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

Carolyn Talcott[1(✉)], Vivek Nigam[2], Farhad Arbab[3,4], and Tobias Kappé[3,4]

[1] SRI International, Menlo Park, CA 94025, USA
carolyn.talcott@sri.com
[2] Federal University of Paraiba, João Pessoa, Brazil
vivek.nigam@gmail.com
[3] LIACS, Leiden University, Leiden, The Netherlands
tkappe@liacs.nl
[4] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
farhad@cwi.nl

Abstract. We are interested in systems of cyber-physical agents that operate in unpredictable, possibly hostile, environments using locally obtainable information. How can we specify robust agents that are able to operate alone and/or in cooperation with other agents? What properties are important? How can they be verified?

In this tutorial we describe a framework called Soft Agents, formalized in the Maude rewriting logic system. Features of the framework include: explicit representation of the physical state as well as the cyber perception of this state; robust communication via sharing of partially ordered knowledge, and robust behavior based on soft constraints. Using Maude functionality, the soft agent framework supports experimenting with, formally testing, and reasoning about specifications of agent systems.

The tutorial begins with a discussion of desiderata for soft agent models. Use of the soft agent framework for specification and formal analysis of agent systems illustrated in some detail by a case-study involving simple patrolling bots. A more complex case study involving surveillance drones is also discussed.

1 Introduction

Consider a future in which an explosion of small applications running on mobile devices combine and collaborate to provide powerful new functionality, not only in the realms such as smart vehicles, disaster response, home care, but also harnessing diverse communication mechanisms and robust people power for new kinds of cyber crowd sourcing tasks.

Complex cyber-physical agents are becoming increasingly ubiquitous, in part, due to increased computational performance of commodity hardware and the widespread adoption of standard mobile computing environments (e.g. Android).

The work was partially supported by ONR grant N00014–15–1–2202.

M. Bernardo et al. (Eds.): SFM 2016, LNCS 9700, pp. 1–35, 2016.
DOI: 10.1007/978-3-319-34096-8_1

For example, one can purchase (for a few hundred US dollars in the retail market) a four rotor "drone" capable of precise, controlled hovering that can be equipped with a portable Android phone that provides integrated communication (e.g. wifi ad hoc) and sensing (e.g. high resolution camera) capability as well as considerable processing power (e.g. multiple GPU cores) and memory.

Already there are impressive examples coming from both research and industrial settings. Here are a few.

Researchers in the Vijay Kumar Lab at the University of Pennsylvania [1–3] are experimenting with autonomous flying quadrotor robots that can be equipped with a variety of sensors such as IMUs (inertial measurement units), cameras, laser range scanners, altimeters and/or GPS sensors. Key capabilities being developed include navigation in 3-dimensional space, sensing other entities, and forming ad hoc teams. Potential applications include construction, search and rescue, first response, and precision farming.

The recent EU ASCENS project [4,5] focused on theoretical foundations and models for reliable and predictable system behavior while exploiting the possibilities of highly dynamic, autonomic components. The project included pragmatic case studies such as generation of a robot swarm with both autonomous and collective behavior.

A case study of an adaptive network consisting of smart phones, robots, and UAVs is reported in [6]. The temporal evolution of the macroscopic system state is controlled using a distributed logic [7,8], while the microscopic state is controlled by an algorithm based on 'artificial physics'. Communication is based on a partially-ordered knowledge sharing model for loosely coupled distributed computing. The ideas were tested via both simulation and in a parking lot using a cyber-physical testbed consisting of robots, quadcopters, and Android devices. This work was part of the SRI Networked Cyber-Physical Systems project [9].

Startups are developing applications for shipping, security, search and rescue, environmental monitoring, and agriculture to name a few. Of course, there are Google's driverless cars and Amazon's drone delivery system.

In October 2015 Stanford hosted *Drone swarms: the Buz of the future* [10]. The event included demonstrations (live and video) of a number of advanced autonomous robotic capabilities. The Knightscope [11] security robots roamed the busy plaza without bumping into people or other objects. Their normal job is surveillance of limited areas, looking for problems. Liquid Robotics [12] presented their *wave glider*, a surfboard size robot that is powered by the ocean, capable of multiple modes of communication and of carrying diverse sensors. Wave gliders are able to autonomously and safely navigate from the US to Australia, and able to call home when pirates try bot-napping gliders, in addition to collecting data.

