  - counter example:
    - an execution/pathway in which a bot reaches the eastern edge, but fails to get back to the western edge (runs out of energy, oscillates …)

Symbolic Analysis IV

- Constraint solving
  - Find values for a set of variables satisfying given constraints.
  - MaxSat deals with conflicts
    - weight constraints
    - find solutions that maximize the weight of satisfied constraints
  - Prune search paths by checking accumulated constraints

Symbolic Analysis V

- Meta analysis
  - reasoning about the model itself
  - specfying execution/search strategies
  - check that transitions satisfy some property, for example preserving number of agents, maintaining knowledge
  - transform a model and property to another logic (for access to tools)
  - transform a model to another executable language for efficient execution on device.

Part II

The soft agent framework I

Rewriting Logic and Maude

What is Rewriting Logic?

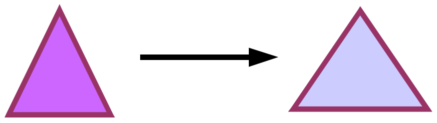
- Rewriting Logic is a logical formalism that is based on two simple ideas
  - states of a system are represented as elements of an equationally specified algebraic data type
  - the behavior of a system is given by local transitions between states described by rewrite rules
- It is a logic for executable specification and analysis of software systems, that may be concurrent, distributed, or even mobile.
- It is also a (meta) logic for specifying and reasoning about formal systems, including itself (reflection!)

Rewriting Logic Formally

- Rewrite theory: (Signature, Labels, Rules)
- Signature: (Sorts, Ops, Eqns) -- an equational theory
  - Specifies data types and functions that represent system state
  - Sorts are partially ordered
- Rules have the form   label : t => t' if cond
- Rewriting operates modulo equations
  - rules apply locally
  - rule application generates computations (pathways)

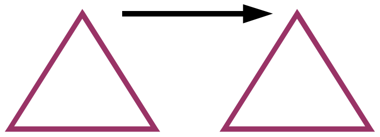
Rewriting with rules

one step rewrite:

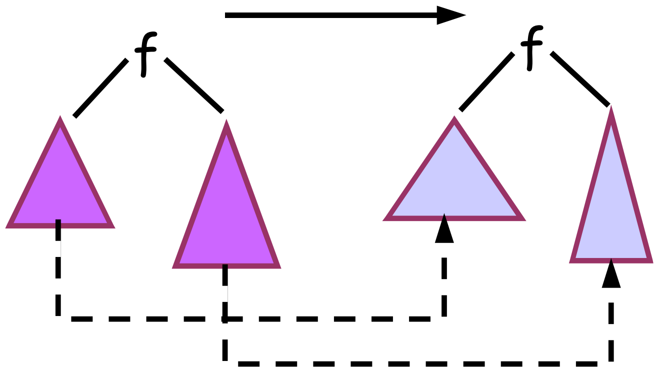


closed under

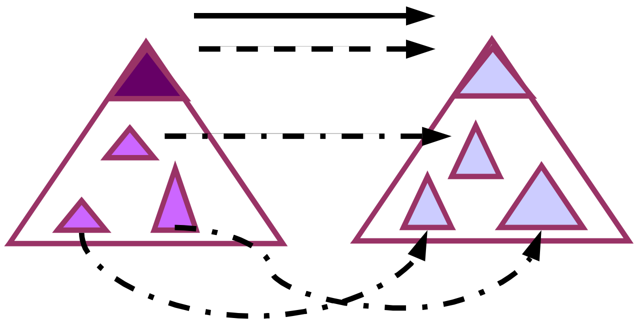
reflexivity:



congruence:



replacement:



Maude

- Maude is a language and tool based on rewriting logic
- See: http://maude.cs.uiuc.edu
- Features:
  - Executability -- position /rule/object fair rewriting
  - High performance engine --- {ACI} matching
  - Modularity and parameterization
  - Builtins -- booleans, number hierarchy, strings
  - Reasoning: search, model-checking, SMT solving
  - Reflection -- using descent and ascent functions

MAUDE Functional Modules:the Equational Part

- A Maude functional module (fmod) describes an equational theory specifying data types, data constructors, and functions

```
fmod <modname> is
  <imports>       *** reuse, composition
  <sorts>         *** data types and subtyping
  <opdecls>       *** names and arities of operations
  <eqns>          *** how to compute functions
endfm
```

- The semantics of an fmod is an initial algebra
  - no junk---every data value is constructable
  - no confusion---two terms are equal only if forced by the equations

Specifying lists

- Lists of natural numbers are specified by

```
fmod NATLIST is
    pr NAT .
    sort NatList .  subsort Nat < NatList .
    op nil : -> NatList .
    op _ _ : NatList NatList -> NatList [assoc id: nil] .
       ....
  endfm
```

Example lists: `nil,  1 2 3`

- Notes
  - the use of `empty syntax'  `_ _`
  - use of ***axioms*** `[assoc id: nil], comm` also possible.
  - axioms allow for efficient matching modulo

