Detection and Diagnosis of Deviations in Distributed Systems of Autonomous Agents

Abstract

1 Overview

We are interested in design and operation of (locally) autonomous cyber-physical agents (CPAs), for example mobile robots (bots) or drones. Such agents have simple goals to achieve and choose actions to achieve their goals based on what they can sense locally, information shared by other agents, priorities and a model of the effects of actions. Agents need to be robust/resilient to changes in their environment (obstacles,) and to faults or noise/variance in sensors and actuators. Agents need to detect detecting when something is wrong, and somehow adapt.

To study this problem we developed the soft agent framework for modeling and analyzing properties of possible behaviors of such CPAs. In the framework, agent knowledge (cyber) is modeled separately from the physical state of the agent and its local environment. In addition, there are mechanisms to specify perturbations such as obstacles or faults. Models are formal executable specifications based on rewriting logic and written in the Maude language. Using the builtin functionality of Maude, a model can be executed in the context of different perturbations using different execution strategies. Search can be used to determine if a state with a given property, good or bad, can be reached. The execution rules can be instrumented to collect logs selecting information of interest. Thus we can explore design space tradeoffs and agent robustness in context of a range of goals and faults.

In order to detect that something has gone wrong a specification of acceptable behavior/progress is needed. To be useful for diagnostic purposes, the specification should determine key events and constraints while not being overly prescriptive leading to 'false' violations, i.e. violations that don't really matter for purpose of the overall system goals. To this end, we propose a notion of protocol consisting of a partially ordered set of event patterns, together with constraints on the pattern variables and invariant constraints. Event logs collect execution events relevant to a protocol from two perspectives, agent and environment. SMT constraint solving is used to determine if a log satisfies a protocol, and if not, maximal satisfiable subprotocols are returned that provide clues as to when something went wrong. Once a candidate causal event is hypothesized, a gedanken experiment is done to check the hypothesis. Namely, the execution is rerun from the state preceding this event, blocking this event, to see if the detected deviation no longer occurs.

In summary, we report here the following contributions:

- Architecture / framework for specifying and analysing CPAs
- A family of fault models and mechanism for composing fault models with the execution environment

- A definition of Soft Agetn Protocol and protocol satisfaction
- Detection and diagnosis methods
  - Expectation failures (Local execution failures)
  - Protocol deviations
  - Gedanken experiments

The technical development has been tested using three case studies that will be used as examples in the text.

- PatrolBots. The agents in this case study patrol along assigned lanes from side to side of a two dimensional grid with a charging station in the center. They are independent, but may compete for access to the charging station. Questions to study include, for a given grid size, how much energy reserve is sufficient to avoid running out of energy under different perturbation conditions. Since these agents are independent, protocols should be satisfied independently and one agent can only affect the diagnosis of another in the role of an obstacle.
- BotTeam. This case study illustrates cooperation/coordination of two bots carrying out maintenance at a given list of locations, using a two phase process where the second bot must complete the second phase within a given time of completion of the first phase by the first bot. This allows us to study questions of how robust the agent coordination is to environmental perturbations. Protocols for the BotTeam have constraints the involve both bots, adding challenges to detection and diagnosis.
- Drones. The drone agents fly over a specified area visiting designated locations. This was initiated as part of a collaboration with a group building small drones with the eventual objective of surveillance of large agricultural areas, such as sugar cane plantations in Brazil. The drone model introduces a third space dimension, and the need to account for takeoff and landing as well as flying from location to location (and crashing if energy is too low). A drone grid has many more points than a PatrolBot or BotTeam grid, to enable finer grain modeling of distance.

The remainder of this document is organized as follows. Section 2 describes the structures, functions and rules provided by the soft agent framework and the case study specific functions the must be defined for each case study. It also defines a notion of execution trace for SA systems. Section 3 describeds the specification and application of fault models. Section 4 defines our notion of protocol and protocol satisfaction by an execution trace. In Section 5 we describe the design of the BotTeam case study and one of the scenario protocols. In Section 6 we discuss how to systematically find deviations: specific faulty actions or sensor readings, and protocol elements that are not satisfied by an execution. Section 7 presents Maude implementations of algorithms that automate some of the methods discussed in section 6. Section 8 summarizes experiments carried out by running the BotTeam scenario in a range of fault models. Section 9 outlines several directions for future work.

The Maude implementation of the SA Framework, case studies, sample output and documentation is availble at `https://github.com/SRI-σCSL/SoftAgentsDiagnosis.git`.

2 The Soft Agent Framework (SAF)

The soft agent (SA) framework provides an architecture and basic functions for defining executable models of SA systems. The framework is formalized in the Maude rewriting logic language and supports exploration of designs (agent strategies) and evaluation in environments with different uncertainties and forms of interference using generic fault models. Key features include

- explicit representation of both the cyber (decision making) and physical (using sensors and actuators) aspects of a SA cyber-physical system
- use of partially ordered knowledge items to representation state, both agent and environment
- communication as knowledge sharing

A system (configuration) consists of one or more agents and one environment object. Each agent has a local knowledge base. This knowledge base can include results of observations (reading sensors), goals, priorities, and information shared by other agents. It is the knowledge an agent uses to make decisions. The environment object also has a knowledge base. This knowledge base is intended to model what holds in the physical world, including agents physical state, the surrounding environment (which may include resources, obstacles, ...), and models for physical actions (including fault models, basic uncertainty parameters, and so on).

Two rewrite rules define the execution/operational semantics of an SA system model, invoking interface functions that need to be defined by each model. The `doTask` rule defines the agents decision making process. The `timeStep` rule executes actions proposed by the decision processes, carries out information sharing, and advances time. The `timeStep` rule has hooks to be used for model specific instrumentation of the execution, for example recording a log or adding metadata needed to compute properties of interest.

Once model specific interface functions have been defined, initial configurations can be explored by using the rewrite command to see one possible execution, using strategy controlled rewriting to explore executions of particular interest, or using search to carry out various forms of reachability analysis.

It is also possible to use an augmented version of the framework to carry out statistical modeling checking in the case that agents or the environment have probabilistic behavior [6]. The basic SA framework is described in more detail in [11].

2.1 Introduction to Rewriting Logic and Maude

Rewriting logic [8] is a logical formalism that is based on two simple ideas: states of a system are represented as elements of an algebraic data type, specified in an equational

theory, and the behavior of a system is given by local transitions between states described by *rewrite rules*. An equational theory specifies data types by declaring constants and constructor operations that build complex structured data from simpler parts. Functions on the specified data types are defined by *equations* that allow one to compute the result of applying the function. A *term* is a variable, a constant, or application of a constructor or function symbol to a list of terms. A specific data element is represented by a term containing no variables. Assuming the equations fully define the function symbols, each data element has a canonical representation as a term containing only constants and constructors.

A rewrite rule has the form $t \Rightarrow t'$ *if* c where t and t' are terms possibly containing variables and c is a condition (a boolean term). Such a rule applies to a system in state s if t can be matched to a part of s by supplying the right values for the variables, and if the condition c holds when supplied with those values. In this case the rule can be applied by replacing the part of s matching t by t' using the matching values for the place holders in t'. The process of application of rewrite rules generates computations (also thought of as deductions).

Maude is a language and tool based on rewriting logic [2, 7]. Maude provides a high performance rewriting engine featuring matching modulo associativity, commutativity, and identity axioms; and search and model-checking capabilities. In Maude data types are called *sorts* and Maude implements order-sorted membership equational logic [1]. The Maude declaration `subsort NZNat < Nat` means that `NZNat` (the non-zero natural numbers) is a subset of `Nat` (the natural numbers). In general we will use `this font` for Maude concepts. Rather than formal operator declarations we will introduce constructors and functions by patterns such as the function that returns the minimum of two numbers `min(i:Nat,j:Nat)`. Following Maude's syntax for inline variable declaration `i:Nat` is a variable of sort `Nat`. Maude specifications are organized in modules that have a hierarchical structure (one module can import others). In the following we suppress that information as it isn't needed to understand the main ideas.

2.2 Basic sorts and functions

The key basic sorts of SAF are: `Id`, `Time`, `Class`, `KB`, `KItem`, `Sensor`, `Task`, `Action`, and `Event`.

`Id` is the sort of identifiers, used to identify agents, and other objects. Each model is responsible for defining its own `Id` constructors. `Time` is the sort used to represent time. It can be discrete (natural numbers) or dense (rationals or reals). In the current SAF we model time using the sort `Nat` of natural numbers. `Class` is the sort of object classes. In the Patrol Bot example there are three classes: `Bot`, the patrolling agents; `Station`, the charging station; and `Obstacle`, entities that occupy locations, preventing bots from passing through; and corresponding `Id` constructors `b(i:Nat)` (bots), `st(i:Nat)` (stations), and `ob(i:Nat)` (obstacles). A constant `eI` of sort `Id` is declared to name the unique environment object in a system configuration.

A knowledge base (sort `KB`) is a set of knowledge items (sort `KItem`). There are two (sub)sorts of knowledge items, persistent (sort `PKItem`) and transient (sort `TKItem`). Transient knowledge items are time stamped information items of the form `info:Info @ t:Time` where terms `info:Info` are expected to have parameters that change over

time. Persistent knowledge items model properties that do not vary over time and thus are not time stamped. There is a distinguished knowledge item `clock(t:Time)` that represents the current time. For example `clock(10)` represents that `10` time units have passed. There two additional builtin knowledge item terms. `class(id:Id,cl:Class)` says that the class of the entity (agent or other object) with identifier `id:Id` is `cl:Class`. It has sort `PKItem` as the class is not expected to change. For example `class(b(0),Bot)` says that the entity with identifier `b(0)` is a `Bot`. `atloc(id:Id,l:Loc) @ t:Time` says that the entity with identifier `id:Id` is at location `l:Loc` at time `t:Time`. We represent location knowledge as a TKItem since in the case of a mobile entity `l:Loc` changes as the agent moves. The sort `Loc` is an interface sort at the framework level. There is a constant `noLoc` indicating an unknown location. In the Patrol Bot example locations are points on a two dimensional grid, represented by terms `pt(x:Nat,y:Nat)` of sort `Pt2` a subsort of `Loc`. The boundaries of the grid or region accessible to the Patrol Bots is represented by `fence(ll:Loc,ur:Loc) @ t:Time` where `ll:Loc,ur:Loc` are the lower left, upper right points of the grid. Similar fence knowledge is used to constrain mobile agents in the BotTeam and Drone case studies. Energy is another key concern of mobile agents. `energy(id:Id,e:Nat) @ t:Time` represents the percent of the energy capacity of the agent with identifier `id:Id` that remains at time `t:Time` is `e:Nat`. The PKItem `comDist(n:Nat)` is part of the environment KB. It determines how close two agents need to be to share information. Finally Patrol Bots have information items `myDir(id:Id,pt:Pt2)`, that specifies the direction, the agent is currently moving, thinking of `pt:Pt2` as a direction vector, and `myY(id:Id,n:Nat)` that specifies the preferred value of the Y coordinate.

Elements of the sort `Sensor` name sensors available to an agent. In the PatrolBot example the Bots have location, energy, and obstacle sensors, associated to the constants `locS`, `energyS`, and `obstacleS` respectively.

Events are used to determine what happens during rule application. The sort `Event` has two subsorts: `DEvents` that are actions (sort `Action`) or tasks (sort `Task`) with delays, and `IEvents` that are events to be processed immediately (implicit delay of 0). The sort `Task` represents tasks to be carried out by agents. Task events are executed by the `doTask` rule. In our case studies only one task is used, namely `tick`. It is used to schedule the next time the agent reads its sensors and proposes actions.

The sort `Action` is used to describe actions proposed by agents and carried out by the environment (the physics), during application of the `timeStep` rule. Each action has an identifier parameter that identifies the agent carrying out the action, and possibly other parameters. PatrolBots have two actions: `mv(id:Id,pt:Pt2)` describing a move of one distance unit in direction `pt:Pt2`; `charge(id:Id)` describing recharging the energy level, if the agent is located at a charging station.

The function `myActs(cl:Class,id:Id,kb:KB)` returns the set of actions that are feasible for an agent with class `cl:Class`, and identifier `id:Id` in the context of the agents local knowledge base, `kb:KB`.

The function `doAct(act:Action,kb:KB)` models the effects of the action `act:Action` in the situation represented by `kb:KB`. It returns knowledge items with new values for information that changed. The argument `kb:KB` can be either an agents local KB, thus representing the agents model of the effect of an action, or the environment KB, thus rep-

resenting the *actual* effect of an action.

2.3 SA system semantics: configuration and rules

A SA configuration (sort `Conf`) is a multiset of configuration elements (sort `ConfElt`). Agent objects (sort `Agent`) and environment objects (sort `Env`) are configuration elements. Formally `Agent` and `Env` are subsorts of `ConfElt`. A well-formed configuration needs a unique environment object and at least one agent object. Other configuration elements can be defined to control execution, and to track properties of interest. Logs and bounds on the time or number of applications of selected rewrite rules are examples of other configuration elements as we will see later. Configurations are *open*, you can add or remove and element and still have a configuration. A SA system `{ conf:Conf }` (sort `ASystem`) specifies exactly which configuration elements are present by enclosing them in `{}`s.

Agent objects. An agent object has the form

```
[id:Id : cl:Class | attrs:AttributeSet ]
```

An attribute is a key-value pair. Agent objects must have at least the following attributes:

- `lkb : lkb:KB` the local knowledge base
- `sensors : ss:SensorSet` the agent's sensors
- `evs : evs:Events` events awaiting processing
- `ckb : ckb:KB` the cache knowledge base, knowledge to be shared

Environment objects. An environment object has the form

```
[eid:Id  | ekb:KB ]
```

The environment KB (`ekb:KB`) contains information about the physical state of each agent such as location and energy, information about resources such as location of charging stations, information about obstacle, global parameters such as communication range, and fault models.

The semantics of a SA configuration is defined by two rewrite rules: `doTask` and `timestep`.