According to a recent Fortune article [13] Burlington Northern Santa Fe (BNSF) Railway has gained FAA approval for a pilot(less) program to use drones to inspect its far-flung network of rails, over 32,500 miles. There are many technical obstacles to overcome, but use of drones has the potential for both greater efficiency and improved safety. Each rail section is inspected at least twice a week, and inspectors may have to deal with disagreeable insects, toxic vegetation, or poisonous snakes.

Formal executable models can provide valuable tools for exploring system designs, testing ideas, and verifying aspects of a systems expected behavior. Executable models are often cheaper and faster to build and experiment with than physical models, especially in early stages as ideas are developing. The value of using formal executable models in the process of designing and deploying a network defense system is illustrated in [14,32].

In the following we describe a *Soft Agent Framework* that provides infrastructure and guidance for building formal executable models of cyber-physical agent systems that we call *soft agent* systems. We present an executable specification in the Maude rewriting logic system [15] and illustrate the ideas with case studies.

Key features of soft agents include

- Simple goal specification (package A must get delivered from point X to point Y).
- Efficient (not necessarily optimal) agent trajectory planning.
- Distributed and local, there is no central planner with perfect global knowledge.
- Agents are resource constrained—limited communication range, energy (actions consume energy), lift power, etc.
- Robustness to unexpected/unplanned situations
  - agent actions (sensor readings, actuator actions, communication) can suffer delays/failure,
  - perturbations in environment. e.g., a gust of wind, a change of goal.

The soft agent framework combines ideas from several previous works: the use of soft constraints [16–18] and soft constraint automata [19] for specifying robust and adaptive behavior; partially ordered knowledge sharing for communication in disrupted environments [8,20–22], and the Real-time Maude approach to modeling timed systems [23]. A novel feature is the explicit representation of both cyber and physical aspects of a soft agent system. An environment component maintains the physical state of the agents and their surroundings. Each agent has a local knowledge base that stores the agents view of the world, which may or may not be the actual state of the world. To specify an agent system one must say what sensors an agent has and what actions it can perform. The 'physics' of these actions must also be specified: how an action affects the state of the agent and its surroundings (in particular how the sensor readings change). One must also specify how an agent decides what actions to carry out in different situations. Although the framework allows complete freedom in how this is done, we will focus on a soft constraint approach.

Soft constraints allow composition in multiple dimensions, including different concerns for a given agent, and composition of constraints for multiple agents. In [17] a new system for expressing soft constraints called Monoidal Soft Constraints is proposed. This generalizes the Semi-ring approach to support more complex preference relations. In [18] partially ordered valuation structures are explored to provide operators for combination of constraints for different features

that respects the relative importance of the features. Steps towards a theoretical foundation for compositional specification of agent systems based on Soft Constraint Automata is presented in [24]. Although soft constraints provide an elegant and powerful foundation, as we will see there is much to do to develop principled ways to decompose problems and compose concerns with predictable results. Having compositional specification mechanisms is a step toward compositional reasoning which is import to manage the complexity of soft agent and similar systems. Systems operating in unpredictable environments, beyond our control are almost guaranteed to exhibit some kind of failure. Having a structured, compositional model will facilitate developing methods for determining the cause of failures and enabling (partial) recovery in many cases.

Plan. Section 2 discusses desiderata for a framework for Soft Agents. Section 3 presents the proposed framework and its formalization in Maude. Section 4 illustrates the application of soft-agents to a system of simple patrolling bots. Application to modeling more complex drone systems is also briefly summarized. Section 5 summarizes and discusses future directions.

2 Desiderata for Soft Agents

Cyber-physical agents must maintain an overall situation, location, and time awareness and make safe decisions that progress towards an objective in spite of uncertainty, partial knowledge and intermittent connectivity. The big question is: how do we design, build, and understand such systems? We want principles and tools for system design that achieve adaptive, robust functionality.

The primary desiderata for our Soft Agents are *localness*, *liveness* and *softness*. We explicitly exclude considering insincere or malicious agents in our current formulation.[1]

Localness. Cooperation and coordination should emerge from local behavior based on local knowledge. This is traditionally done by consensus formation algorithms. Consensus involves agreeing on what actions to take, which usually requires a shared view of the system state. In a distributed system spread over a large geographic area beyond the communication reach of individual agents, consensus can take considerable time and resources, but our agents must keep going. Thus consensus may emerge but can't be relied on, nor can it be be forced.

In less than ideal conditions what is needed is a notion of *sufficient consensus*: for any agent, consensus is sufficient when enough of a consensus is present so that agents can begin executing actions that are likely to be a part of a successful plan, given that there is some expectation for the environment to change.