A function on lists

- Extend NATLIST with a function to sum the elements of a list

```
var n : Nat. var nl : NatList .

op sum : NatList -> Nat .
eq sum(nil) = 0 .
eq sum(n nl) = n + sum(nl) .
```

- Equations are used to reduce terms to canonical form

```
sum(1 2 3) = 1 + sum(2 3)
           = 1 + 2 + sum(3)
           = 1 + 2 + 3
           = 6
```

MAUDE System Modules: the Rules Part

- System dynamics are specified using rewrite rules

```
mod <modname> is
*** functional part
  <imports>        *** reuse, modularity
  <sorts>          *** data types and subtyping
  <opdecls>        *** names/and arities of operations
  <eqns>           *** how to compute functions
***
  <rules>
endm
```

- A system module defines a set of computations (derivations) over the data types specified by the functional part.

Rule declarations

- The <rules> part of a system module consists of rule declarations having one of the forms

```
     rl[<id>]: <lhs> => <rhs>   .

    crl[<id>]: <lhs> => <rhs> if <cond> .
```

- <lhs>, <rhs>, <cond> are terms possibly containing variables.

- A rule applies to a term T if there is a substitution S (mapping variables to terms) such that S(<lhs>) is a subterm of T (<lhs> matches a subterm of T) and S(<cond>) rewrites to true.

- In this case T can be rewritten by replacing the matched subterm by the matching instance of <rhs>, S(<rhs>).

Rule Application

- Suppose we have the following rules (in a suitable module).

```
rl[fa2b]: f(a,x) => f(b,x) .
rl[hh2h]: h h(y) => h(y) .
```

- then

```
g(c,f(a,d)) => g(c,f(b,d))
```

- and

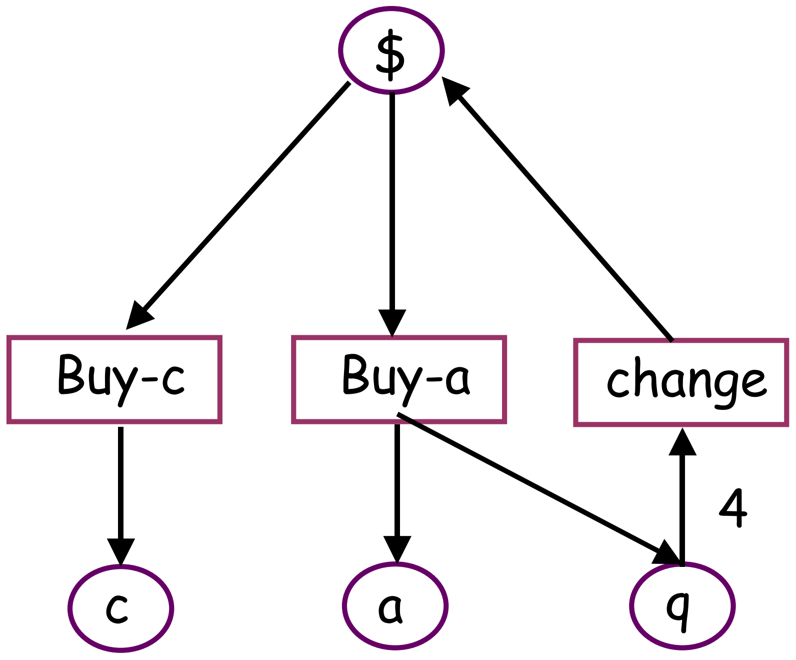
```
h h(g(c,f(a,d))) => h(g(c,f(b,d)))
```

- also using replacement.

- Before applying a rewrite rule, Maude reduces a term to canonical form using equations. The rule lhs is not reduced.

Vending Machine Specification

Petri Net



Maude

```
mod VENDING-MACHINE is
  sorts Coin Item Place Marking .
  subsorts Coin Item < Place < Marking .
  op null : -> Marking .
             *** empty marking
  ops $ q : -> Coin .
  ops a c : -> Item .
  op _ _ : Marking Marking -> Marking
           [assoc comm id: null] .
           *** multiset using axioms
  rl[buy-c]: $ => c .
  rl[buy-a]: $ => a q .
  rl[change]: q q q q => $ .
endm
```

Using Maude to analyze the Vending Machine

- What is one way to use 3 $s?   Use the rewrite command (rew).

```
Maude> rew $ $ $ .
result Marking: q a c c
```

- How can I get 2 apples with 3 $s?   Uses the search command.

```
Maude> search $ $ $ =>! a a M:Marking .

Solution 1 (state 8)
M:Marking --> q q c

Solution 2 (state 9)
M:Marking --> q q q a

No more solutions.
states: 10  rewrites: 12)
```

Model Checking I

- The job of a model checker is to determine if M |= P (M satisfies P) where M is a `model' and `P' is a property.

- In our case a model is a Maude specification of a system together with a state of interest.

- A property is a `temporal logic' formula to be interpreted as a property of computations of the system (linear sequences of states generated by application of rewrite rules).