2.3.1 Rule doTask

The `doTask` applies to configurations, it has no need to know all the configuration elements. This rule applies to any agent with a task that has 0 delay (`(task @ 0)`). The term before the =>, the premiss is matched to a configuration by finding a substitution for the variables that makes the term a sub-configuration. This sub-configuration is replaced by the term after the => instantiated by the matching substitution extending with the bindings following the `if`.

```
crl[doTask]:
[id : cl | lkb : lkb, evs : ((task @ 0) evs), ckb : ckb,
           sensors : sset, ats] [eid | ekb ]
=>
[id : cl | lkb : lkb', evs : evs',  ckb : ckb', sensors : sset, ats]
[eid | ekb' ]
if t := getTime(lkb)
/\ {ievs,devs} := splitEvents(evs,none)
/\ {skb,ekb'} := readSensors(id,sset,ekb)
/\ kekset := doTask(cl, id, task,ievs, devs, skb, lkb)
/\ {lkb', evs', kb} kekset0 := selectKeK(lkb,kekset)
/\ ckb' := addK(ckb,kb) .
```

When the `doTask` rule is applied the agent reads its sensors (`readSensors(id,sset,ekb)`) and proposes a set of triples of the form `{lkb', evs', kb}` (sort `KBEventsKB`, the `kekset` binding) where `lkb'` is the agent's new local KB, `evs'` updates the pending events, and `kb` is new information to share. As the name suggests, the function `splitEvents` splits its first argument into two sets, the immediate events `ievs` and the events with delays `devs`. The agent processes `ievs`. It may modify `devs`, but usually just adds new tasks and/or actions. The function `readSensors` returns the union `skb` of the result of reading each sensor in `sset` independently, together with an updated environment. To read a single sensor, the sensor knowledge is retrieved from the environment KB along with any fault models for that sensor. If there are no fault models, the sensor knowledge is returned. If there is a fault model, it is applied using `applySensorF(id,s:Sensor,skb0,fkb,ekb)` where `skb0` is the sensor knowledge and `fkb` is the fault model. The environment update is an artifact of how fault models are applied using Maude's random number generator. Fault models and their application are discussed in section 3). As an example the knowledge associated to the location sensor `locS` of a Bot with identifier `b(0)` has the form `atloc(b(0),loc:Loc) @ t:Time`. The function `selectKeK(kb:KB,kekset: KBEventsKBSet` is an interface function to select from the options returned by `doTask`. These are the options Maude has for rewriting and search. The framework provides two implementations: `allKeK(kb:KB,kekset: KBEventsKBSet)` selects all options; and `bestKeK(kb:KB,kekset: KBEventsKBSet)` selects the best options, i.e. those with maximally ranked actions. A model can use one of these or define another selection function.

The SA framework provides a compositional structure for the function `doTask` illustrated by the Maude code from the PatrolBot model.

```
ceq doTask(cl,id,tick,ievs,devs,skb,lkb) =
  if racts == none
  then {lkb2, devs (tick @ botDelay), none }
  else selector(doTask$(id,getThresh(id,lkb2),lkb2,
                        devs (tick @ botDelay),racts))
   fi
 if lkb0 := handleS(cl,id,lkb,ievs)
 /\ lkb2 := proSensors(id,lkb0,skb)
 /\ racts := actSCP(cl,id,lkb2)  .
```

Each model provides equations defining the component functions `handleS`, `proSensors`, and `actSCP`. `handleS` processes `ievs` including information shared by other agents,

`rcv(kb)`. The default is to add this new information to the local KB. The `proSensors` function uses new sensor information to make high-level decisions, and update local parameters, often represented by states of an automaton. For example a Patrol Bot will notice when it arrives at the edge of the grid and reverse `myDir`.

The function `actSCP` has the following structure

```
ceq actSCP(cl,id,kb) = racts
  if acts0 := myActs(cl,id,kb)
  /\ racts := solveSCP(id,kb,acts0) .
```

As discussed above, `myActs` returns actions allowed for the agent given its local knowledge base `kb`. `solveSCP` implements a soft constraint solver and returns a ranked set of actions `racts` with elements of the form `{rval:RVal,act:Action}`.

The SAF provides the basic infrastructure for soft constraints including the module `RVAL` that declares the sort `RVal` and provides minimal structure including a zero element and a partial order relation and the module `SOLVE-SCP` that implements a simple solver, parameterized by a model specific definition of a valuation function `val(id:Id,kb:KB,act:Action)` that returns a value in `RVal`. The solver uses an interface function `updateRks(rks:RActSet,act:Actio` to accumulate ranked actions to return. The framework provides two implementations of this function: `updateRksMx(rks:RActSet,act:Action,v:RVal)` returns only maximally ranked actions; `updateRksAll(rks:RActSet,act:Action,v:RVal)` returns all non-zero ranked actions. A model is free to use one of these or define its own version. Each model defines its own subsort of `RVal`, typically as a lexicographic order of valuations for different concerns. For the Patrol Bots, the concerns are maintaining energy above the reserve level, stored in the local knowledge item `ereserve(id:Id,e:Nat) @ t:Time)`, keeping the Y component of its location close to the preferred Y value (`myY`), and progressing to the opposite edge of the grid. The energy reserve is intended to be sufficient for the an agent to be able to get to a charging station even though there may be obstacles or some sensor or actuator failures.

The soft constraint aspect of SA is discussed in more detail in [10, 9] and the use of semi-ring based preferences in Soft Constraint/Component Automata is discussed in [4, 5, 3].

2.3.2 Rule timeStep

The `timeStep` rule only applies when the function `mte` returns a non-zero time. This is the amount of time that must pass before an agent task is ready (0 delay). Thus `timeStep` rule applies only if the `doTask` rule can not be applied. Since the rule needs to know all configuration elements it applies to systems, `{ aconf }`.

```
crl[timeStep]:
{ aconf }
=>
{ aconf2 }
if nzt := mte(aconf)
/\ t := getTime(envKB(aconf))
/\ evs := effActs(aconf)
```

```
/\ ekb0 := doEnvAct(t, nzt, envKB(aconf), evs)
/\ ekb' := resolveKB(getEnvId(aconf), ekb0, envKB(aconf))
/\ aconf0 := updateEnv(ekb',timeEffect(aconf,nzt))
/\ aconf1 := shareKnowledge(aconf0)
/\ aconf3 := updateLog(aconf1, t,nzt,evs)
/\ aconf2 := updateConf(aconf3)
.
```

`effActs(aconf)` is the set of actions with zero time delay in some agents `evs` attribute. Actions in `evs` are carried out 'concurrently' by `doEnvAct(t, nzt, envKB(aconf), evs)` (explained below), producing updates for the environment knowledge base `ekb0`. The effects of the actions may be modified by various faults according to the fault models present in the environment knowledge base (see section 3). Once the results of the concurrent actions are collected, conflicts in the resulting knowledge base updates are resolved, `resolveKB(getEnvId(aconf), ekb0, envKB(aconf))`, to produce a consistent global update. In the current models, the only conflicts are when two ore more agents want to move to the same position. This is resolved by choosing one of the agents to succeed, the others must stay where they started.

`timeEffect(aconf,nzt)` advances time in the configuration `aconf` by `nzt`, and the environment KB in the resulting configuration is updated using `ekb'`. `shareKnowledge` considers the `ckb` attributes for each pair of agents that are within the model's communication distance (stored in `comDist`), computes what is new from each agents perspective and adds `rcv(newkb)` to the agents `evs` attribute.

The final two functions, `updateLog` and `updateConf`, are hooks to allow for instrumenting configurations. Execution logs are a common form of instrumenting executions. The SAF provides a data structure and parametric procedure to collect log information. A log (sort `Log` is a sequence of log items with elements of the sequence separated by `;`. In our case studies where the time to pass `nzt` is 1, `updateLog(aconf1, t,1,evs)` returns `aconf1` if `aconf1` has no log element. Otherwise it adds to the log a logitem `{t:Time,acts,lconf:Conf}` where `acts` is the actions with delay 0 in `evs` and `lconf` is computed by the interface function `kblog(aconf1,none)` (the `none` is the initial value of the configuration element accumulator).

The default for `updateConf` is to return its argument unchanged. The SAF provides a configuration element `bound(n:Nat)` to limit the number of `timeStep` rule applications. If this configuration element is present then `updatConf` acts as follows

```
eq updateConf(bound(0) aconf) = stopped(aconf) .
eq updateConf(bound(s n) aconf) = bound(n) updateConf(aconf) .
```

There are no rewrite rules for `stopped(aconf)`, thus rewriting terminates.

doEnvAct. In the case studies under consideration, the time to advance, `nzt` is `1`. In this case `doEnvAct(t, 1, ekb, evs)` computes `doEAct(t,a:Action,ekb)` for each action occurring in `evs` and returns the accumulated knowledge base updates. In the absence of faults, `doEAct` is just `doAct`. The effect if action faults is discussed in section 3).

2.4 Executions

Using the two rewrite rules, `doTask` and `timeStep`, executions of an SA system alternate steps where agents observe and decide on actions, and steps where the actions are realized by the environment. Thus an execution trace Tr of length $k+1$ is a sequence of transition segments of the form

```
C.0 -doTasks-> C.0a -acts.0-> C.1
...
C.k -doTasks-> C.ka -acts.k-> C.k+1
```

`C.i -doTasks-> C.ia` abbreviates

```
C.i = C.i.0 -doTask[id.0]-> C.i.1
          ...
C.i.j -doTask[id.j]-> C.i.j+1 = C.ia
```

where `id.0` ... `id.j` are the identifiers of agents with a pending task in `C.i` and `C.i.x -doTask[id.x]-> C.i.x+1` is an instance of the `doTask` rule for `id.x`. Note that `C.ja` differs from `C.j` only in agent attributes (and possibly associated random number counters in the environment if there are active fault models).

In `acts.j` transitions, `acts.j` is the set of actions with zero delay proposed by some agent, `evs := effActs(aconf)` in a single application of the `timeStep` rule. `C.j+1` differs from `C.j` in the environment KB and in the `evs` attributes of agents due to the knowledge sharing. Also, all clocks are incremented by the amount of time that should pass (`nzt` in the `timeStep` rule). In our current models, `nzt` is always 1 and clocks start at 0, thus the end time is $k+1$.

2.5 Rewriting and Search

We conclude the discussion of SAF by showing an example initial configuration for Patrol Bots, `initC`, and illustrating the use of rewriting and search.

```
Maude> red initC .
result Conf:
[eI |
  clock(0) comDist(4) randInc(3)
  class(b(0), Bot) class(b(1), Bot) class(st(0), Station)
  (atloc(b(0), pt(0, 1)) @ 0) (atloc(b(1), pt(6, 3)) @ 0)
  (atloc(st(0), pt(3, 2)) @ 0)
  (energy(b(0), 100) @ 0) (energy(b(1), 100) @ 0)
  (fence(b(0), pt(0, 0), pt(6, 4)) @ 0)
  (fence(b(1), pt(0, 0), pt(6, 4)) @ 0)
  (rand(eI, 2, 0) @ 0) (rand(b(0), 0, 0) @ 0)
  (rand(b(1), 1, 0) @ 0)]
[b(0) : Bot |
  lkb : (clock(0) class(b(0), Bot)
   class(st(0), Station) (atloc(st(0), pt(3, 2)) @ 0)
   (myY(b(0), 1) @ 0) (myDir(b(0), pt(-1, 0)) @ 0)
   (offTrack(b(0), 0) @ 0) (ereserve(b(0), 15) @ 0)
```

```
   (fence(b(0), pt(0, 0), pt(6, 4)) @ 0)) ,
  ckb : none,
  sensors : (locS obstacleS energyS),
  evs : (tick @ 0)]
[b(1) : Bot |
  lkb : (clock(0) class(b(1), Bot)
   class(st(0), Station) (atloc(st(0), pt(3, 2)) @ 0)
   (myY(b(1), 3) @ 0) (myDir(b(1), pt(1, 0)) @ 0)
   (offTrack(b(1), 0) @ 0) (ereserve(b(1), 15) @ 0)
   (fence(b(1), pt(0, 0), pt(6, 4)) @ 0)),
  ckb : none,
  sensors : (locS obstacleS energyS),
  evs : (tick @ 0)]
```

The configuration has an environment object `[eI | ... |]` and two bots `[b(0) : Bot | ...]` and `[b(1) : Bot | ...]`. The bots are initially at `pt(0,1)` and `pt(6,3)` with 100% energy level. `(myDir(b(0), pt(-1, 0)) @ 0)` says `b(0)` is going left/west. Since he is at the west boundary (X coordinate is 0, fence west boundary is $X = 0$) he will change direction and start moving east. Dually for `b(1)`

The system configuration below is the result of 30 rewrite steps, which is 10 rounds with two applications of `doTask` (one for each bot) and one application of `timeStep`. Thus the time advances by 10.

```
Maude> rew [30] {initC} .
result ASystem: {
[eI |
  clock(10) (atloc(b(0), pt(2, 1)) @ 10) (atloc(b(1), pt(4, 3)) @ 10)
  (energy(b(0), 50) @ 10) (energy(b(1), 50) @ 10) ...]
[b(0) : Bot |
  lkb : (clock(10) (lastAct(mv(b(0), pt(-1, 0))) @ 9)
   (atloc(b(0), pt(3, 1)) @ 9) (atloc(b(1), pt(2, 3)) @ 8)
   (myDir(b(0), pt(-1, 0)) @ 9) (offTrack(b(0), 0) @ 9)
   (energy(b(0), 55) @ 9)),
  ckb : (class(b(0), Bot) class(b(1), Bot)
   (atloc(b(0), pt(3, 1)) @ 9) (atloc(b(1), pt(3, 3)) @ 9) ...),
  sensors : (locS obstacleS energyS),
  evs : (rcv(atloc(b(1), pt(3, 3)) @ 9) tick @ 0)]
[b(1) : Bot |
  lkb : (clock(10)  (lastAct(mv(b(1), pt(1, 0))) @ 9)
    (atloc(b(0), pt(4, 1)) @ 8) (atloc(b(1), pt(3, 3)) @ 9)
    (myDir(b(1), pt(1, 0)) @ 9) (offTrack(b(1), 0) @ 9)
     (energy(b(1), 55) @ 9) ... ),
  ckb : (class(b(0), Bot) class(b(1), Bot)
        (atloc(b(0), pt(3, 1)) @ 9) (atloc(b(1), pt(3, 3)) @ 9)),
  sensors : (locS obstacleS energyS),
  evs : (rcv(atloc(b(0), pt(3, 1)) @ 9) tick @ 0)]}
rew [30] {initC} .
```

Now according to the `lkb` attribute of `b(0)`, his energy level is 55%, he is at `pt(3,1)` going east (`(myDir(b(0), pt(-1, 0)) @ 9)`) and last action was `mv(b(0), pt(-1, 0))`. This is the knowledge `b(0)` had when proposing the recorded last action (at time 9).