Our partially ordered knowledge sharing (POKS) model for communication takes care of agreeing on state to the degree possible. In a quiescent connected

[1] This is a strong assumption, although not unusual. The soft agents framework supports modeling of an unpredictable or even "malicious" environment. We discuss the issue of trust or confidence in knowledge received as part of future work.

situation all agents will eventually have the same knowledge base. As communication improves, an soft agent system approaches a typical distributed system without complicating factors. This should increase the likelihood of reaching actual consensus, and achieving ideal behaviors.

A key question here is how a system determines the minimal required level of consensus? In particular what quality of connection/communication is required to support formation of this minimum level of consensus?

Safety and Liveness. Another formal property to consider concerns safety and liveness: often explained as something bad does not happen and something good will eventually happen. From a local perspective this could mean avoiding preventable disaster/failure as well as making progress and eventually sufficiently satisfying a given goal.

- An agent will never wait for an unbounded time to act.
- An agent can always act if local environment/knowledge/situation demands.
- An agent will react in a timely manner based on local information.
- An agent should keep itself "safe".

We note that the quality calculus [25,26] provides language primitives to support programming to meet such liveness requirements and robustness analysis methods for verification. One of the motivations of the Quality Calculus was to deal with unreliable communication. It will be interesting to investigate how soft constraints and the quality calculus approach might be combined to provide higher level specification and programming paradigms.

Softness. We want to reason about systems at both the system and cyber-physical entity/agent level and systematically connect the two levels. Agent behavior must allow for uncertainty and partial information, as well as preferences when multiple actions are possible to accomplish a task, as often is the case.

Specification in terms of constraints is a natural way to allow for partiality. *Soft constraints* provide a mechanism to rank different solutions and compose constraints concerning different aspects. We refer to [17] for an excellent review of different soft constraint systems. Given a problem there will be system wide constraints characterizing acceptable solutions, and perhaps giving measures to rank possible behaviors/solutions. Depending on the system wide constraints, global satisfaction can be a safety requirement, a liveness/progress requirement, a cooperation requirement, to name a few. Rather than looking for a distributed solution of global constraints, each agent will be driven by a local constraint system. Multiple soft constraint systems maybe be involved (multiple agent classes) and multiple agents may be driven by the same soft constraint system. A key challenge is developing reasoning and analysis methods to determine conditions under which satisfaction of local constraints will lead to satisfaction of global constraints, monotonically.

3 The Soft Agent Formal Framework

The soft-agent framework provides infrastructure and context for developing *executable* specifications of cyber-physical agents that operate in unpredictable environments, and for experimenting with and analyzing the resulting system behaviors. The framework, specified in the rewriting logic language Maude [15], provides generic data structures for representing system state (cyber and physical), interface sorts and functions to be used to specify the environment, agent capabilities (physical) and behavior (cyber). The semantics, how a system evolves, is given by a small number of rewrite rules defined in terms of these sorts and functions. A soft agent model specializes the framework by refining the sort hierarchy, adding model specific data structures, and providing definitions of the interface functions.

What can we do with an executable specification? Once a model is defined, we can define specific agent system configurations and explore their behavior by rewriting and search. We can watch the system run by rewriting according to different builtin or user defined strategies to choose next steps. In this way, specific executions and their event traces can be examined. Search allows exploration of all possible executions of a given configuration (up to some depth), to look for desired or undesirable conditions. Metadata can be added to a configuration to collect quantitative information such as the lowest energy before charging, close encounters, time elapsed between events, and so on.

As an example we introduce our main case study, *Patrol bots*, informally. A patrol bot moves on a grid along a fixed track (fixed value of the y coordinate). One or more squares of the grid are charging stations. We want to specify patrol bot behavior such that in a system with one or more patrol bot agent: agents do not run out of energy, and they keep patrolling.

In a little more detail, a soft agent system configuration consists of an environment component and a collection of agents. Each agent has local knowledge which can be shared, selectively and opportunistically with other agents when they come within "hearing distance" (in contact). For example, patrol bots share location. In a more complex scenario agents could learn terrain features and share that information. Knowledge sharing replaces traditional message passing as a communication mechanism. The framework caches shared knowledge to support propagation through a network of agents as they move around and meet different agents. Knowledge comes with a partial order that is used by the framework to replace stale or superseded knowledge. Of course an agent can keep a history by aggregating superseded information locally, for example as a list of values.