Model Checking II

- Temporal formulas are built from propositions about states, using boolean connectives, and temporal operators [] (always) and <> (eventually).
- Satisfaction for M |= A, for proposition A, is axiomatized by equations
    - M |= P if C |= P for every computation C of M
    - C |= A if the first state of C satisfies A
    - C |= [] P if every suffix of C satisfies P
    - C |= <>P if some suffix of C satisfies P

Model checking: Defining properties

```
mod MC-VENDING-MACHINE is
  inc VENDING-MACHINE .  inc MODEL-CHECKER .  inc NAT .

  op vm : Marking -> State [ctor] .
  op countPlace : Marking Place -> Nat .
  op value : Marking -> Nat .  *** in units of quarters
  ....

  ops fiveQ lte4Q : -> Prop .
  ops nCakes nApples val : Nat -> Prop .

  eq vm(M) |= fiveQ = countPlace(M,q) == 5 .
  eq vm(M) |= lte4Q = countPlace(M,q) <= 4 .
  eq vm(M) |= nApples(n) = countPlace(M,a) == n .
  eq vm(M) |= val(n) = value(M) == n .
endm
```

Model checking the vending machine I

- Starting with 5 $s, can we get 6 apples without accumulating more than 4 quarters? Model check the claim that we can't.

```
Maude>
   red modelCheck(vm($ $ $ $ $),[]~(lte4Q U nApples(6))) .
result ModelCheckResult: counterexample(
  {vm($ $ $ $ $),'buy-a}
  {vm($ $ $ $ q a),'buy-a}
  {vm($ $ $ q q a a),'buy-a}
  {vm($ $ q q q a a a),'buy-a}
  {vm($ q q q q a a a a),'change}
  {vm($ $ a a a a),'buy-a}
  {vm($ q a a a a a), 'buy-a},
  {vm(q q a a a a a a),deadlock})
```

- A counter example to a formula is a pair of transition lists representing an infinite computation which fails to satisfy the formula. A transition is a state and a rule identifier. The second list (deadlock) represents a loop.

Model checking the vending machine II

- Starting with 5 $s, can we get 5 quarters then 6 apples?

```
Maude> red modelCheck(vm($ $ $ $ $),
               []~(<>fiveQ /\ (fiveQ |-> nApples(6))) .
result ModelCheckResult: counterexample(...)
```

- Is value conserved?

```
Maude> red modelCheck(vm($ $ $ $ $),[]val(20) .
result Bool: true
```

Soft agent framework organization

- Lib -- the framework -- used by all models
    - time.maude -- essential features of time, discrete or dense
    - soft-agents.maude
        - KNOWLEDGE -- knowledge constructions, and partial order
        - EVENTS --- task and action data types and api
        - AGENT-CONF — configuration elements
    - rules.maude -- for executing tasks, actions, passing time
- Models --- specific agent models
    - PatrolBot --- patrol bot specific elements and semantics
    - Drone-live —- surveillance drone specification
    - ….

Specifying Agents and Configurations

Specifying an agent

- Cyber part (tasks)
  - processing information
  - deciding actions, based on
    - local view
    - agent’s model of action effects
- Physical part (actions)
  - specifies effects of actions
  - accounts for environment effects, faults ...
- Cyber and physical states represented as sets of knowledge items
- Events formalize activities to be done: tasks and actions.

Formalizing knowledge

- Sorts
  - Id < IdSet -- agent identifiers
  - Item < KB -- Knowledge items
    - PKItem < KItem -- Persistent KItems
    - TKItem < KItem -- Transient KItems (have time stamp)
- Partial order on KItems
  - kitem0 << kitem1 -- kitem1 subsumes/replaces kitem0
  - model specific
- Basic Knowledge processing: addK(kb0:KB,kb1:KB)
  - the union of kb0 and kb1 with subsumed kitems removed

Builtin knowledge

- Notation: x:X a variable x of sort X, exp :: X an expression of sort X
- sort Class
  - class(id:Id,cl:Class) :: PKItem
  - agents class -- groups agents with comment capabilities

- clock(t:Time) :: KItem -- current time
  - eq clock(t0) << clock(t1) = t0 < t1 .

- sort Loc --- represenation of location
  - atloc(id:Id,l:Loc) @ t:Time ::TKItem —the location of agent id at time t
  - atloc(id:Id,l0:Loc) @ t0:Time << atloc(id:Id,l1:Loc) @ t1:Time
    - if t0:Time < t1:Time

Events: sorts and constructors

- Task -- cyber activity
  - tick — builtin task, maybe all that is needed
- Action — physical activity

- IEvent DEvent < Event
  - IEvent — immediate events
    - rcv(kb:KB) :: IEvent — incoming knowledge to process
  - DEvent — delayed events
    - task:Task @ t:Time — task to be carried out after delay t
    - act:Action @ t:Time ; d:Time
      - action to be executed after delay t:Time for duration d:Time