The environment knowledge base reflects the state after the bots actions have been carried out. Thus, at time 10 `(b0)` is now at `pt(2,1)` with energy level 50% as expected.

Now we search for a system configuration in which `b(0)` is at the charging station (`(atloc(b(0), pt(3,2))`). We add `bound(20)` to the configuration to ensure a finite search space. The `[1]` specifies we only want the first solution, which will be the first time the bot visits the charging station.

```
search [1] {initC bound(20)} =>+
   {[b(0) : Bot |
      lkb : ((atloc(b(0), pt(3,2)) @ t:Nat) lkb:KB),
      atts:AttributeSet]
   aconf:Conf} .

Solution 1 (state 65)
states: 66
aconf --> bound(4)
[eI |
  clock(16) (atloc(b(0), pt(3, 2)) @ 16) (atloc(b(1), pt(3, 3)) @ 15)
  (energy(b(0), 20) @ 16) (energy(b(1), 20) @ 16) ...]
[b(1) : Bot |
  lkb : (clock(16)  (lastAct(mv(b(1), pt(0, -1))) @ 15)
   (atloc(b(0), pt(2, 1)) @ 14) (atloc(b(1), pt(3, 3)) @ 15)
   (myDir(b(1), pt(-1, 0)) @ 15) (energy(b(1), 25) @ 15) ...),
  ckb : ...,
  sensors : (locS obstacleS energyS),
  evs : (rcv(atloc(b(0), pt(3, 1)) @ 15) tick @ 0)]
atts:AttributeSet -->
  ckb : ... ,
  sensors : (locS obstacleS energyS),
  evs : ((tick @ 1) {{cv(100),u(1.0)},charge(b(0))} @ 0)
lkb --> clock(16) (lastAct(charge(b(0))) @ 16) (atloc(b(1), pt(3, 3)) @ 15)
  (myDir(b(0), pt(1, 0)) @ 16) (offTrack(b(0), 1) @ 16)
  (energy(b(0), 20) @ 16) ...
t:Nat --> 16
```

The result of a search is a substitution for variables on the righthand side of the search arrow =>+ matching the search pattern to the state found. We see that the time is 16 and from the `evs` attribute for `b(0)` (part of the `atts:AttributeSet` binding) `b(0)` has requested action `charge(b0)`. This also is recorded im `b(0)`'s local knowledge base using `(lastAct(charge(b(0))) @ 16)`. Notice that the energy level of `b(0)` has reached 20% which the cost of a unit move above the reserve threshold, `(ereserve(b(0), 15) @ 0)`.

3 Fault Models

The SAF provides a general mechanism, *fault models*, for adding faults to a SA system semantics. The fault models serve two purposes. One is to provide a setting for developing detection and diagnosis methods. The other is to support exploring the design space for soft agent behavior in terms of resilience and adaptability in the presence of imperfect

sensors and actuators and unpredictable environment perturbations. There are two kinds of fault, sensor faults and action faults, which we describe in the following. Environment perturbations such as obstacles are modeled as knowledge items in the environment KB. Suitable sensors are needed to detect these perturbations, which can be faulty!

First the parameters used to specify fault models are defined, next sensor faults and their effect on sensor readings are described, and finally, action faults and their effect on the execution of actions are described.

3.1 Fault Parameters

There are three sorts used to parameterize fault models: `FType`, `FVal`, and `FPars`.

`FType` is the sort of fault types. The framework currently provides two sensor fault types: a boolean fault type, `boolFT`, a fault that is either present or not; and a simple fault type, `simpleFT`, with fault effect parameterized by elements of sort `FPars`. `FVal` provides a super sort for collecting parameters with different structures.

The parameters for a fault of type `boolFT` are terms of sort `FPars` of the form `bFP(bp:Rat)`. `bp:Rat` specifies the probability of a fault. The parameters for a fault of type `simpleFT` are terms of sort `FPars` of the form `sFP(fp0:Rat,fp1:Rat,fv:FVal)`. The two rational arguments, `fp0:Rat,fp1:Rat`, are typically used as probability thresholds, for example determining if a sensor is broken (no reading) or not, and if not broken whether there is a reading error. The `FVal` argument is an interface sort. Each SA model and sensor will have its own subsorts.

3.2 Sensor Faults

Sensor faults can model imprecision of sensors, broken sensors, environmental interference such as mud on a camera or interference with a GPS signal. A sensor fault is specified by a term of sort `Info` of the form `sF(id:Id,s:Sensor,ft:FType,fp:FPars)` where `id:Id` is the identifier of the agents whose sensor `s:Sensor` is faulty. For example, `sF(b(0),locS,simpleFT,sFP(1/10,1/5,ptV(pt(0,1))))` specifies a simple location sensor fault for Patrol Bot `b(0)`. With probability $1/10$ the sensor gives no reading (`noLoc`) and if it gives a reading, the reading is off by 1 unit north with probability $1/5$.

The interpretation of a sensor fault model is defined by the function call

```
applySensorF(id:Id,s:Sensor,skb:KB,fm:KB,ekb:KB)}
```

which is invoked by the `readSensor` function as described in section 2.3.1. `id:Id` is the identifier of the agent reading its sensors, `s:Sensor` is the sensor being read, `skb:KB` is the sensor information in the absence of faults `fm:KB` is the fault model to apply, and `ekb:KB` is the environment KB where the sensor is being read. The result is a pair of knowledge bases, the faulty sensor knowledge and the updated environment KB.

For example, the Patrol Bot interpretation of a location sensor fault is given by the following conditional equation.

```
ceq applySensorF(id,locS,atloc(id,l) @ t0,
```

```
                (sF(id,locS,simpleFT,sFP(fp0,fp1,ptV(pt0))) @ t1),
                (rand(id,i,j) @ t4 ) ekb)
  = {atloc(id,l1) @ t, ekb (rand(id,i,s s j) @ t)}
if (fence(id,ll,ur) @ t2) clock(t) randInc(n) ekb0 := ekb
/\ rp0 := float(random( (n * j) + i) / randMax)
/\ rp1 := float(random( (n * (j + 1)) + i) / randMax )
/\ dl1 := if rp1 < fp1 then pt0 else pt(0,0) fi
/\ l0 := sum(l,dl1)
/\ l1 := (if (rp0 < fp0)
          then noLoc
          else (if inBounds(l0,ll,ur) then l0 else l fi) fi) .
```

Sampling a probability distribution is done using Maude's random number generator. The function `random(n:Nat)` returns the n^{th} random number in a random number sequence determined by choice of random seed when Maude is started. Dividing `random(n:Nat)` by `randMax` corresponds to sampling a uniform probability distribution. To ensure that each sample is different, each agent and the environment, is allocated a subsequence `(n * j) + i)` where n is the number of subsequences, i picks agent is subsequence, and j is the current position in agent is subsequence. This is tracked in the knowledge items `rand(id,i,j) @ t4` and `randInc(n)`. The need to update `rand` is why the environment KB gets updated.

Two random probabilities, `rp0` and `rp1`, are generated. If `rp0` is below the threshold parameter `fp0`, then the sensor is broken and the reading is `noLoc`. Otherwise, if `rp1` is below the threshold parameter `fp1`, then the error parameter, `pt0`, is added (as a vector) to the sensor reading giving the candidate sensor reading, `l0`. If `l0` is inside the fenced area, `inBounds(l0,ll,ur)`, it is returned otherwise the actual sensor reading, `l` is returned, along with the environment KB with the random counter for agent `id` `rand(id,i,j)` updated.

3.3 Action fault models

Action faults can be used to model action imprecision or failure, environmental interference such as wind blowing a drone or a steep hill reducing the effectiveness of a move action. As for sensors, we need names for action types. The sort `Act` is the sort of action (type) names. In the Patrol Bot example the names are `mvA` (move action) and `chargeA` (charge action). Similar to sensor faults, an action fault is specified by terms of sort `Info` of the form `aF(id:Id,a:Act,ft:FType,fp:FPars)`. For example, `aF(b(0),mvA,simpleFT,sFP(1/10,1/10,ptV(pt(-1,0))))` specifies a move action fault for Patrol Bot `b(0)` with probability $1/10$ of a broken actuator (no move), probability $1/10$ of a faulty move, if the actuator is not broken. with error given by the vector `pt(-1,0)`.

As discussed in section 2.3.2, the interpretation of an action fault model is defined by the `doEAct` function which returns an update to the environment KB. As an example, the effect of a simple move action fault is defined by the following equation.

```
ceq doEAct(t, mv(id, dl),
           (atloc(id,l) @ t0) (energy(id,e) @  t1)
```

```
          (rand(id,i,j) @ t5 ) ekb)
=
   (atloc(id,l1) @ s t) (energy(id,e1) @ s t)
   (rand(id,i,s s j) @ s t )
if (aF(id, mvA,simpleFT,sFP(fp0, fp1, ptV(pt2))) @ t6)
   (fence(id,ll, ur) @ t3)  randInc(n) ekb0
   := ekb
/\ rp0 := random( (n * j) + i) / randMax
/\ rp1 := random( (n * (j + 1)) + i) / randMax
/\ dl0 := if rp0 < fp0 then pt(0,0) else dl fi
/\ dl1 := if (rp1 < fp1 and rp0 >= fp0) then pt2 else pt(0,0) fi
/\ l0 := sum(sum(l,dl0),dl1)
/\ l1 := (if (inBounds(l0,ll,ur) and not(occupied(l0,ekb)))
          then l0
          else l fi)
/\ e1 := max(e - costMv,0)  .
```

Two random probabilities, `rp0` and `rp1`, are generated. If `rp0` is below the threshold parameter `fp0`, then the basic move component `dl0` is `pt(0,0)` (the move actuator is broken), otherwise it is the requested move, `dl`. If `rp1` is below the threshold parameter `fp1`, and the actuator is not broken, then the move error is `pt2`, otherwise it is `pt(0,0)` (no error). The tentative move result, `l0`, is the sum of the current location, the basic move and the error `sum(sum(l,dl0),dl1)`. If `l0` is inside the fenced area, `inBounds(l0,ll,ur)`, and it is not occupied according to the environment KB `ekb`, the new location is `l0` otherwise it is the original location `l`. In either case, the energy level is reduced by the cost of the attempted move `max(e - costMv,0)`. Thus the environment KB update for the agent with identifier `id` has three parts: the updated location `(atloc(id,l1) @ s t)`; the reduced energy `(energy(id,e1) @ s t)`; and the new position in the random number sequence `(rand(id,i,s s j) @ s t)` all time stamped with the incremented time `s t`. The sequence counter is incremented by two as two random numbers were used in the computation.

Much more complex location sensor ad move action faults are possible, for example using random choice of magnitude and direction of the error. The Bot Team case study uses these more complex models.

4 Protocols and Satisfaction

In order to detect when something goes wrong in an soft agent system, one need to know what is right! Furthermore, finding the cause when an goal is not achieved or a requirement/invariant is not satisfied it helps to have some idea of the steps expected to lead to success.

Inspired by rules for carrying out scientific experiments and clinical trials, we propose the notion of Soft Agent Protocol to specify expected behavior of an SA system. Intuitively, a protocol specifies observable events that should take place (stages or steps making progress) along with any ordering or timing conditions and possibly other conditions. To allow for flexibility and avoid specifying values, such as exact times, we propose that a protocol consist of a set of observable event patterns and constraints expressed in terms of

pattern variables, together with invariant constraints (defining what is called adverse events in clinical protocols).

Given a protocol, we need an associated notion of compliance or satisfaction – when does an execution of a soft agent system meet the requirements specified by that protocol. Using the intuition above, the idea is to map protocol event patterns to observable events of the execution and then check that the instantiated constraints hold.

4.1 SA Protocols

A SA protocol is a partial order of observable event patterns together with constraints on the pattern variables and possibly additional invariant constraints independent of the event pattern variables. Using event patterns rather than concrete events allows for compact specification and also for specifications that can be met under many conditions. For example, there is no need to specify concrete times for events, when relative times and time intervals are what is important; one may not want to specify precise locations, but only neighborhoods; and in some cases, the order between a pair of events may not matter as long as both happen.

SA event patterns are represented by symbolic knowledge items, i.e. time stamped information items where the time stamp or the parameters of the information part may be variables. The partial order is specified by ordering relations on the time stamp variables. We require that each event has a unique symbolic time stamp. Thus, a symbolic event can be identified by its time stamp and so multiple occurrences of the same information template can be distinguished. Invariants are typically used to represent safety envelopes, such as maintaining energy above a given minimum level or ensuring that the distance between pairs of agents is greater that a given minimum.

For a system with agents whose identifiers are the elements of `AgentIds`, a protocol $\mathcal{P}$ is a tuple $(TL, Eps, LO, GO, CS, Inv)$ where

- TL is a list of timelines, where the timeline for an agent is the list of time stamp variables for events under control of that agent.
- Eps is the set of protocol event patterns
- LO is the set of local order constraints for each agent. A local order constraint for an agent has the form `s0:Time < s1:Time` where `s0:Time, s1:Time` are time stamp variables in the timeline for that agent.
- GO is the set of global order constraints, `s0:Time < s1:Time` where `s0:Time, s1:Time` belong to the timelines of different agents.
- CS consists of additional constraints on protocol event pattern variables.
- Inv is the set of invariant constraints. These are implications `epat => c` where `c` is a constraint on the variables of the event pattern `epat`.