A specific model will specify for each agent some ability to observe its physical state and local environment, and some ability to act locally to change its physical state and local environment. A patrol bot can move one square at a time in any direction (i.e., to an adjacent square), subject to staying on the grid, and not moving to a blocked or occupied square. Moving uses energy. If the bot is at a charging square it can charge for some time until its energy capacity is reached. A patrol bot has sensors to read its location on the grid, and its energy level.

An agent's state consists of several attributes including its local knowledge base, pending events (tasks for the agent, or actions to be carried out), and cached knowledge to share with other agents.[2] A patrol bot's local knowledge includes the grid dimensions and charging station locations. It may store its location and energy, or simply read the sensors when it needs that information. In a context with more than one patrolling bot, it may store information about the location of other bots.

The environment is modeled as a knowledge base that contains information about the physical state of each agent, along with non-agent specific information such as features of the terrain, or current weather.

How do agents work? Agents carry out tasks (the cyber/reasoning part) and schedule actions to be executed by the environment (the physical part). The typical process for an agent carrying out a task is the following:

- read and process shared knowledge it has received;
- read and process sensor data (local information about the agents state and local environment conditions);
- think and decide
  - new knowledge to share
  - actions to execute
  - next tasks
  - updates for the local knowledge base: sensor readings to remember, new information learned or computed; new goals, things to check, ...

In the Patrol Bot model there is just one task, `tick`. This triggers the processing described above. In selecting actions, if not at a charging station, a patrol bot must decide whether to continue on its track or to detour to the nearest charging station.

How does the system run? Agents schedule tasks with a delay, often just one time unit, Actions are scheduled with a delay (possibly zero) and a duration. If an agent has a ready task (zero delay) then it can be carried out. Once there are no more ready tasks, ready actions are executed concurrently, sharable knowledge is exchanged, and time passes until there are one or more ready tasks.

In a scenario with two patrol bots, each with a ready task, then both will execute their tasks (order doesn't matter) scheduling the next `tick` with one time unit delay. Suppose both also schedule moves to different locations with 0 delay. Then the moves will be executed concurrently, one unit of time will pass, and now the two bot will again process their `ticks`.

Making explicit models of the physical state and agents knowledge allow modeling a range of situations, including the case where an agent has an accurate view of its situation and a variety of cases with discrepancies by independently varying different parts of the model. For example as we will see, it is easy to add wind effects to the model and see how well the agents adapt.

[2] The shared cache is separated from the local knowledge base so an agent can choose what to share and what to keep to itself.

The knowledge sharing communication model supports opportunistic communication where a connected network is not possible. In an ideal situation, knowledge propagates quickly and agents can have good situation awareness. However, an agent should not wait to hear from others. If an agent senses that action is needed, it should figure out the an action that is safe and makes progress towards its goal if possible, based on local information.

3.1 Introduction to Rewriting Logic and Maude

Rewriting logic [27] 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).

We note that rewriting with rules is similar to rewriting with equations (traditional term rewriting), in that we match the lefthand side and replace the matched subterm by the instantiated righthand side. The difference is in the way the rewriting is used. Equations are used to define functions by providing a means of computation the value of a function application. This means that the equations of an equational theory should give the same result independent of the order in which they are applied. Furthermore, equational rewriting should terminate. In contrast, rules are used to describe change over time, rather than computing the value of a function. They often describe non-deterministic possibly infinite behavior.

Maude is a language and tool based on rewriting logic [15,28]. Maude provides a high performance rewriting engine featuring matching modulo associativity, commutativity, and identity axioms; and search and model-checking capabilities. Thus, given a specification S of a concurrent system, one can execute S to find one possible behavior; use search to see if a state meeting a given condition can be reached; or model-check S to see if a temporal property is satisfied, and if not to see a computation that is a counter example.

To introduce Maude notation and give some intuition about how concurrent systems are specified and analyzed in Maude we consider specification of a simple Vending Machine. A Maude specification consists of a collection of modules. A module has a name, a set of imports (possibly empty), declarations of sorts and operations (constants, constructors, functions), equations defining functions, and rewrite rules. The vending machine specification is given in a module named `VENDING-MACHINE`.

```
mod VENDING-MACHINE is
  sorts Coin Item  Marking .
  subsorts Coin Item < Marking .
  op null : -> Marking [ctor] .  *** empty marking
  op _ _ : Marking Marking -> Marking
             [ctor assoc comm id: null] .
  ops $ q : -> Coin [ctor] .  *** dollar, quarter
  ops a c : -> Item [ctor] .  *** apple, cake
  rl[buy-c]: $ => c .
  rl[buy-a]: $ => a q .
  rl[change]: q q q q => $ .
endm
```