Task handling

- Agents have class specific procedures for executing scheduled tasks
- The interface to the framework is the function doTask
- doTask returns a KBEventKBSet:
  - a set of triples {kb0:KB,evs:EventSet,kb1:KB}
    - kb0:KB — new local knowledge base
    - evs:EventSet — updated event set, new actions, tasks
    - kb1:KB — knowledge to post for sharing

doTask scheme

```
ceq doTask(cl,id,tick,ievs,devs,ekb,lkb) =
   if acts == none then {lkb2, devs (tick @ botDelay), none }
   else selector(doTask$(id,lkb2,devs (tick @ botDelay),acts))  fi
if lkb0 := handleS(cl,id,lkb,ievs)— process new knowledge
/\ lkb1 := getSensors(id,ekb) — read local sensors
/\ lkb2 := proSensors(id,lkb0,lkb1) — process this information
/\ acts0 := myActs(cl,id,lkb2)—possible actions in the current situation
/\ acts := solveSCP(id,lkb2,acts0) .
  —top ranked actions according to the agents preferences
  — formalized by soft constraints using function doAct
```

API functions underlined.

Task handling

- In the simplest case
    - handleS(cl,id,lkb,rcv(kb) just adds kb to kb, with subsumption
    - proSensors(id,lkb0,lkb1) adds lkb1 to lkb0, with subsumption
        - could do more, for example keep a history of locations, keep statistics on temperature, ….
- solveSCP(id,lkb2,acts0) uses doAct(id,lkb,act)
    - which formalizes the agents model of effects of actions
- doTask$(id,lkb2,devs,acts) — makes a triple for each act in acts
    - by default: {lkb2,devs (act @ 0 ; 1), tell(lkb1)}
    - tell(lkb1) chooses what information to share

Soft constraint problems

- A soft constraint problem (SCP) consists of
  - a partially ordered domain (typically a constraint semi-ring) V,
  - a set of variables with associated ranges, X,
  - a function mapping assignments over X to V, giving preferences to assignments with greater values in V.
- A soft constraint solver takes an SCP and returns a set of assignments of maximal value.
- A soft agent SCP consists of
  - a partially ordered valuation domain V,
  - a set of actions, acts,
  - a function, val, that given a local context assigns a preference, a value in V, to each action.

A Simple SCP solver: valuation domain

- The solver is parameterized by a valuation module that must implement the soft-agent valuation theory VALUATION

```
fth VALUATION is
  pr BOOL .   inc SOFT-AGENTS .
  sort Grade .  — a partially ordered valuation domain
  op equivZero : Grade -> Bool .
  op _<_ : Grade  Grade -> Bool .
  op val : Id KB Action  -> Grade . — an agents preference for an action
endfth
```

A simple SCP solver: parameterization

```
fmod SOLVE-SCP{Z :: VALUATION} is
 inc SOFT-AGENTS .  — declares solveSCP
 sorts RankEle{Z} RankSet{Z} .  — parameterized sorts
 op {_,_} : Z$Grade ActSet -> RankEle{Z} .
             —Z$Grade is the sort mapped to Grade in the parameter module
vars rks rks1 : RankSet .
eq solveSCP(id,kb,acts) = solveSCP$(id,kb,acts,none) .
 eq solveSCP$(id,kb,none,rks) = getAct(rks) . -- union of the top act sets
 ceq solveSCP$(id,kb,act actset,rks) = solveSCP$(id,kb,actset,rks1)
  if v0 := val(id,kb,act)   — val from the parameter module
  /\ rks1 := updateRks(rks,act,v0) .
…  endfm
```

Updating the rank set

(filling in some …s)

```
ceq updateRks({v0,acts0} rks,act1,v1) = {v0,acts0} rks
     if v1 < v0 .
ceq updateRks({v0,acts0} rks,act1,v1) = updateRks(rks,act1,v1)
     if v0 < v1 .
eq updateRks({v1,acts0} rks,act1,v1) = {v1,acts0 act1} rks .
eq updateRks(rks,act1,v1) =
       if equivZero(v1) then rks else {v1,act1} rks fi
     [owise] .
```

Part III

Soft-agent Rules

doTask
timeStep

System configuration: sorts and construction

- ASystem — top level system configuration { c:Conf } :: System
- Agent Env < ConfEle < Conf — multiset of configuration elements
- Elements of sort Agent have the form
  - [id : class | lkb: localkb, ckb: cachedkb, evs: events, ...]
    - id :: Id, class :: Class — the agents unique identifier and class
    - localkb :: KB -- the agents local knowledge
    - cachedkb :: KB -- knowledge to be opportunistically shared
    - events :: EventSet—pending actions, tasks, shared knowledge
    - … — other attributes, model specific
- Elements of Env have the form [eid | ekb]
  - ekb :: KB -- represents the physical environment include each agents physical state

doTask rule

Involves a single agent with a 0 delay task, and its local environment

```
crl[doTask]:
  [id : cl | lkb : lkb,  evs : (task @ 0) evs,  ckb : ckb, atts]   [eid | ekb ]
=>
  [id : cl | lkb : lkb', evs : evs',  ckb : ckb', atts] [eid | ekb ]
if t := getTime(lkb)
/\ {ievs,devs} := splitEvents(evs,none) — immediate and delayed events
/\ {lkb', evs', kb} kekset := doTask(cl, id, task,ievs, devs, ekb, lkb)
/\ ckb' := addK(ckb,kb)
 [print "doTask: " id " ! " task  " time: " t " !! " evs' " \n " kekset ] .
```