The following is an example protocol (in the syntax used by our constraint solving tool, % begins a comment line) for two Patrol Bots on a 7×5 grid ($0 \leq x \leq 6, 0 \leq y \leq 4$). Bot

`b(0)` starts at $(0, 1)$ and should traverse the grid going to $(6, y)$ and back staying within 1 unit of $y = 1$. Bot `b(1)` starts at $(6, 3)$ and should traverse the grid going to $(0, y)$ and back staying within 1 unit of $y = 3$. The bots should maintain at least 15% energy (by recharging at the charging station as needed). The protocol does not specify charging actions.

```
AgentIds: b(0) b(1)

timeline(b(0), s00:Time, s01:Time, s02:Time)
%events(b(0)):
  atloc(b(0),pt(0,1)) @ s00:Time
  atloc(b(0),pt(6,n01:Nat)) @ s01:Time
  atloc(b(0),pt(0,n02:Nat)) @ s02:Time

timeline(b(1), s10:Time, s11:Time, s12:Time)
%events(b(1)):
  atloc(b(1),pt(6,3)) @ s10:Time
  atloc(b(1),pt(0,n11:Nat)) @ s11:Time
  atloc(b(1),pt(6,n12:Nat)) @ s12:Time

%localOrder(b(0)):
  s00:Time < s01:Time
  s01:Time < s02:Time
%localOrder(b(1)):
  s10:Time < s11:Time
  s11:Time < s12:Time

%otherConstraint:
% max delay between patrol allowing slack time for charging
% time2patrol + time2charge + time2station = 6 + 5 + 2 = 13
  s01:Time - s00:Time < 14
  s02:Time - s01:Time < 14
  s11:Time - s10:Time < 14
  s12:Time - s11:Time < 14

% Invariants
% staying on track
atloc(b(0),pt(n0x:Nat,n0y:Nat)) @ s20:Time
      => n0y:Nat >= 0 and 2 >= n0y:Nat
atloc(b(1),pt(n1x:Nat,n1y:Nat)) @ s21:Time
      => n1y:Nat >= 2 and 4 >= n1y:Nat
energy(b(0),e0:Nat) @ s30:Time => e0:Nat >= 15
energy(b(1),e1:Nat) @ s31:Time => e1:Nat >= 15
```

Typically in an execution of the protocol the local order for an agent is a total-order. The order need not be totally specified, meaning that multiple orders are acceptable. Some actions may cause change in more than one observable and hence the corresponding events will appear simultaneous.

4.2 Protocol Satisfaction by Event Logs

Not all knowledge items make sense as events. The realization of a protocol event is the result of an agent action, that is it is among the knowledge items returned by `doAct` or

`envAct`. In particular, for the agents to execute the protocol these events should be under the control of the agents, not environment actions. In the case of the simple patrol bots protocol events are location and energy knowledge items. If the bots had cameras, knowledge items relevant to taking pictures could be protocol events.

We define the event language, EL, of a protocol to be the knowledge items generated by the information constructors used in the event patterns appearing in the protocol (both the partial order and the invariants). This language is specified by a sub-signature of the Maude knowledge base signature. For example, the event language of the simple patrol bot protocol above is given by the constructors

```
atloc : Id Loc -> Info
energy : Id Nat -> Info
```

An event log, EvL, over a given event language and `AgentIds` is a concrete set of events, i.e., ground knowledge terms whose information component is in the given language and whose agent identifiers are in `AgentIds`.

Let $\mathcal{P} = (TL, Eps, LO, GO, CS, Inv)$ be a protocol for `AgentIds` with event language, EL. Let EvL be an event log over `AgentIds` and EL. EvL satisfies $\mathcal{P}$ ($EvL \models \mathcal{P}$) is defined as follows.

- $EvL \models \mathcal{P}$ iff $EvL \models (Eps, LO, GO, CS)$ and $EvL \models Inv$
- $EvL \models (Eps, LO, GO, CS)$ iff there is a substitution, σ, embedding Eps in EvL ($\sigma(Eps) \subseteq EvL$) such that $\sigma(LO)$, $\sigma(GO)$, $\sigma(CS)$ are all true.
- $EvL \models Inv$ iff $EvL \models$ `ep => c` for each invariant expression `ep => c` in Inv
- $EvL \models$ `ep => c` iff for each instance σ_0(`ep`) in EvL, σ_0(`c`) is true.

4.3 Protocol Satisfaction by Execution Traces

In section 2.3.2 a generic mechanism for collecting execution logs was described. Here we consider logs appropriate for checking satisfaction and detecting deviations from a protocol and derive event logs from these execution logs.

Recall that a log item has the form `{t, acts, lconf}` where `t` is a timestamp, `acts` an action set, and `lconf` a set of configuration elements. `t` is the time just before the actions are carried out. A log Lg is a sequence of log items with time increasing. We write $len(Lg)$ for the length of the sequence. Define $Lg[t]$ to be the (unique) log item of Lg with time stamp `t`, i.e. the log item of the form `{t, acts, lconf}`. $Lg[t]$ is `nil` if there is no such log item. Note that the logs generated by `updateLog` have a log item for each time from 0 up to (not including) the length of the log.

Here we restrict attention to logged configurations `lconf` that have the form

```
[eid |ekbl] ... [id : Cl | lkb: lkbl] ... ).
```

In particular, only the `lkb` attribute of agents is kept. How much of the execution configuration knowledges bases to keep in the log depends on the intended use. If one is only checking protocol satisfaction, then only the knowledge items in the protocol event language are needed. For detecting deviations or diagnosis more information is usually needed. For checking properties other than protocol satisfaction, log items might collect metadata rather than environment and agent knowledge.

We begin with some notation. s Let Tr be an execution trace as defined in section 2.4. For j from 0 up to (not including) $len(Tr)$, define $Tr[j]$ to be the j^{th} trace element

```
C.j -doTasks-> C.ja -acts.j-> C.j+1
```

thus j is the time in `C.j`, and $j+1$ is the time in `C.j+1`.

Let Lg be a log and for t a timestamp of Lg we let the log item $Lg[t]$ be `{t,acts,lconf}` where `lconf` has the form

```
[eid |ekbl]] ... [id : Cl | lkb: lkbl.id] ... ).
```

then we define

- $Lg[t][eid] =$ `ekbl`
- $Lg[t][id] =$ `lkbl.id`
- $Lg[t][a] =$ `acts`

Finally, if `[eid | ekb] ... [id : cl | lkb: lkb.id ...] ...` is the final configuration of $Tr[t]$ then

- $Tr[t][eid] =$ `ekb`
- $Tr[t][id] =$ `lkb.id`
- $Tr[t][a]$ – the actions labeling the timeStep transition of Tr[t]

Let `AgentIds` be the identifiers of agents in the initial configuration of Tr (and hence of each configuration of Tr).

Define

- A log Lg is a log for Tr if $Lg[j]$ is a log item for $Tr[j]$ for j a timestamp of Lg.
- $Lg[t]$ is a log item for $Tr[t]$ iff
  - $Lg[t][eid]$ is a subset of $Tr[t][eid]$
  - $Lg[t][id]$ is a subset of $Tr[t][id]$ for id in `AgentIds`
  - $Lg[t][a]$ = Tr[t][a]

Lg is a full log for Tr if the set of times of Lg is the same as the times of Tr. Note that if Lg is a full log for Tr then Tr and Lg have the same length.

$LMax(Tr)$ is the full log for Tr defined by

- $LMax(Tr)[t][eid] = Tr[t][eid]$

- $LMax(Tr)[t][id] = Tr[t][id]$ for id in `AgentIds`

Given a log Lg

1. the *environment* view of events of Lg, $EvsE(Lg)$ is defined by

$$EvsE(Lg) = \bigcup_{t \text{ a timestamp of } Lg} Lg[t][eid]$$

2. the Agents view of events of Lg, $EvsA(Lg)$ is defined by

$$EvsA(Lg) = \bigcup_{t \text{ a timestamp of } Lg} EvsA(Lg[t])$$

where

$$EvsA(Lg[t]) = \bigcup_{id \text{ in } \texttt{AgentIds}} Lg[t][id]$$

Trace satisfaction. We now define satisfaction of a protocol from the environment and agent perspective for traces and logs by reducing it to associated event logs.

- $Tr \models_E \mathcal{P}$ if $EvsE(Tr) \models \mathcal{P}$
- $Tr \models_A \mathcal{P}$ if $EvsA(Tr) \models \mathcal{P}$
- $Lg \models_E \mathcal{P}$ if $EvsE(Lg) \models \mathcal{P}$
- $Lg \models_A \mathcal{P}$ if $EvsA(Lg) \models \mathcal{P}$

A useful log should collect enough information such that if Lg is a log for Tr then

$$Tr \models_E \mathcal{P} \leftrightarrow Lg \models_E \mathcal{P} \text{ and } Tr \models_A \mathcal{P} \leftrightarrow Lg \models_A \mathcal{P}$$

A question remains as to how to determine, given Lg, what are the traces Tr such that Lg is a log for Tr. A key issue is the range of starting states for Tr.

5 BotTeam Case Study

The BotTeam case study was designed to explore simple coordination in the presence of faults. In this section we describe the protocol the agents should follow and the model specific components for the BotTeam. Experiments with applying fault models and diagnostics will be summarized in section 8.

The BotTeam consists of two bots with different roles, an initializer and a finisher. They are to do maintenance at a list of locations (a parameter). The initializer applies its treatment twice resulting in treat stage 2 status. Then the finisher applies its treatment twice, resulting in treat stage 4 status. The finisher should complete its treatment at each location soon after the initializer finishes. The initializer just needs energy to carry out its treatment. The finisher needs some material available at a supply depot. The protocol does not specify when bots should recharge or visit a supply depot.

5.1 BotTeam protocol

We considered two BotTeam protocols, one with two locations to treat, and one with three locations. The two location protocol was instantiated with two different location lists. We show the protocol for one of the two location scenarios. The event pattern language for the BotTeam protocol consists of the following patterns

```
atloc(id:Id,l:Loc) @ s:Time          %agent location
treatStage(l:Loc,j:[0-4]) @ s:Time   %treatment stage at l:Loc
energy(id:Id,e:Nat) @ s:Time         %energy level of bot id:Id
```

The protocol for two locations, `pt(4,0)` and `pt(5,4)` on a 7×6 grid (in the syntax of our constraint solving tool) follows.

```
%timelines
timeline(b(0), s00:Time, s01:Time, s02:Time, s03:Time, s04:Time, s05:Time)
timeline(b(1), s10:Time, s11:Time, s12:Time, s13:Time)

% b(0) controlled events
atloc(b(0),pt(4, 0)) @ s00:Time
treatStage(pt(4, 0),2) @ s01:Time
atloc(b(0),l0:Pt2) @ s02:Time
atloc(b(0),pt(5, 4)) @ s03:Time
treatStage(pt(5, 4),2) @ s04:Time
atloc(b(0),l1:Pt2) @ s05:Time

% b(1) controlled events
atloc(b(1),pt(4, 0)) @ s10:Time
treatStage(pt(4, 0),4) @ s11:Time
atloc(b(1),pt(5, 4)) @ s12:Time
treatStage(pt(5, 4),4) @ s13:Time

% b(0) local order
s00:Time < s01:Time
s01:Time < s02:Time
s02:Time < s03:Time
s03:Time < s04:Time
s04:Time < s05:Time

% b(1) local order
s10:Time < s11:Time
s11:Time < s12:Time
s12:Time < s13:Time

% bot sequencing
s02:Time < s10:Time
s05:Time < s12:Time

%%%%%%%%%% other constraints
% bot0 moves away from target
l0:Pt2 =/= pt(4, 0)
l1:Pt2 =/= pt(5, 4)

% max delay between stages
```

```
s11:Time - s01:Time < 10
s13:Time - s04:Time < 10
% max delay between targets
s03:Time  - s01:Time < 15

%invariant
energy(id:Id,e:Nat) @ s:Time => e >= 35
```

5.2 BotTeam agent model

The two bot roles are represented by the classes `BotI` (initializer) and `BotF`. Initially the bots are given a list of target locations. In `ready` mode `BotI` picks the next target location, removes it from the list, shares the selection with `BotF`, moves to the selected target, does its treatment action twice, and becomes `ready`. If the target location list is empty, then a `ready` bot heads home (its starting location). When `BotF` receives new target information it moves to that target, possibly stopping at the supply depot to load. When `BotI` has finished and left, `BotF` does the final 2 stages of treatment. Each bot maintains its energy level by going to the charging station when ever energy gets to low.

Actions available to the agents are

- `charge(id:Id)` at charging station, increases energy level.
- `mv(id:Id,dir:Pt2)` moves one unit in the direction of the vector. `dir:Pt2`
- `treat(id:Id)` increments treatment stage by 1, requires some energy for either bot. For `BotF` it also uses some of the supply.
- `load(id:Id)` available to `BotF` at the supply depot to replenish its supply.

The bot sensors are `obstacleS` for detecting nearby obstacles, including other agents; `locS` and `energyS` for location and energy level; `treatS` to determine the treatment level at a target location; and `supplyS` to measure the supply level (`BotF` only). Reading `obstacleS` produces location and class of the obstacle, reading `locS` resp. `energyS` produces current location resp. energy of the agent. Reading `treatS` at a target location produces a knowledge item of the form `treatStage(tgt:Loc,n:[0-4]) @ t:Time` where `n` is the accumulated treatment stage. Other key knowledge items are

- `targets(locs:LocList) @ t:Time` the list of target locations remaining to visit
- `target(loc) @ t` current target, initially no target info
- `targetF(loc) @ t` used by BotF to remember its current target in case BotI shares new target information before BotF is done.

The high level plan of the agents is represented as an automaton with states, called modes, `ready`, `enroute(loc:Loc)`, `treat`, `loading`, and `done`. Figure 1 shows the automaton transition system for the two bot classes.

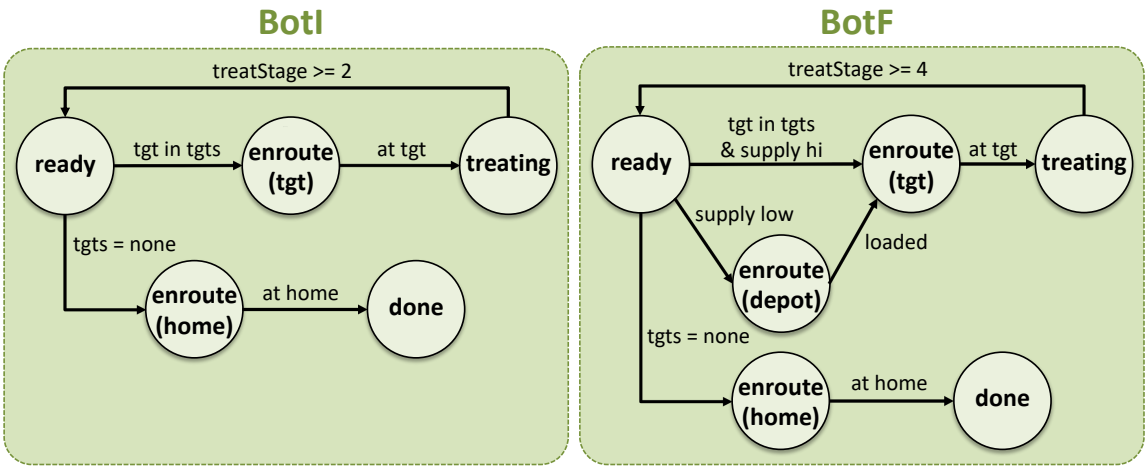


Figure 1: The automaton transition system for the two bot classes

The automaton transition system is implemented by the BotTeam `proSensors` function using knowledge items `mode(id:Id,m:Mode) @ t:Time` to store the current state. In each mode a subset of actions is available and the bots use a simple soft constraint solving system to rank the available actions. Under the hood, the energy concern of the soft constraint system drives the agent to the charging station when energy is low. Thus the high-level automata does not need to worry about this.