First several sorts (think sets or data types) are declared (key word `sort`). The basic sorts are `Coin` and `Item`. They represent what you put in and get out of the machine. The sort `Marking` consists of multisets of items and coins. This is specified by the subsort (subset) declarations saying that coins and items are (singleton) markings; and the declaration of the union operator (`_ _`, key word `op`). The blanks indicate operator argument positions, thus union of two markings is represented by placing them side by side, just as one represents a string of characters. The operator attributes `assoc`, `comm`, and `id:null` declare union to be associative and commutative with identity `null`, the empty marking. (Text following `***` is a comment.) After defining the data types to be used, some specific constants are declared: `$` (dollar) and `q` (quarter) of sort `Coin`; and `a` (apple) and `c` (cake) of sort `Item` (the keyword `ops` is used when declaring multiple constants of the same sort). The key word `ctor` in the operator attributes indicates the a constructor. Constructors are used to *construct* data elements, while non-constructor operators are used to name constants and functions defined by equations. The idea is that, using the equations, each term can be rewritten to an equivalent term in canonical form built from constructor operations. Maude takes care of all this under the hood, allowing the user to think abstractly in terms of equivalence classes.

Finally there are three rewrite rules specifying the vending machine behavior. The rule labeled `buy-c` says that if you have a dollar you can buy a cake. More formally, any marking containing an occurrence of `$` can be rewritten to one in which the `$` is replaced by a `c`. Similarly the rule labeled `buy-a` says that with a dollar you can also get an apple and a quarter change. The rule labeled `change` says that when four quarters have accumulated they can be changed into a dollar. Note that if a dollar is present in a marking, there are two ways that

the marking could be rewritten, each with a different outcome. If four quarters are also present, the change rule could be applied before or after one of the buy rules without affecting the eventual outcome.

To find one way to use three dollars, ask Maude to rewrite, and a quarter, an apple, and two cakes are the result.

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

Although there are several ways to rewrite three dollars, the Maude rewrite command uses a specific strategy for choosing rules to apply, and in this case chose to apply `buy-c` twice and `buy-a` once.

To discover more possibilities Maude can be asked to search for all ways of rewriting three dollars, such that the final state matches some pattern. For example, we can find all ways of getting at least two apples using the pattern

```
a a M:Marking
```

that is matched by any state that has at least two `a`s.

```
Maude> search $ $ $ =>! a a M:Marking .
Solution 1 (state 8)
M:Marking --> q q c
Solution 2 (state 9)
M:Marking --> q q q a
```

There are two ways this can be done. In one solution the remainder of the state consists of a cake and two quarters, (indicated by `M:Marking -> q q c` in Solution 1). In the other solution, there is a third apple and three quarters.

We can ask Maude to show us a path (list of rules fired) corresponding to one of these solutions using the command

```
Maude> show path labels 8 .
buy-c
buy-a
buy-a
```

to find the path to state 8. One can also ask for the full path to state 8. In this case the sequence of states and rules applied will be printed.

3.2 Key Sorts and Functions for Agent State

Now we see how the key sorts and functions for soft agent state are specified in Maude. Agent state is represented using knowledge and events.

Knowledge. Identifiers are used to identify agents and to link knowledge and events to agents. The module `ID-SET` declares the sort `Id` of identifiers and defines `IdSet` to be multisets of identifiers, using the same mechanism that we explained in the Vending machine definition of `Marking`. The attributes `assoc`, `comm`, and `id: none`, say that the union operation (`_ _`) is associative and commutative with identity `none`. The subsort declaration `Id < IdSet` says that identifiers are also singleton identifier sets.

```
fmod ID-SET is
  sort Id .  sort IdSet .   subsort Id < IdSet .
  op none : -> IdSet [ctor] .
  op _ _ : IdSet IdSet -> IdSet [ctor assoc comm id: none] .

  var id : Id .  var ids : IdSet .
  op member : Id IdSet -> Bool .
  eq member(id, id ids) = true .
  eq member(id, ids) = false [owise] .
endfm
```

The boolean function, `member`, tests if an identifier is an element of a given set. Variables are declared with the key word `var` and give the variable name and its sort. The definition of `member` uses Maude's builtin matching modulo AC capability to check if the second argument can be written in the form `id ids`. If so, we have found `id` in the given set. If the match fails, then `id` is not a member of the given set. The equation with the `owise` attribute applies in this case.