timeStep rule: auxiliaries

mte(c:Conf) — how much time can elapse before an event is ready

- 0 — if any tasks have 0 delay
- inf — if there is nothing todo
- nz — the least non-zero delay of a task otherwise

doEnvAct executes actions concurrently one time unit at a time.

```
ceq doEnvAct(t, nzt, ekb, evs)
  = doEnvAct(s(t), nzt monus 1, ekb', timeEffect(evs,1))
                  — inc time and repeat with new state ekb’
  if ekb' := doUnitEnvAct(t, ekb, evs)  — one time unit
  /\ okEnv(ekb')  .  — consistency check
 eq doEnvAct(t, 0, ekb, evs) = ekb .  — done
```

doUnitEnvAct returns ekb updated by doing available actions for 1 time unit duration

timeStep rule

The timeStep rule simultaneously performs actions using doEnvAct.

```
crl[timeStep]:
  { aconf } => { aconf2 }
if nzt := mte(aconf) — the amount of time to step
/\ t := getTime(envKB(aconf))  — the current time
/\ ekb' := doEnvAct(t, nzt, envKB(aconf), effActs(aconf))
             — the result of executing ready actions for unto nzt
/\ aconf0 := updateEnv(ekb',timeEffect(aconf,nzt))
/\ aconf1 := shareKnowledge(aconf0) — delivers posted knowledge
/\ aconf2 := updateConf(aconf1)
  — model specific hook for monitoring, accumulating metadata, …
 [print "eAct: " ekb' "\ntimeStep: " t " ++ " nzt] .
```

Specifying Properties

Specifying properties

- We consider two kinds of properties
  - Critical configurations — that should not be reached
  - Goal configurations — that should be reached
- Auxiliary ConfElts
  - bound(n:Nat) — to specify bounded search or model-checking
  - criticalConf goalConf — configuration markers
- Predicates:
  - critical(c:Conf)
  - goal(c:Conf)

Specifying properties

timeStep processing

```
  eq updateConf(bound(n) aconf) =
    (if critical(aconf) then criticalConf aconf   — mark it critical
    else (if goal(aconf bound(n)) then goalConf  — done
    else bound(monus(n)) aconf fi) fi) fi) .  — count down and continue
```

Trivial goal

```
  eq goal(aconf bound(0)) = true .
  eq goal(aconf) = false [owise] .
```

Patrol Bots

Patrol bot knowledge

- Bot, Station :: Class
- Location: pt(x:Nat,y:Nat) :: Loc
- Bot specific knowledge
    - caution(id:Id,e:FiniteFloat) @ t -- how much energy to reserve
    - myY(id:Id,n:Nat) @ t -- the horizontal track to patrol along
    - myDir(id:Id,dir:Dir) @ t-- the current direction (E or W)
- Partial Order -- fresh information subsumes stale
    - eq (myY(id,y0) @ t0) << (myY(id,y1) @ t1) = t0 < t1 .
    - eq (myDir(id,dir0) @ t0) << (myDir(id,dir1) @ t1) = t0 < t1 ..
    - eq (energy(id,e0) @ t0) << (energy(id,e1) @ t1) = t0 < t1 .
    - eq (caution(id,e0) @ t0) << (caution(id,e1) @ t1) = t0 < t1 .

Patrol bot parameters and actions

- Parameters
  - gridX gridY :: Nat — Grid dimensions
  - chargeUnit :: FiniteFloat — charge per unit time
  - maxCharge :: FiniteFloat — stop charging when full
  - costMv :: FiniteFloat —energy cost to move one step
- Actions
  - mv(id,dir) — move in direction dir, one square
  - charge(id) — charge for one time unit, only at the station

Patrol bot action model

(Assume actions are valid, execute for 1 time unit)

```
 eq doMv(pt(x0,y0),E) = (if (s x0 < gridX) then pt(s x0,y0) else pt(x0,y0) fi) .
 ...
ceq doAct(id,kb,charge(id)) = kb1
  if t := getTime(kb)
  /\ e := min(getEnergy(id,kb) + chargeUnit, maxCharge)
  /\ kb1 := addK(kb, clock(s t) (energy(id,e) @ s t)) .
ceq doAct(id,kb,mv(id,dir)) = kb1
  if t := getTime(kb)
  /\  e := getEnergy(id,kb) - costMv
  /\ loc := doMv(getLoc(id,kb),dir)
  /\ kb1 := addK(kb, clock(s t) (atloc(id,loc) @ s t) (energy(id,e) @ s t) ) .
```

Soft constraint problem for patrol bots

- Actions are evaluated from two perspectives
  - energy (safety)
  - patrolling (progress)
- the results combined.