6 Diagnosis

Given definitions of system behavior (traces), protocols specifying events and invariants, and a notion of satisfaction $Tr \models \mathcal{P}$ we are interested in situations in which a SA system fails to satisfy a protocol. That is, we assume that under normal/ideal conditions with sensors and actuators working as expected, and absence of interference from the environment, the SA system will succeed in satisfying the protocol. In the real world, sensors may fail or be inaccurate, actuators may not have the expected effect, weather conditions and objects in the external environment may interfere or compete for resources. Under these conditions the SA system may fail to satisfy the protocol. SA systems are intended to be resilient and robust to faults and unexpected situation so its important to determine the cause of failure when it happens.

We first informally classify what can go wrong. That is, in what ways can satisfaction of a protocol fail. These are called *deviations*. The aim is a classification that aids detection and fault ascription. For each class of deviation we discuss informally the process for identifying the cause.

We next present a mapping of the satisfaction problem to an SMT sat solving problem. The SMT tool can check satisfaction incrementally, identifying minimal points of failure. We then present a simple local detection algorithm that can be used to suggest candidate causes of minimal points of failure found by SMT solving. This is done by checking an agents expected effect of actions against observed effects. Finally candidate causes can be checked by what we call *gedanken* experiments, which are inspired by ideas from counter

factual reasoning. The idea is to generate a trace that follows the faulty execution up to the identified cause and chooses a continuation that avoids the cause but otherwise follows the original trace as closely as possible.

6.1 Deviations

In the following we fix a scenario:

- a SA model and initial configuration, `iC`, with agent's identifiers in `AgentIds`, and a family of fault models, FM. There are fixed sets of sensors and actions available to each agent.
- an Event language EL (sublanguage of the SA model knowledge language)
- a protocol $\mathcal{P} = (TL, Eps, LO, GO, CS, Inv)$ over EL
- An event log EvL from an execution trace from $Tr(\{\texttt{iC}\}, FM)$.

EvL can be from the environment perspective or the agent perspective. $Tr(\{\texttt{iC}\}, FM)$ stands for the set of traces $Tr(\{\texttt{addFaultModel(iC,fm)}\})$ for `fm` in FM.

Recall that TL is a list of timelines, one for each agent, where the timeline for an agent is the list of time stamp variables for events under control of that agent. Eps is the set of protocol event patterns over EL. LO is the set of local order constraints for each agent. GO is the set of global order constraints, relating events controlled by different agents. CS contains any additional constraints on variables of Eps. Inv contains invariant constraints, `epat => c`, with variables disjoint from the variables of Eps.

Suppose $EvL \not\models \mathcal{P}$. Thus there is no instantiation of P that can be embedded in EvL satisfying all the constraints (LO, GO, CS, Inv). What is wrong? What part of the protocol is unsatisfiable? We call the unsatisfied bits deviations and write $Dev(EvL, \mathcal{P})$. Since the invariants Inv quantify over all matches and are independent from the event patterns that make up the partial order of events, we consider two cases: (1)$EvL \not\models Inv$ or (2) $EvL \not\models (Eps, LO, GO, CS)$. Note that the cases are not disjoint in the sense that both parts of the protocol could be unsatisfiable. In case (1) $Dev(EvL, \mathcal{P})$ includes the set of I_j in Inv such that $EvL \not\models I_j$. In the case (2) we consider three further cases: (2a) embeddings exist such that

$$EvL \models (Eps, LO, GO)$$

but none satisfy CS; (2b) full embeddings of Eps in EvL exist but none satisfy the all of the ordering constraints in (LO, GO); and (2c) no full embedding of Eps in EvL exists.

In case (2a) let $\sigma_0, \ldots, \sigma_k$ enumerate the embeddings of (Eps, LO, GO) in EvL and let C_j be the subset of CS such that $\sigma_j(C_j)$ is false. Then $Dev(EvL, \mathcal{P})$ includes the set $(\sigma_0, C_0), \ldots, (\sigma_k, C_k)$. The case (2b) is similar, but here the C_j are subsets of (LO, GO) as well as possibly from CS. In case (2c) we consider maximal embeddings $\sigma_0, \ldots, \sigma_k$ and cases as for (2a,b) restricting constraints to $dom(sb_j)$, adding the missed events to the deviations.

6.2 Towards finding causes

We continue the scenario of the previous section (6.1). Thus we are given a protocol $\mathcal{P}$, an event log EvL from log Lg for Tr in $Tr(iC, FM)$ as in the previous section. We consider identifying the cause of deviations in $Dev(EvL, \mathcal{P})$ according to the identified cases.

6.2.1 Case 1. Invariant Violation

Suppose invariant I fails. We assume the agents have a 'plan' to maintain the invariants and that there is an execution that does. Either an agent has chosen the wrong action among apparently equivalent choices, or some action was chosen based on faulty information, or some action didn't have the expected effect, (or an agent did no action at some point thus delaying progress).

Example i. Suppose the protocol contains an invariant of the form

```
energy(id:Id,e:Nat) @ s:Time => e:Nat >= 15
```

Let t be least such that `energy(i,n) @ t` is in EvL and $n < 15$. In the absence of additional information, we consider that some action or subgoal used more energy than expected, or that the energy sensor is faulty. The process is to look in the log prior to t to find execution deviations related to energy (charging, actions that use energy, energy sensors) that might be the cause and then do a gedanken experiment to check to see if any of these deviations could be the cause. If we know how the agent is attempting to maintain a given invariant, that information could be used to better pinpoint the cause of failure. For example, in the soft constraint approach to choosing actions, a constraint for the energy concern to be non-zero (hence disallowing an action) is for the energy after the action to be at least the required reserve plus the cost to get to a charging station. Knowing this we can look for the time when this constraint fails and find the fault that caused this failure. For example an unknown obstacle, possibly another agent, occupies some point in the shortest path to a charging station, could increase the actual cost to get to a charging station. A future improvement of our diagnosis procedure is to provide a mechanism to connect selected soft constraint concerns to protocol invariants.

Example ii. Suppose we have a safe distance requirement formalized as an invariant of the form

```
(atloc(id0:Id,l0:Loc) @ s:Time) (atloc(id1:Id,l1:Loc) @ s:Time)
=>
distance(l0:Loc,l1:Loc) > safeDistance
```

and suppose EvL contains `(atloc(b(0),loc0) @ t) (atloc(b(1),loc1) @ t)` where `distance(loc0,loc1) <= safeDistance` and t is the earliest time this violation occurs. Thus at the previous time the invariant held and one or both agents moved into the danger zone. Assuming no agent purposely violated the invariant, we need to look for execution deviations where an agent has bad location information (out-of-date information about another agent, or faulty obstacle sensor), or a faulty move action. These

are the sensors and actions that can effect the invariant. Again, if there were information about how the agents are attempting to maintain the invariant, the cause might be pinpointed more precisely.

6.2.2 Case 2a

In this case there are embeddings of Eps into EvL such that $EvL \models (Eps, LO, GO)$ but none satisfy CS. For example in the BotTeam case study, the protocol contains event patterns `treatStage(l,2) @ s02:Time` and `treatStage(l,4) @ s12:Time` and the constraint `s12:Time - s02:Time < 10`. The event log contains unique instances `treatStage(l,2) @ 15` and `treatStage(l,4) @ 28`. Bot `b(0)` is responsible for the first event (`s01:Time` is in the timeline for `b(0)`) and `b(1)` is responsible for the second. Either `s02:Time` $= 15$ is too early or `s12:Time` $= 28$ is too late. Without further information we look for execution deviations relevant to movement/treatment actions of `b(1)` and check to see if any (one or more) can be blamed. If we know that `b(0)` has a constraint that `b(1)` must be ready to act when `b(0)` is done, then we can look for execution deviations where that constraint fails, for example inaccurate information about the location of `b(1)`, or `b(1)` appeared ready at some point but did not stay that way.

6.2.3 Case 2b.

In this case, all event patterns can be instantiated but some ordering constraints fail.

Hypothetical example Suppose two different actions are to be done at location `l` resulting in information items `done(l,act1)`, `done(l,act2)` respectively, suitably time stamped. Furthermore, `act1` should be done before `act2`. Suppose `done(l,act1) @ 22`, `done(l,act2) @ 15` are part of the embedded protocol, showing `act2` was done before `act1`. What went wrong?

If the actions are done by different agents, the second agent may think `act1` has been done due to a faulty sensor. If the actions are to be done by one agent, we assume under normal conditions the agent behaves properly. Maybe `act1` was executed but it had the effect of `act2`, and dually when `act2` was executed the effect of `act1` happened. For example, maybe the actions involve applying treatments from different tubs, and the tubs were permuted.

6.2.4 Case 2c.

In this case no full embedding exists, and so there are missing events. We first consider maximal subsets Eps_0 of Eps such (Eps_0, LO_0) is satisfied in EvL, where LO_0 is the restriction of LO to variables in Eps_0. We can also consider the maximal embeddable subsets Eps_0 ignoring the local order constraints.

What causes the missing events? Let `ep` be a missing event pattern and with `id` the identifier of the responsible agent (the time stamp variable of `ep` is in the timeline of `id`). We consider three cases: (2c.i) the responsible agent is stuck, either unable to act, or going in a loop; (2c.ii) the agent has executed an action `act` such that `doAct(act,kb)`

should generate an instance of ep, but the action did not have the expected effect (and the agent didn't try again); (2c.iii) the agent is busy doing other tasks it is possible that if the execution continues the event will appear, In case (2c.i) we need to determine why the agent is stuck. In the absence of further information, we look for execution deviations that might lead to this condition. If we have more information about the agents rules for ranking actions we can likely be more precise. For example, if there are no permissible actions, perhaps some invariant failed, some preconditions to being able to act fail. Information about the agents intermediate goals may also help.

For each maximal subset Eps_0 as above we can also identify deviations in the global ordering and other constraints restricted to the variables in Eps_0 in the same way as for full embeddings.

7 Tools

In this section we describe the code developed to partially automate the detection and diagnosis process: generating execution logs, extracting event logs from an execution log; identifying protocol deviations; and listing execution deviations; and checking candidate causes. The section concludes with a discussion of the scripts and visualization tools used to partially automate the process.

7.1 Generating logs

Generating a log has two realizations. The first is instrumenting the execution so that the final state contains not only the final system configuration, but also a log of the execution. The second is generating a log as a term that can be used as input for other functions. These are different because the result of rewriting is just text that is printed as part of the Maude read-eval-print loop. However, due the Maude's support for reflection, going from the first to the second is easy as we will see below.

To instrument the execution, recall that the timeStep rule provides a hook and data structures for log generation, as well as the function updateLog which invokes the model specific function kbLog to compute the configuration component of a log item. In the Patrol Bot case study kbLog collects the agent sensor readings from the environment for the environment part of the log item, and collects the location and energy knowledge items for each agent log component. This is formalized by the following equations

```
eq kbLog(aconf,lconf) = kbLogX(aconf,lconf,none) .

eq kbLogX([id : cl:BotClass | lkb : lkb, (sensors : sset), attrs]
          aconf [eid | ekb], lconf,skb) =
    kbLogX(aconf [eid | ekb],
           lconf [id : cl:BotClass | lkb : getDoActInfo(id,lkb) ],
           addK(getSensorsKI(id,sset,ekb),skb)) .
eq kbLogX(aconf [eid | ekb],lconf,skb) = lconf [eid | skb] [owise] .

eq getDoActInfo(id,kb) =
   (getLocInfo(kb,none))  --- all location information
```

```
   (getEnergyInfo(id,kb)) .
```

The definition of `kblog` for the other case studies is similar.

The function `genLog`, defined by the following equation uses the logging mechanism and reflection to produce a log as a term. It reflects the term `asys` that specifies the initial system configuration up to the metalevel, does n rewrites using `metaRewrite`, reflects the result down to the object level and returns the log configuration element.

```
ceq genLog(asys,n) = getLog(getConf(asys1))
if asysT := upTerm(asys)
/\ res? := metaRewrite(['SCENARIO], asysT,n)
/\ asys1T := if res? :: ResultPair then getTerm(res?) else asysT fi
/\ asys1 := downTerm(asys1T,{(nil).Log})
.
```

Extracting Event Logs. To check satisfaction of a SA protocol we need to extract an event log from an execution log. Given a log `lg`, the function `flatLog(lg,{none,none}` return a pair `{ekbAll,lkbAll}` where `ekbAll` is the union of the environment knowledge bases in the log items of `lg` and `lkbAll` is the union of the `lkb` attributes of each agent in the log items of `lg`. That is `ekbAll` is $EvsA(\texttt{lg})$ and `lkbAll` is $EvsE(\texttt{lg})$ as define in section 4.3.

7.2 Protocol Deviations

We developed a tool called `deviate` to detect deviations from a protocol by transforming a protocol plus event log into a logical theory and using the Yices SMT solver to find models of the event partial order component.

To do this `deviate` first translates the event log into a complete set of atomic and negated atomic formula (clauses), with no free variables, in a finite language over a finite domain. It does this by assuming that if an event does not occur in the log, then the event is in fact false. The protocol is then translated into a simple set of quantifier free constraints over the theory constructed from the log. The satisfiability of the constraints are then checked by using, the python interface to, the Yices SMT solver. If there is no solution, then we can further investigate the cause by looking at various sub problems. Either by limiting the events in the log to an initial segment, or by restricting the form of the constraints generated from the protocol. In this manner, we can further clarify why the initial constraint set has no solution.

The usage of `deviate` is:

```
usage: deviate [-h] [--verbose] [--debug] [--all] [--incremental] [--frontier]
               [--timeline TIMELINE] [--entire] [--missing] [--unsat]
               [--consistency] [--configuration CFG_FILE]
               [--theory THEORY_FILE]
               log_file constraint_file
```

In the example below, `deviate` takes two inputs, `log_file` for event logs (i.e., `initS1alocsf1xEventsEnv.txt`) extracted from execution traces as explained in

Section 7.5 and `constraint_file` for a protocol (i.e., `protocol_initS1a.txt`) as explained in Section 5.1. The example output indicates the event log does not satisfy the protocol.

```
bash-3.2$ deviate initS1alocsf1xEventsEnv.txt protocol_initS1a.txt
Maximum time = 24
FYI: no events at [7, 16].
No solution: smt_stat = 4
```

To further investigate, one can use `-i` or `-incremental` option as below, which shows there is level 0 unsat core for `treatStage` event.

```
bash-3.2$ deviate initS1alocsf1xEventsEnv.txt protocol_initS1a.txt -i
Maximum time = 24
FYI: no events at [7, 16].
Level 0: context.check_context_with_assumptions returned UNSAT
Level 0 unsat core is:

(treatStage (pt 5 4) 4 s13)
```

The `-fv` or `-frontier` option finds the maximal satisfiable subprotocols.

```
bash-3.2$ deviate initS1alocsf1xEventsEnv.txt protocol_initS1a.txt -fv
Maximum time = 24
FYI: no events at [7, 16].
The timeline interpretation is: Timeline timestamps are distinct

(0, 0)
SAT
  s00 is 5
  s10 is 17
(0, 1)
SAT
  s00 is 5
  s10 is 15
  s11 is 19
(0, 2)
UNSAT

Unsat core:

(treatStage (pt 4 0) 4 s11)
(atloc (b 1) (pt 5 4) s12)
(< (+ s10 (* -1 s11)) 0)
(< (+ s11 (* -1 s12)) 0)

(1, 0)
SAT
  s00 is 5
  s01 is 11
  s10 is 17
(1, 1)
SAT
  s00 is 5
  s01 is 11
  s10 is 17
  s11 is 19
(2, 0)
SAT
  s00 is 5
  s01 is 11
  s02 is 15
```

```
  l0 is pt_3_2
  s10 is 17
(2, 1)
SAT
  s00 is 5
  s01 is 11
  s02 is 15
  l0 is pt_3_2
  s10 is 17
  s11 is 19
(3, 0)
SAT
  s00 is 5
  s01 is 11
  s02 is 15
  l0 is pt_3_2
  s03 is 22
  s10 is 17
(3, 1)
SAT
  s00 is 5
  s01 is 11
  s02 is 15
  l0 is pt_3_2
  s03 is 22
  s10 is 17
  s11 is 19
(4, 0)
UNSAT

Unsat core:

(< (+ s03 (* -1 s04)) 0)
(atloc (b 0) l0 s02)
(< (+ s01 (* -1 s02)) 0)
(< (+ s02 (* -1 s10)) 0)
(treatStage (pt 5 4) 2 s04)
(treatStage (pt 4 0) 2 s01)
(atloc (b 1) (pt 4 0) s10)
(< (+ s00 (* -1 s01)) 0)
(< (+ s02 (* -1 s03)) 0)

Frontier: set([(0, 2), (4, 0)])
```

To check for missing protocol events in the log, per timeline, `-m` or `-missing` option can be used.

```
bash-3.2$ deviate initS1alocsf1xEventsEnv.txt protocol_initS1a.txt -m
Maximum time = 24
FYI: no events at [7, 16].
There are 2 timelines (Timeline timestamps are distinct)

UNSAT at level 4, i.e. timestamp s04:Time

Unsat core :

(treatStage (pt 5 4) 2 s04)

UNSAT at level 2, i.e. timestamp s12:Time

Unsat core :

(atloc (b 1) (pt 5 4) s12)
```

The unsat core according to Yices SMT solver can be found by using `-u` or `-unsat` option.

```
bash-3.2$ deviate initS1alocsf1xEventsEnv.txt protocol_initS1a.txt -u
Maximum time = 24
FYI: no events at [7, 16].
context.check_context_with_assumptions returned UNSAT
(< (+ s12 (* -1 s13)) 0)
(< (+ s03 (* -1 s04)) 0)
(< (+ s05 (* -1 s12)) 0)
(< (+ s00 (* -1 s01)) 0)
(< (+ s01 (* -1 s02)) 0)
(< (+ s04 (* -1 s05)) 0)
(treatStage (pt 5 4) 4 s13)
(< (+ s02 (* -1 s03)) 0)
```

7.3 Execution Deviations

An execution deviation is a result of an agent action that differs from what is predicted by the agents model, or a broken sensor reading. An execution deviation (sort `EDev`) has one of the forms

```
{ract,lkb,lkb0,ekb0}
{ract,lkb,lkb0,{lkb1,ekb0}}
```

where `ract` is the ranked action executed, `lkb` is the executing agent's local knowledge base, `lkb0` is the knowledge items updated when the action is carried out in `lkb`, and `ekb0` is the knowledge items in the environment knowledge base, after the action, that disagree with `lkb0`. In the second case `lkb1` are items in `lkb0` that have no corresponding update in the environment knowledge base (for example if a move action failed, `lkb0` will have a location update, but the environment will not). [1] An execution deviation element (sort `EDevsElt`) is a time stamped set of execution deviations `[t:Time,edevs:EDevSet]`. The function `getEDevsL` uses `genLog` to produce an execution log, then extracts a list of execution deviation elements from (a segment of) this log using `log2edevsl` which applies `litem2edevs` to each element of the log (segment).

```
ceq getEDevsL(asys,bconf,n,j,j0) = log2edevsl(log0,bconf)
if  log := genLog(asys,n)
/\ log0 := subLog(log,j,j0) .
```

`bconf` is a base agent configuration that contains the constant knowledge needed to compute the effects of an action. For example, the grid boundaries are constant knowledge needed to compute the effects of a move action. For each logitem `{t,racts,lconf}`, the function `litem2edevs` collects

```
act2edevs(t,{rval,act},lconf,bconf))
```

for each ranked action `{rval,act}` in `racts` along with the deviations due to broken sensor readings, and adds the time stamp `t` to form an `EDevsElt`.

The function `act2edevs` is the core. It calls the function `doAct` with arguments the action, the agents augmented local knowledge base and the current time. It then uses

[1] Recall that in a logitem, the environment knowledge base reflects the state after the actions have been carried out while the agent local knowledge bases represent the state before the actions, i.e. the information used by the agents to choose actions.

kb2edevs to extract the elements ekb0 of the environment knowledge base ekb that disagree with the results of doAct, and the elements lkb1 of lkb0 that can't be compared.

```
ceq act2edevs(t,{rval,act},lconf,bconf) =
  (if (ekb0 == none and lkb1 == none)
   then none
   else (if (lkb1 =/= none)
         then {{rval,act},lkb,lkb0,{lkb1,ekb0}}
         else {{rval,act},lkb,lkb0,ekb0}
         fi)  fi)
if id := actId(act)
/\ [id : cl | lkb : lkb, attrs] [eid | ekb] lconf0 := lconf
/\ [id : cl | lkb : bkb, attrs0] bconf0 := bconf
/\ lkb0 := doAct(act,addK(bkb,lkb) clock(t))
/\ {lkb1,ekb0} := kb2edevs(lkb0,ekb)
     --- ekb0 is the elements of ekb that disagree w lkb0
```

7.4 Checking Candidate Causes

We continue the scenario of the previous section (6.1 with protocol $\mathcal{P}$, an event log EvL from log Lg for Tr in $Tr(iC, FM)$. To check that a candidate event in the log is a cause for a deviation in $Dev(EvL, \mathcal{P})$ we do a *gedanken experiment*. That is we find a trace Tr' in $Tr(iC, FM)$ that agrees with Tr up to the event, avoids the identified event an continues maintaining maximal agreement with Tr. The idea is inspired by counterfactual reasoning.

```
ceq gedanken(asys,asysB,n,j) = log1
if  log := genLog(asys, ((s n)  * 3))  --- 3 rew per time unit
/\ litem := subLog(log,n,1)            --- last trace element
/\ ekb := getLIEnvKB(litem)            --- current environment KB
/\ aconf := getLIAgents(litem)         --- current agent state
/\ asys1 := updateSys(asysB,s n, ekb,aconf) ---
/\ log1 := genLog(asys1,j)             --- hoping for missing deviation
.
```

asys is the system (with faults) generating Tr and asysB contains the base configuration information needed to continue avoiding the candidate event. The n is the time just before the candidate event, and j is how long to continue to check that the deviation is gone. asys1 augments the logged environment and agent state with knowledge that is constant, such as fence boundary or resource location knowledge, that are not logged, but needed to execute the rules. The logged environment knowledge base does not contain fault models, but may contain obstacles and these must be removed for the gedanken experiment. Note that for the purpose of gedanken the log needs to collect all of the agent state that changes and that is used to decide actions. It also needs to collect all environment knowledge of changing information that is needed for sensor readings or to compute the effects of actions. This may extend what is collected just for determining protocol satisfaction.

7.5 Putting it all together

We provide scripts to orchestrate case studies and visualize the results. We explain the scripts using examples from the BotTeam case study.

```
 1: maude -no-banner -no-wrap < initS1alocsf1x.maude > initS1alocsf1x.txt

 2: maude -no-banner -no-wrap < initS1alocsf1xEvents.maude > initS1alocsf1xEvents.txt
 3: ./run_loctreat.sh initS1alocsf1xEvents.txt initS1alocsf1xEventsEnv.txt \
        initS1alocsf1xEventsBot.txt data_trajectory.csv
 4: Rscript --vanilla trajectory.R  data_trajectory.csv
 5: mv Rplots.pdf botteam_initS1alocsf1x_trajectory.pdf

 6: maude -no-banner -no-wrap < initS1alocsf1xEventsX.maude > initS1alocsf1xEventsX.txt
 7: ./run_energy.sh initS1alocsf1xEventsX.txt initS1alocsf1xEventsXEnv.txt \
        initS1alocsf1xEventsXBot.txt data_energy.csv
 8: Rscript --vanilla energy.R  data_energy.csv
 9: mv Rplots.pdf botteam_initS1alocsf1x_energy.pdf

10: maude -no-banner -no-wrap < initS1alocsf1xEDevsL.maude > initS1alocsf1xEDevs.txt
```

Line 1 of the above script takes the input file `initS1alocsf1x.maude` to load the initial configuration for Patrol Bots (via the `load` command) and executes the model (via `rewrite` command) with two treat locations (i.e., `pt(4,0), pt(5,4))` in the presence of a `b(1)` location sensor fault, `locsf(1)`, as shown below.

```
load ../../Models/BotTeam/load1
rew [175]  addFaultsR(mkInitS(pt(4, 0) ; pt(5, 4), true), locsf(1), 500) .
```

The function `mkInitS` adds an empty log configuration element to the initial configuration, so a log will be collected during execution. The result of rewriting is output to `initS1alocsf1x.txt` as you see below.

```
Maude> ==========================================
rewrite [175] in SCENARIO-DIAGNOSIS : addFaultsR(mkInitS(pt(4, 0) ; pt(5, 4), true),
        locsf(1), 500) .
rewrites: 107811 in 100ms cpu (104ms real) (1070509 rewrites/second)
result ASystem: {
[eI |
  clock(58) comDist(12) randInc(3) class(b(0), BotI) class(b(1), BotF)
  class(st(0), Station) class(dp(0), Depot) (atloc(b(0), pt(5, 4)) @ 22)
  (atloc(b(1), pt(3, 1)) @ 24) (atloc(st(0), pt(3, 2)) @ 0) (atloc(dp(0), pt(3, 3)) @ 0)
  (energy(b(0), 80) @ 22) (energy(b(1), 10) @ 24) (supply(b(1), 60) @ 19)
  (treatStage(pt(4, 0), 4) @ 19) (treatStage(pt(5, 4), 0) @ 0)
  (fence(b(0), pt(0, 0), pt(6, 5)) @ 0) (fence(b(1), pt(0, 0), pt(6, 5)) @ 0)
  (rand(eI, 2, 500) @ 58) (rand(b(0), 0, 500) @ 0) (rand(b(1), 1, 616) @ 57)
  sF(b(1), locS, simpleFT, sFP(1/10, 1/5, ptV(pt(0, 1)))) @ 0] (
{0,
  {{cv(100),u(1)},mv(b(0), pt(1, 0))},
[eI |
  (atloc(b(0), pt(1, 1)) @ 1) (atloc(b(1), pt(0, 4)) @ 0) (energy(b(0), 95) @ 1)
  (energy(b(1), 100) @ 0) supply(b(1), 100) @ 0]
[b(0) : BotI |
  lkb : ((targets(pt(5, 4)) @ 0) (target(pt(4, 0)) @ 0) (atloc(b(0), pt(0, 1)) @ 0)
  (energy(b(0), 100) @ 0) mode(b(0), enroute(pt(4, 0))) @ 0)]
[b(1) : BotF |
  lkb : ((atloc(b(1), pt(0, 4)) @ 0) (energy(b(1), 100) @ 0) (mode(b(1), ready) @ 0)
  supply(b(1), 100) @ 0)]}
 ;
....
```

```
 {57,
  none,
[eI |
  (atloc(b(0), pt(5, 4)) @ 22) (atloc(b(1), pt(3, 1)) @ 24) (energy(b(0), 80) @ 22)
  (energy(b(1), 10) @ 24) (supply(b(1), 60) @ 19) treatStage(pt(5, 4), 0) @ 0]
[b(0) : BotI |
  lkb : ((targets(nil) @ 12) (target(pt(5, 4)) @ 12) (atloc(b(0), pt(5, 4)) @ 22)
  (atloc(b(1), pt(3, 2)) @ 53) (energy(b(0), 80) @ 22) (mode(b(0), treat) @ 57)
  treatStage(pt(5, 4), 0) @ 0)]
[b(1) : BotF |
  lkb : ((targets(nil) @ 20) (target(pt(5, 4)) @ 12) (targetF(pt(5, 4)) @ 20) (
  atloc(b(0), pt(5, 4)) @ 22) (atloc(b(1), noLoc) @ 54) (energy(b(1), 10) @ 24)
  (mode(b(1), enroute(pt(5, 4))) @ 57) (supply(b(1), 60) @ 19)
  treatStage(pt(5, 4), 0) @ 0)]}
)
[b(0) : BotI |
  lkb : (clock(58) class(b(0), BotI) class(b(1), BotF) class(st(0), Station)
  (lastAct(mv(b(0), pt(0, 1))) @ 21) (targets(nil) @ 12) (target(pt(5, 4)) @ 12)
  (atloc(b(0), pt(5, 4)) @ 22) (atloc(b(1), pt(3, 2)) @ 53) (atloc(st(0), pt(3, 2)) @ 0)
  (energy(b(0), 80) @ 22) (ereserve(b(0), 35) @ 0) (mode(b(0), treat) @ 58)
  (treatStage(pt(4, 0), 2) @ 11) (treatStage(pt(5, 4), 0) @ 0) (treatRisk(b(0), true) @ 0)
  (fence(b(0), pt(0, 0), pt(6, 5)) @ 0) home(b(0), pt(0, 1))),
  ckb : (class(b(0), BotI) class(b(1), BotF) (target(pt(5, 4)) @ 12)
  (atloc(b(0), pt(5, 4)) @ 22) atloc(b(1), noLoc) @ 54),
  sensors : (locS obstacleS energyS treatS),
  evs : (tick @ 1)]
[b(1) : BotF |
  lkb : (clock(58) class(b(0), BotI) class(b(1), BotF) class(st(0), Station)
  class(dp(0), Depot) (lastAct(charge(b(1))) @ 53) (targets(nil) @ 20)
  (target(pt(5, 4)) @ 12) (targetF(pt(5, 4)) @ 20) (atloc(b(0), pt(5, 4)) @ 22)
  (atloc(b(1), noLoc) @ 54) (atloc(st(0), pt(3, 2)) @ 0) (atloc(dp(0), pt(3, 3)) @ 0)
  (energy(b(1), 10) @ 24) (ereserve(b(1), 35) @ 0) (mode(b(1), enroute(pt(5, 4))) @ 57)
  (supply(b(1), 60) @ 19) (treatStage(pt(4, 0), 4) @ 19) (treatStage(pt(5, 4), 0) @ 0)
  (treatRisk(b(1), true) @ 0) (fence(b(1), pt(0, 0), pt(6, 5)) @ 0) home(b(1), pt(0, 4))),
  ckb : (class(b(0), BotI) class(b(1), BotF) (target(pt(5, 4)) @ 12)
  (atloc(b(0), pt(5, 4)) @ 22) atloc(b(1), noLoc) @ 54),
  sensors : (locS obstacleS energyS treatS supplyS),
  evs : (rcv(none) tick @ 0)]}
Maude> Bye.
```