- The energy valuation domain: TriVal = bot < mid < top
- The patrolling valuation domain: UVal = FiniteFloats between 0 and 1
- The combined domain: BUVal = TriVal x Uval

```
 eq {b1,u1} < {b2,u2} = (b1 < b2) or (b1 == b2 and u1 < u2) .
     — lexicographic combination of partial orders
  eq equivZero({b1,u1}) = (equivZero(b1)) .
     —zero energy preference overrides/vetoes
```

Valuation from energy perspective

```
val-energy(id,kb) =
  top -- if cost2station + caution < energy level
  mid -- if cost2station  < energy level
  bot -- owise


let v = val-energy(id,doAct(id,kb,mv(id,dir)))
  val-energy(id,kb,mv(id,dir)) =
    if v =/= mid or "if dir is towards the charging station"
    then v else bot
 val-energy(id,kb,charge(id) =
    if "at the charging station and not fully charged"
   then top
   else bot
```

Valuation from patrolling perspective

```
let y0 = the preferred Y-coordinate,
    y  = the actual value,
   dir = the current direction
eq val-patrol(id,kb, mv(id,dir1)) =
    if (y0 < y)
    then (if  (dir1 == S) then 0.9 else 0.0)
    else  (if (y < y0)
             then (if  (dir1 == N) then 0.9 else 0.0)
             else (if  (dir == dir1) then 0.9 else 0.0)

 eq val-patrol(id, kb, charge(id))  = 1.0 .  — happy to charge
```

Instantiating the solver

SOLVE-SCP is instantiated with VAL-PATROL-ENERGY-CONSERVATIVE

that defines

```
  eq val(id,kb) = {val-energy(id,kb),1.0} .
  eq val(id,kb,act) = {val-energy(id,kb,act),val-patrol(id,kb,act)} .
```

using the view

```
view  valuation2valpatrolenergyconservative from
    VALUATION to VAL-PATROL-ENERGY-CONSERVATIVE is
    sort Grade to BUVal .  endv
```

giving

```
  SOLVE-SCP{valuation2valpatrolenergyconservative}
```

Action physics

```
ceq doUnitEnvAct(t, ekb, (mv(id,dir) @ 0 ; nzt) evs)
        = doUnitEnvAct(t, ekb’, evs)
 if l0 := getLoc(id,ekb)
 /\ l1 := doMv(l0,dir)
 /\ e0 := max(getEnergy(id,ekb) - costMv,0.0)
 /\ l2 := windEffect(l1,ekb)
 /\ l3 := (if occupied(l1,ekb) then l0
            else (if occupied(l2,ekb) then l1 else l2 fi) fi)
 /\ ekb’ := addK(ekb,(atloc(id,l3) @ s(t)) energy(id,e0) @ s(t)) .

ceq windEffect(l0, (clock(t)) (wind(dir,n) @ t0) ekb) = doMv(l0,dir)
     if t rem n == 0 .
eq windEffect(l0,ekb) = l0 [owise] .
```

Execution and Analysis

Configurations

```
1asys(e,ws,wn) :: ASystem with bound=200
  1 bot at pt(0,0), going W along y=0, energy 15.0, caution e;  wind(S,ws), wind(N,wn)
1asys(1.0,5,7) =
{[el | clock(0) class(b(0), Bot) (atloc(b(0), pt(0,0)) @ 0)  energy(b(0), 1.5e+1) @ 0
     class(st(0), Station) (atloc(st(0), pt(2, 1)) @ 0) (wind(S, 5) @ 0) (wind(N,7) @ 0) ]
[b(0) : Bot |
  lkb : (clock(0) class(b(0), Bot) (atloc(b(0), pt(0, 0)) @ 0) caution(b(0),1.0) @ 0)
          (myY(b(0),0) @ 0) (myDir(b(0),W) @ 0) energy(b(0),15.0) @ 0)
         class(st(0), Station) (atloc(st(0), pt(2, 1)) @ 0),
  ckb : none,   evs : (tick @ 1)]  bound(200)}
2asys(e,ws,wn)  :: ASystem with
 1 bot at pt(0,0), going W along y=0, energy 15.0, caution e
 1 bot at pt(4,2), going E along y=2, energy 15.0, caution e
 wind(S,ws), wind(N,wn)
```

DEMO

Analysis: what is critical?

```
ceq critical([eid | (energy(id,ff) @ t) kb ] aconf) = true
   if equivZero(val(id,(energy(id,ff) @ t) kb)) .

eq critical(aconf) = false [owise] .
```

Analysis — single bot scenarios

- search [1] 1asys(1.0,0,0) =>+ {criticalConf aconf} .
  - No solution.
  - no wind, no problem
- search [1] 1asys(1.0,5,7) =>+ { criticalConf aconf } .
  - Solution 1 (state 2215)
  - clock(191) (atloc(b(0), pt(2,0)) @ 191)(energy(b(0), 1.0) @ 191)
  - frequent wind, low caution, bot in trouble
- search [1] 1asys(2.0,5,7) =>+ { criticalConf aconf } .
  - No solution.
  - more caution fixes the problem