Next, line 2 of the script generates events for checking protocol deviations as described in Section 7.2. The input file (i.e., `initS1alocsf1xEvents.maude`) below loads the required model, scenario, and configuration definitions and evaluates `getLocTreatEvs` (via `red` short for `reduce`) to extract the events of the BotTeam protocol language (e.g., `atloc`, `treatStage`). The function `getLocTreatEvs` uses `genLog` (7.1) to generate a log for the given system configuration using 180 rewrites. It then uses `flatLog` to extract the full environment and agent event logs and from these it extracts the `atloc`, `treatStage` events.

```
load ../../Models/BotTeam/load1
red getLocTreatEvs(addFaultsR(mkInitS(( pt(4,0) ; pt(5,4) ),true), locsf(1), 500), 180) .
```

The output file (i.e., `initS1alocsf1xEvents.txt`) below shows the events (extracted from both `env` and `bot` KBs).

```
reduce in SCENARIO-DIAGNOSIS :
  getLocTreatEvs(addFaultsR(mkInitS(pt(4, 0) ; pt(5, 4), true), locsf(1), 500), 180) .
rewrites: 111822 in 166ms cpu (168ms real) (672884 rewrites/second)
result KBPair: {
(atloc(b(0), pt(1, 1)) @ 1) (atloc(b(0), pt(2, 1)) @ 2) (atloc(b(0), pt(3, 0)) @ 13)
```

```
(atloc(b(0), pt(3, 1)) @ 3) (atloc(b(0), pt(3, 1)) @ 14) (atloc(b(0), pt(3, 2)) @ 15)
(atloc(b(0), pt(4, 0)) @ 5) (atloc(b(0), pt(4, 1)) @ 4) (atloc(b(0), pt(4, 2)) @ 19)
(atloc(b(0), pt(5, 2)) @ 20) (atloc(b(0), pt(5, 3)) @ 21) (atloc(b(0), pt(5, 4)) @ 22)
(atloc(b(1), pt(0, 4)) @ 0) (atloc(b(1), pt(1, 4)) @ 2) (atloc(b(1), pt(2, 4)) @ 3)
(atloc(b(1), pt(3, 1)) @ 24) (atloc(b(1), pt(3, 2)) @ 23) (atloc(b(1), pt(3, 4)) @ 4)
(atloc(b(1), pt(4, 0)) @ 15) (atloc(b(1), pt(4, 0)) @ 17) (atloc(b(1), pt(4, 1)) @ 9)
(atloc(b(1), pt(4, 1)) @ 12) (atloc(b(1), pt(4, 1)) @ 21) (atloc(b(1), pt(4, 2)) @ 8)
(atloc(b(1), pt(4, 2)) @ 22) (atloc(b(1), pt(4, 3)) @ 6) (atloc(b(1), pt(4, 4)) @ 5)
(treatStage(pt(4, 0), 0) @ 0) (treatStage(pt(4, 0), 1) @ 10)
(treatStage(pt(4, 0), 2) @ 11) (treatStage(pt(4, 0), 3) @ 18)
(treatStage(pt(4, 0), 4) @ 19) treatStage(pt(5, 4), 0) @ 0
,
(atloc(b(0), pt(0, 1)) @ 0) (atloc(b(0), pt(1, 1)) @ 1) (atloc(b(0), pt(2, 1)) @ 2)
(atloc(b(0), pt(3, 0)) @ 13) (atloc(b(0), pt(3, 1)) @ 3) (atloc(b(0), pt(3, 1)) @ 14)
(atloc(b(0), pt(3, 2)) @ 15) (atloc(b(0), pt(4, 0)) @ 5) (atloc(b(0), pt(4, 1)) @ 4)
(atloc(b(0), pt(4, 2)) @ 19) (atloc(b(0), pt(5, 2)) @ 20) (atloc(b(0), pt(5, 3)) @ 21)
(atloc(b(0), pt(5, 4)) @ 22) (atloc(b(1), pt(0, 4)) @ 0) (atloc(b(1), pt(1, 4)) @ 2)
(atloc(b(1), pt(2, 4)) @ 3) (atloc(b(1), pt(3, 2)) @ 30) (atloc(b(1), pt(3, 2)) @ 38)
(atloc(b(1), pt(3, 2)) @ 52) (atloc(b(1), pt(3, 2)) @ 53) (atloc(b(1), pt(3, 3)) @ 23)
(atloc(b(1), pt(3, 4)) @ 4) (atloc(b(1), pt(4, 0)) @ 17) (atloc(b(1), pt(4, 1)) @ 9)
(atloc(b(1), pt(4, 1)) @ 12) (atloc(b(1), pt(4, 1)) @ 16) (atloc(b(1), pt(4, 1)) @ 21)
(atloc(b(1), pt(4, 2)) @ 8) (atloc(b(1), pt(4, 2)) @ 11) (atloc(b(1), pt(4, 2)) @ 22)
(atloc(b(1), pt(4, 4)) @ 5) (atloc(b(1), pt(4, 4)) @ 7)
(treatStage(pt(4, 0), 0) @ 0) (treatStage(pt(4, 0), 1) @ 10)
(treatStage(pt(4, 0), 2) @ 11) (treatStage(pt(4, 0), 3) @ 18)
(treatStage(pt(4, 0), 4) @ 19) treatStage(pt(5, 4), 0) @ 0
}
```

Line 3 of the script takes the above event file and outputs the environment and the agent events to separate files `initS1alocsf1xEventsEnv.txt` and `initS1alocsf1xEventsBot.txt`. The command `run_loctreat.sh` generates the input file for `R` visualization (i.e., `data_trajectory.csv` which has comma separated line format: `bot, x-coordinate, y-coordinate, timestamp` for `atLoc` and `treatStage` events. Lines 4-5 visualize each bot's trajectory and treatment stages as depicted in Figure 2(left) using `RScript`. Lines 6-9 repeat similar processing for `energy` events, which depicts each bot's energy levels as you see in Figure 2(right).

Line 10 generates list of execution deviations. The input file (i.e., `initS1alocsf1xEDevsL.maude`) below loads the required model, scenario, and configuration definitions

```
load ../../Models/BotTeam/load1
red getEDevsL(addFaultsR(mkInitS(pt(4, 0) ; pt(5, 4), true), locsf(1), 500),
        bot0base bot1base, 175,0,175) .
```

and evaluates `getEDevsL` to generate the list of execution deviations (shown below) and output it in the file `initS1alocsf1xEDevs.txt`.

```
Maude> ========================================
reduce in SCENARIO-DIAGNOSIS : getEDevsL(addFaultsR(mkInitS(pt(4, 0) ; pt(5, 4), true),
  locsf(1), 500), bot0base bot1base, 175, 0, 175) .
rewrites: 118854 in 135ms cpu (140ms real) (880165 rewrites/second)
result EDevsList:
[6,
{{(zero).RVal,noAct},
(targets(pt(5, 4)) @ 1) (target(pt(4, 0)) @ 0) (targetF(pt(4, 0)) @ 1)
  (atloc(b(0), pt(4, 0)) @ 5) (energy(b(1), 75) @ 6) (mode(b(1), enroute(pt(4, 0))) @ 6)
  supply(b(1), 100) @ 0,
atloc(b(1), noLoc) @ 6,
atloc(b(1), pt(4, 3)) @ 6}
]
```

```
 ;
[7,
{{{cv(100),u(1)},mv(b(1), pt(0, -1))},
(targets(pt(5, 4)) @ 1) (target(pt(4, 0)) @ 0) (targetF(pt(4, 0)) @ 1)
  (atloc(b(0), pt(4, 0)) @ 5) (atloc(b(1), pt(4, 4)) @ 7) (energy(b(1), 75) @ 6)
  (mode(b(1), enroute(pt(4, 0))) @ 7) supply(b(1), 100) @ 0,
(atloc(b(1), pt(4, 3)) @ 8) energy(b(1), 70) @ 8,
atloc(b(1), pt(4, 2)) @ 8}
]
 ;
 .....
[57,
{{(zero).RVal,noAct},
(targets(nil) @ 20) (target(pt(5, 4)) @ 12) (targetF(pt(5, 4)) @ 20)
  (atloc(b(0), pt(5, 4)) @ 22) (energy(b(1), 10) @ 24)
  (mode(b(1), enroute(pt(5, 4))) @ 57) (supply(b(1), 60) @ 19)
  treatStage(pt(5, 4), 0) @ 0,
atloc(b(1), noLoc) @ 54,
atloc(b(1), pt(3, 1)) @ 24}
]

Maude> Bye.
```

We see that at time `6` bot `b(1)` had a broken location sensor (sensor reading `noLoc`) and at time `7`, bot `b(1)` thought he was at `pt(4,4)` moving to `pt(4,3)`. While the actual location after the move is `pt(4,2)`. This deviation could be a location error at time `7` or a move action error. Given the previous location error, one might prioritize the first possibility.

Section 7.2 shows the results of checking protocol satisfaction using the `deviate` tool, and the protocol shown in section 5.1.

8 Summary of Experiments

In this section we summarize the results of diagnosis for executions of the BotTeam case study, presented in Section 5, using two and three target locations and a variety of fault models. We also briefly summarize results of executing a two drone scenario with fault models of increasing probability.

8.1 Summary of BotTeam Execution results

The scenario for this case study has two bots on a 7×6 grid: `b(0)`, the initiator (class `BotI`), with sensors `locS energyS obstacleS treatS`, initially at `pt(0,1)`; and `b(1)`, the finisher (class `BotF`), with sensors `locS energyS obstacleS treatS supplyS`, initially at `pt(0,4)`. There is a charging station at `pt(3,2)` and a supply depot at `pt(3,3)`. Experiments were done for two target lists with initial system configurations `initS1a` (2 targets) and `initS2` (3 targets).

Fault models. The fault models used in the experiments are described in the following two tables. The first column is the name used for the fault model in the corresponding experiment summary table. The second column is the knowledge item(s) added to the

environment knowledge base. The annotation `rand 500` means the random sequences are shifted by `500`.

```
InitS1a  loclist  pt(4,0) ; pt(5,4)
  ensfm1  sF(b(1),energyS,simpleFT,sFP(1/10,1/5,intV(-10))) @ 0 rand 500
  locsf1x sF(b(1),locS,simpleFT,sFP(1/10,1/5,ptV(pt(0,1)))) @ 0 rand 500
  mvaf1   aF(b(1),mvA,simpleFT,sFP(1/10,1/5,ptV(pt(0,-1)))) @ 0
  mvaf1x  aF(b(1),mvA,simpleFT,sFP(1/10,1/5,ptV(pt(0,-1)))) @ 0 rand 500
  Obs4-2  class(ob(0), Obstacle) atloc(ob(0), pt(4, 2)) @ 0

InitS2 loclist  pt(5,1) ; pt(2,0) ; pt(1,4)
  ensf0   sF(b(0),energyS,simpleFT,sFP(1/10,1/5,intV(10))) @ 0  rand 500
  ensf1   sF(b(1),energyS,simpleFT,sFP(1/10,1/5,intV(10))) @ 0
  locsf0x sF(b(0),locS,simpleFT,sFP(1/10,1/5,ptV(pt(0,1)))) @ 0 rand 500
  locsf1  sF(b(1),locS,simpleFT,sFP(1/10,1/5,ptV(pt(0,1)))) @ 0 rand 500
  mvaf1   aF(b(1),mvA,simpleFT,sFP(1/10,1/5,ptV(pt(0,-1)))) @ 0
```

Table 1 summarizes the results of experiments using the two target scenario `initS1a` using the SA protocol shown in section 5. Table 2 summarizes the results of experiments using the three target scenario `initS2` using the three target analog of the SA protocol shown in section 5. The Faults column names the fault model used (as defined above). The Close column specifies how close the `BotI` bot requires the `BotF` bot to be before beginning treatment at a target location. For the experiments the possible values are 1 and 3 distance units. The PDevs column lists the minimal (with respect to the protocol partial order) protocol deviation(s) if any. The EDevs column gives the number of execution deviations prior to the protocol deviation. In the Gedanken column a +/- indicates that the gedanken experiment defined by the candidat execution deviation causes confirmed/failed to confirm that elimination of the deviation events eliminated the protocol deviation. na abbreviates "not applicable". We use the following notation for protocol deviations. `delta(x,y)` abbreviates: the log events have

```
(treatStage(pt(x,y),2) @ t0) (treatStage(pt(x,y),4) @ t1)
```

with `t1 - t0 >= 10` violating the constraint on delay between initial and final treatments. `delta((x0,y0),(x1,y1))` abbreviates: the log events have

```
(treatStage(pt(x1,y1),2) @ t0) (tloc(b(0),pt(x0,y0)) @ t1)
```

with `t1 - t0 >= 15` violating the constraint on time between target locations. `atloc(b(j),pt(x,y))` indicates a missing event.