Analysis — 2 bots

- search[1] 2asys(1.0,0,0) =>+ {criticalConf aconf} .
  - Solution 1 (state 86) clock(15)
  - (atloc(b(0), pt(2, 1)) @ 14)(energy(b(0), 7.0) @ 15)
  - (atloc(b(1), pt(2, 2)) @ 15) energy(b(1), 1.0) @ 15]
  - no wind, minimal caution, the bots conflict

- search[1] 2asys(4.0,0,0) =>+ {criticalConf aconf} .
  - No solution.
  - more caution keeps them working

Analysis — 2 bots with wind

- search[1] 2asys(6.0,5,7) =>+ {criticalConf aconf} .
  - Solution 1 (state 15023) clock(68)
  - (atloc(b(0), pt(2, 2)) @ 68) (energy(b(0), 1.0) @ 68)
  - (atloc(b(1), pt(2, 1)) @ 67) (energy(b(1), 2.0e+1) @ 68)
  - frequent wind, much caution does not help
- search[1] 2asys(6.0,7,11) =>+ {criticalConf aconf} .
  - Solution 1 (state 6047) clock(88)
  - (atloc(b(0), pt(2, 2)) @ 88) (energy(b(0), 1.0) @ 88)
  - (atloc(b(1), pt(2, 1)) @ 87) (energy(b(1), 2.0e+1) @ 88)
  - less frequent wind, still critical
- search[1] 2asys(4.0,11,17) =>+ {criticalConf aconf} .
  - No solution.
  - this seems to be the most wind 2 bots can handle

Analysis: what went wrong?

searchTag(asys:ASystem, 'criticalConf.Conf) uses meta-level search to find a path to a critical state, and summarize the changing information along the path as abstract transitions

botstate -> envstate

the last two elements of searchTag(1asys(1.0, 5, 7), 'criticalConf.Conf) :

```
  [b(0) : Bot |  lkb : ((atloc(b(0), pt(2, 0)) @ 189) energy(b(0), 3.0) @ 189),
                  evs : (mv(b(0), N) @ 0 ; 1)]      ->
  [el | clock(190) (atloc(b(0), pt(2, 2)) @ 190) energy(b(0), 2.0) @ 190]) ;


  [b(0) : Bot |  lkb : ((atloc(b(0), pt(2, 2)) @ 190) energy(b(0), 2.0) @ 190),
                 evs : (mv(b(0), S) @ 0 ; 1)] ->
  [el | clock(191) (atloc(b(0), pt(2, 0)) @ 191) energy(b(0), 1.0) @ 191]
```

b(0) attempts to move N to (2,1), is blown by wind to (2,2)

Analysis: what went wrong?

searchTag(2asys(1.0, 0, 0), 'criticalConf.Conf) .

- Both bots are 1 square from the station, each with energy 3.
- Both attempt to enter the station, b0 wins.
- b1 keeps trying because its knowledge is out-of-date, and runs out of energy

searchTag(2asys(4.0, 7, 11), 'criticalConf.Conf) .

- b1 is one move from the station with energy 2, and the wind blows him too far.

searchTag(2asys(6.0, 7, 11), 'criticalConf.Conf) .

- b0 waiting for b1 to leave station, b1 leaves, but wrong direction,
- and returns, b0 thinks b1 is gone and uses last bit of energy for naught.

Surveillance Drones

Surveillance drones — setup

The scenario :