Table 1: initS1a targets: pt(4,0) ; pt(5,4)

Faults	Close	PDevs	Edevs[0]	Gedanken
none	3,1	none	none	na
ensfm1	3,1	none	6	na
locsf1x	3	delta(5,4)	6	+
	1	atloc(b(1),pt(5,4))	many	+
mvaf1	3,1	none	10	na
mvaf1x	3,1	none	8	na
Obs4-2	3,1	atloc(b(1),pt(5,4))	0[1]	+

Notes. [0] If PDevs is none then EDevs counts deviations in the full execution. [1] Execution deviations due to obstacles are not currently collected. In this experiment, removing the obstacle at event before deviation corrects the behavior.

Table 2: initS2 targets: pt(5,1) ; pt(2,0) ; pt(1,4)

Faults	Close	PDevs	Edevs	Gedanken
none	3,1	none	none	na
ensf0x	3,1	delta((1,4),(2,0)	2,1	+
ensf1	3	delta(2,0)	5	+
	1	none	11	na
locsf0x	3	delta(1,4)	6	- [2]
	1	treatStage(pt(2,0),2)	1	+
locsf1x	3	delta(5,1)	8	+
	3,1	atloc(b(1),pt(2,0))	1	+
mvaf1	3,1	delta(2,0)	1	+
	3,1	delta((2,0),(1,4))		+

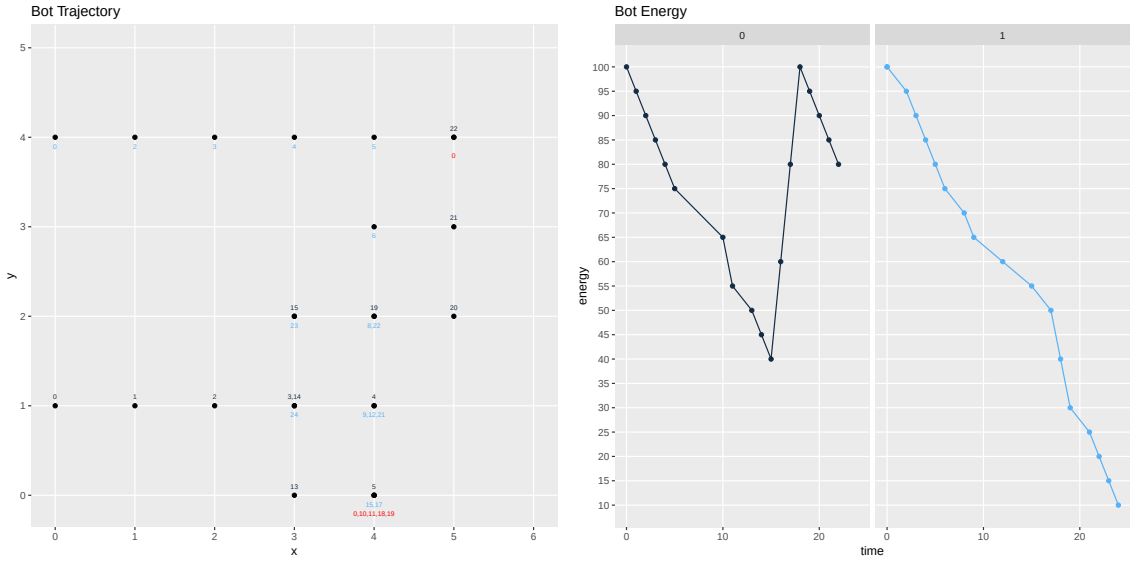
Notes. [2] `b(1)` gets stuck at `pt(1,3)` going to `pt(3,3)`.

Experiment steps. For each possibly faulty scenario, the initial configuration was executed to produce a log. Protocol events were then extracted from the log and checked for deviations using `deviate`. If deviations were found, minimal deviations with respect to the protocol partial order were selected, and maximal subprotocols satisfied found. We then looked for early execution deviations following events in these subprotocols and carried out the gedanken experiments using `gedanken` to check if eliminating the execution deviations eliminated the protocol deviations, i.e., are the identified execution deviations possible causes of the protocol deviations.

From tables 1 and 2 we see that in all but one case this process was able to identify execution deviations which, when removed, eliminated the protocol deviation. The exceptional case involved the three target scenario with location sensor faults for `BotI`. The deviation

is an event for which `BotF` is responsible. The actual problem is that `BotF` passes through a location adjacent to the target location on the way to the supply depot. It gets stuck there waiting for `BotI` to finish its treatment, because the constraint system gives 0 preference to moving if it is adjacent to its target and `BotI` is there treating. For the diagnosis system to understand this, more information is needed about the constraints underlying the bots' action decisions.

Drilling down. We described how to use `deviate` to check for protocol deviations in Section 7.2 and described scripts to produce necessary logs and to visualize the execution from those logs in Section 7.5. Here, for the BotTeam case study scenario with location sensor fault for bot `b(1)` (i.e., `initS1a locsf1x`), we describe the diagnosis process in some detail.



(a) (b)

Figure 2: Bot trajectory and energy level for the BotTeam case study scenario with two target locations and a location sensor fault for the `BotF` bot (b(1)) `initS1a locsf1x`. On the left, the grid of size 7×6 is annotated with the timestamp when each bot's `atloc` event occurs in black and blue colors for `b(0)` and `b(1)`, respectively. The `treatStage` events are annotated with red color. On the right each bot's energy level is depicted based on the `energy` events.

Figure 2 was generated by using scripts described in Section 7.5. As shown in Section 7.2, `deviate` provides the frontier information

```
Frontier: set([(0, 2), (4, 0)])
```

and missing events with two unsat cores,

```
(treatStage (pt 5 4) 2 s04)  (atloc (b 1) (pt 5 4) s12)
```

which indicate that the first treat location (i.e., `pt(4, 0)`) has been treated by both bots while the second treat location (i.e., `pt(5, 4)` can not been treated. The unsat core also indicates that `b(1)` has never arrived at the second treat location as you can also see from the trajectory of bots in Figure 2(left).

The execution deviations (i.e., `initS1alocsf1xEDevs.txt`) can be investigated for further information. There is a location sensor deviation with sensor reading `(atloc(b(1), pt(3, 3)) @ 23)` while the actual location was `pt(3,2)`. This lead `b(1)` to move towards south instead of charging. This was followed by no readings (i.e., `noLoc` as its value) for 6 time units. At time `30`, another `locationS` fault `(atloc(b(1), pt(3, 2)) @ 30)` off by `(0,-1)` makes `b(1)` attempt to charge (at `pt(3,1)` where there is no charging station) during times `30-42` rather than moving to the charging station. Similar situations repeat, preventing `b(1)` from charging and moving towards the second treat location. Figure 2(right) also shows that the energy level of `b(1)` runs low (i.e., under the given `ereserve` value of `35`) from timestamp `19` and onwards since the faulty location sensor readings result to wrong decisions such as moving away from the charging station or trying to charge at the wrong location as explained above. Doing the gedanken experiment that eliminates the location faults at 23-30 shows that bot `b(1)` now succeeds in reaching `pt(5,4)` and eliminating the protocol deviations.

8.2 Summary of Drone Execution Results

The third case study models autonomous drones doing surveillance. In contrast to the ground bots, in our drone model, there are no charging stations. A drone heads home when it runs low on energy. Thus a drone should not crash due to running out of energy and it is bad to land far from home (but better than crashing). The model was tested under a range of location sensor and motion faults to determine when faults caused deviations and the tradeoff between caution and achieving surveillance goals. Specifically, we focused on effects of increasing probability of faults on the ability of a drone to complete their tasks (visit all points) and return home safely. We also looked at the ability of a drone to estimate its time of arrival at a selected location.

The scenarios consisted of two drones on a 100×100 grid, each with two locations to visit. Drone `b(0)` starts at its home, `pt(2,2,0)`, and visits `pt(10,80,25)` and `pt(80,10,25)`. Drone `b(1)` starts at its home, `pt(98,98,0)`, and visits `pt(90,20,35)` and `pt(20,90,35)`. We summarize results for move action (`gtooA`) and location sensor (`locS`) fault models shown below.

```
aF(b(0),goToA,simpleFT,sFP(1/12,1/12,ffV(1)))
aF(b(0),goToA,simpleFT,sFP(1/6,1/6,ffV(2)))
aF(b(0),goToA,simpleFT,sFP(1/4,1/4,ffV(3)))

sF(b(0),locS,simpleFT,sFP(1/15,1/15,ffV(1)))
sF(b(0),locS,simpleFT,sFP(2/15,2/15,ffV(2)))
sF(b(0),locS,simpleFT,sFP(1/5,1/5,ffV(3)))
```

The interpretation of fault models for drones is similar to that for bots. The first two parameters are the probability of a broken sensor/actuator and the probability of error, respectively. In the case of error a displacement vector is chosen randomly and scaled by the

third parameter (`ffV(n)`). Since the two drones are independent and equivalent, we only applied faults to one of the drones (`b(0)`).

With no external faults the drones succeed in visiting the two points and landing at home with a small amount of energy left. The drones time estimates are within 3 time units of actual times. In the presence of a location sensor fault of increasing probability the drone manages to visit all points and return close to home with a small amount of energy left. The time it takes to do this increases with increasing fault probability. The estimated arrival time is less accurate for the first point visited. In the presence of a move action fault of increasing probability the drone manages to visit all points but it lands further from home as the probability of fault increases.

9 Next Steps

In the work reported here we focused on a global view of diagnostics working with traces collected by some imaginary agent with global a global view. Some of the diagnosis is systematic but still manual. More automation is needed. The real challenge is to distribute the monitoring and analysis to the agents in the system.

Collecting the agent part of the log can be done by simply adding log knowledge to each agents local KB. Agents already record the last action requested. Thus the local deviation detection code can be run by the (currently implicit) monitoring component of `doTask`. Presence of obstacles currently is treated specially. The collection of execution deviations should be extended to report effects of encountering an obstacle. The `remember` component of `doTask` can accumulate a local projection of the log. The `tell` component can include the latest local log item. The `handleS` component can be extended to incrementally build a view of the global agent component of the log. The environment component of the current log is not going to be part of the internalized detection and analysis. For critical observations, agents should be equipped with multiple sensors / multiple independent ways of measuring, to reduce the chance of having a faulty model of the world. They can also use their model of the world and combined knowledge to check sensor readings.

Once local agents collect their own views of an execution log, they can in principle run essentially the same detection and diagnostics code locally and incrementally, leveraging the incremental protocol satisfaction checks provided by `deviate`. Results will reflect the collected information so the timeliness will depend on communication connectedness and frequency of agent sharing information.

The case studies should be extended to include examples with more coordinate between agents and more agent awareness of other agents. For example an agent may take into account the timestamp of shared information, if information is too old it might not be reliable. For some application areas such as drones and bots with tools, it is interesting to explore learning models of the effects of actions on the fly.

We also need to abstract key elements of an agents decision process, for example as reasons why some invariant or other constraint should hold. A step in this direction is to formalize the automata for higher level planning as well as formally connecting protocol invariants to aspects of the constraint semi-ring and constraint solving rules.

Another important future extension is consideration of security issues. Attacks on sen-

sors and actuators are in some sense incorporated in the fault models, but work is needed to develop realistic models where mechanical/electrical faults are likely to have different profiles than the results of malicious attacks. Information sharing is currently assumed to be secure in the sense that the information received is what was sent and that sharing eventually happens if agents come close enough to each other. A big question is what is the right threat model, and how can agents feasibly protect themselves (and their collaborators)?

References

[1] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[2] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[3] Tobias Kappé. Logic for soft component automata. Master's thesis, Leiden University, Leiden, The Netherlands, 2016.

[4] Tobias Kappé, Farhad Arbab, and Carolyn L. Talcott. A compositional framework for preference-aware agents. In *Proc. Workshop on Verification and Validation of Cyber-Physical Systems (V2CPS)*, pages 21–35, 2016.

[5] Tobias Kappé, Farhad Arbab, and Carolyn L. Talcott. A component-oriented framework for autonomous agents. In José Proença and Markus Lumpe, editors, *Formal Aspects of Component Software*, volume 10487 of *LNCS*, pages 20–38. Springer, 2017.

[6] Ian A. Mason, Vivek Nigam, Carolyn Talcott, and Alisson Brito. A framework for analyzing adaptive autonomous aerial vehicles. In *1st Workshop on Formal Co-Simulation of Cyber-Physical Systems*, 2017.

[7] The maude system. last accessed: 2018-12-15.

[8] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[9] Carolyn Talcott. From soft agents to soft component automata and back. In *The Joy of Coordination*, LNCS. Springer, 2018.

[10] Carolyn Talcott, Farhad Arbab, and Maneesh Yadav. Soft agents: Exploring soft constraints to model robust adaptive distributed cyber-physical agent systems. In *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*, volume 8950 of *LNCS*. Springer, 2015.

[11] Carolyn Talcott, Vivek Nigam, Farhad Arbab, and Tobia Kappe. Formal specification and analysis of robust adaptive distributed cyber-physical systems. In *SFM 2016: Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems*, volume 9700 of *LNCS*, pages 1—-35. Springer, 2016.