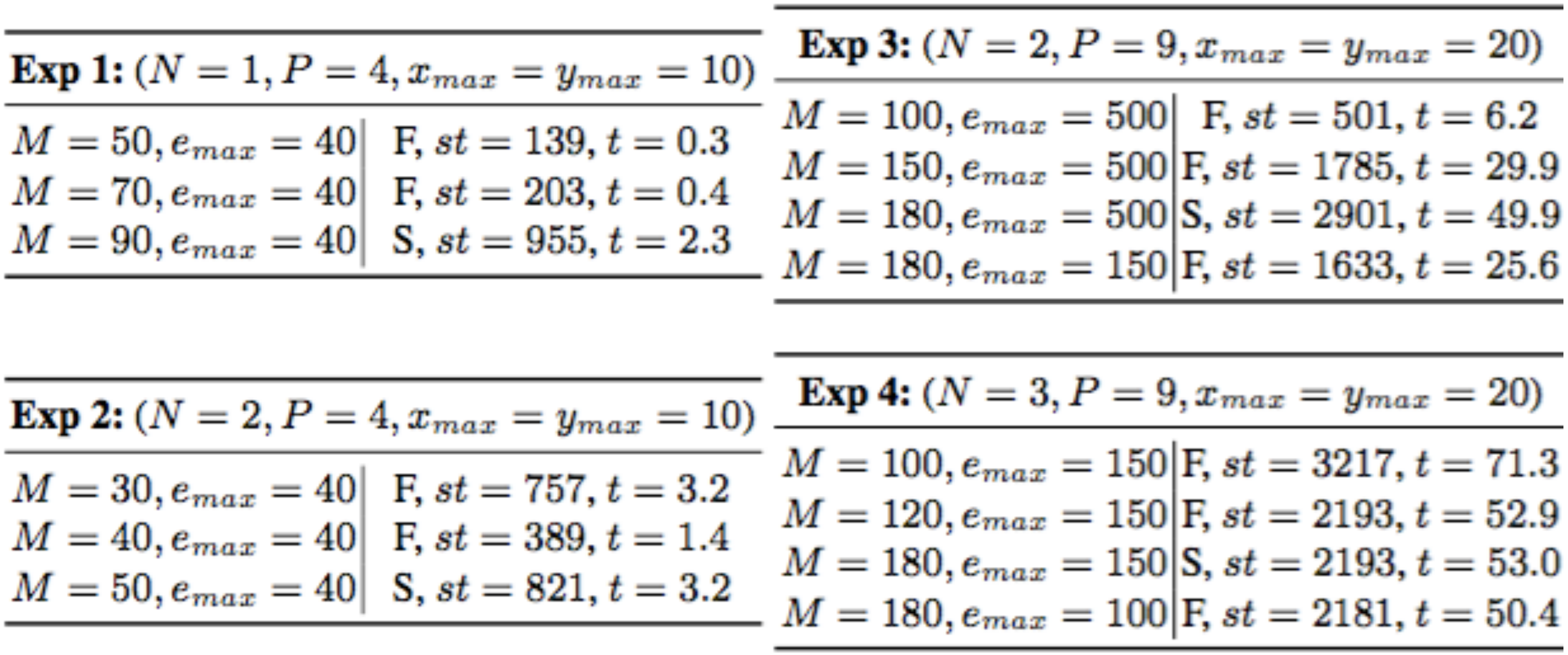
- 3-D grid with *xmax = ymax (10 or 20)*
- N (1-3) drones that can move in a 3 D grid, take photos, or recharge
- P (4 or 9) points of interest
- A base/charging station for at the center.
- Drones have maximum energy of emax.
- There should be recent pictures (<= M time units old) at each point
- Drones rank their actions based on the drone's position, energy, and photo knowledge (which photos are going stale).

We carried out bounded search/model-checking with bound 4 x M, varying emax and M where

```
critical(kb,t) =  if t < M  then false    —  No critical state in the beginning
    else picOfAll(cs,objLocs)             — does every point have a photo?
```

where cs is the set of photo info {id,res,t0} from kb such that t - t0 < M

Surveillance drones analysis summary



Exp 1: $(N = 1, P = 4, x_{max} = y_{max} = 10)$	
$M = 50, e_{max} = 40$	F, $st = 139, t = 0.3$
$M = 70, e_{max} = 40$	F, $st = 203, t = 0.4$
$M = 90, e_{max} = 40$	S, $st = 955, t = 2.3$

Exp 2: $(N = 2, P = 4, x_{max} = y_{max} = 10)$	
$M = 30, e_{max} = 40$	F, $st = 757, t = 3.2$
$M = 40, e_{max} = 40$	F, $st = 389, t = 1.4$
$M = 50, e_{max} = 40$	S, $st = 821, t = 3.2$

Exp 3: $(N = 2, P = 9, x_{max} = y_{max} = 20)$	
$M = 100, e_{max} = 500$	F, $st = 501, t = 6.2$
$M = 150, e_{max} = 500$	F, $st = 1785, t = 29.9$
$M = 180, e_{max} = 500$	S, $st = 2901, t = 49.9$
$M = 180, e_{max} = 150$	F, $st = 1633, t = 25.6$

Exp 4: $(N = 3, P = 9, x_{max} = y_{max} = 20)$	
$M = 100, e_{max} = 150$	F, $st = 3217, t = 71.3$
$M = 120, e_{max} = 150$	F, $st = 2193, t = 52.9$
$M = 180, e_{max} = 150$	S, $st = 2193, t = 53.0$
$M = 180, e_{max} = 100$	F, $st = 2181, t = 50.4$

Table 1: N is the number of drones, P the number of points of interest, $x_{max} \times y_{max}$ the size of the grid, M the time limit for photos, and e_{max} the maximum energy capacity of each drone. We measured st and t, which are, respectively, the number of states and time in seconds until finding a counter example if F (fail), and until searching all traces with exactly $4 \times M$ ticks if S (success).

Future directions

- Better sensors -- object/obstacle detection
- Automating diagnostics
- Distributed diagnostics
- Simulator in the loop
- Soft Constraint Automata Models
  - Compositionality
  - Abstracting Maude models
- Language for behavior/properties — getting systematic
- Probabilistic environment models and execution
- Scaling — “swarms”

Acknowledgements

Soft-agents team*

- Farhad Arbab, CWI/LIAC
- Tobias Kappe, CWI/LIAC
- Vivek Nigam, U. Pariba
- Carolyn Talcott
- Maneesh Yadav

*Alphabetical order

Funding

- NSF grant CNS-1318848
  - Extensible Symbolic Analysis Modulo SMT: Combining the Powers of Rewriting, Narrowing, and SMT Solving in Maude
- ONR grant N00014-15-1-2202
  - Detection and diagnosis of deviations in distributed systems of autonomous agents
  - PhD position available at CWI/LIAC

Thats all folks!