The structure of knowledge is specified in the `KNOWLEDGE` module.

```
fmod KNOWLEDGE is
  inc ID-SET .  inc NAT-TIME-INF .

  sort KB .  subsort KItem < KB .
  op none : -> KB [ctor] .
  op __ : KB KB -> KB [ctor assoc comm id: none] .

  sorts PKItem TKItem .  *** Persistent, Timed
  subsort PKItem TKItem < KItem .

  sort Info .
  op _@_ : Info Time -> TKItem .

  *** knowledge partial order
  op _<<_ : KItem KItem -> Bool .
  eq ki << ki' = false [owise] .

    op addK : KB KB -> KB .
    ...
endfm
```

The statements

```
inc ID-SET .  inc NAT-TIME-INF .
```

are import statements. They effectively include the declarations and operations of the modules `ID-SET` and `NAT-TIME-INF` into `KNOWLEDGE`. The module `NAT-TIME-INF` models discrete time as natural numbers and adds a notion of infinity which is a convenient for many purposes.

The main knowledge sorts are `KItem` (knowledge item) and `KB` (knowledge base, a multiset of knowledge items). `PKItem` (persistent knowledge items) and `TKItem` (time dependent knowledge items) are subsorts of `KItem`. The sort `Info` represents the content of a `TKItem`, and becomes a `TKItem` when timestamped (`info @ t`). Key to managing evolving knowledge is the partial order, `<<`, on knowledge items. `ki << ki'` is intended to mean that `ki'` supersedes `ki`. Often this is because `ki'` has a later timestamp, but this is not necessarily so. The function `addK(kb0,kb1)` adds the knowledge base `kb1` to `kb0` using the partial order to drop any superseded knowledge.

The following extract from modules `LOCATION-KNOWLEDGE` and `CLOCK-KNOWLEDGE` specifies three forms of knowledge provided by the soft agent framework for use in specific agent models.

```
*** knowledge elements available to all models
  sort Class .
  op class : Id Class -> PKItem .
  op clock : Time -> KItem [ctor] .
  sort Loc .
  op noLoc : -> Loc [ctor] .
  op atloc : Id Loc -> Info [ctor] .

  vars t0 t1 : Time .
  var id : Id .   vars loc0 loc1 : Loc .
  eq clock(t0) << clock(t1) = t0 < t1 .
  ceq atloc(id,loc0) @ t0 << atloc(id,loc1) @ t1 =  t0 < t1 .
```

Soft agents are organized in classes, similar to object oriented models. The operator `class` defines a (partial) mapping from identifiers to classes. Soft agent classes are intended to capture fixed physical aspects that don't change over time. Thus the return sort for `class` is `PKItem`. For example, the intent is that an agent of class `Bot` (such as our patrol bot), that rolls on wheels, should not turn into an agent of class `Drone`, that flys. Information constructors could be added to capture information about behavior class that might change over time, for example to specify an agents *role* in a protocol or other coordinated activity.

The term `atloc(id,loc0) @ t0` says that the agent with identifier `id` was at location `loc2` at time `t0`. The operator `clock` is an exception to the convention that time sensitive items are obtained by time stamping information terms. A term `clock(t0)` in a local knowledge base says that the agents local time is `t0`.

The equations defining << for clocks and location formalize the decision that only the latest information should be kept. For example letting

```
lkb = clock(0) class(b(0),Bot) atloc(b(0),pt(0,0)) @ 0)
```

where `b(0)` is an identifier, `pt(0,0)` and `pt(1,0)` are grid locations, and `Bot` is a class, we have the following

```
addK(lkb, atloc(b(0),pt(1,0)) @ 1) =
   clock(0) class(b(0),Bot) (atloc(b(0),pt(1,0)) @ 1)
```

because $0 < 1$.

Events. Agents behavior is organized using events, formalized in the module `EVENTS` starting with the main sorts, `Event` and `EventSet` (multisets of events).

```
fmod EVENTS is inc KNOWLEDGE .

  sorts Event EventSet .   subsort Event < EventSet .
  op none : -> EventSet .
  op __ : EventSet EventSet -> EventSet [ctor assoc comm id: none] .

*** Tasks
  sort Task .
  op tick : -> Task [ctor] .  --- default tasks

*** Actions (must be annotated by ids)
  sorts Action ActSet .  subsort Action < ActSet .
  op none : -> ActSet [ctor] .
  op __ : ActSet ActSet -> ActSet [ctor assoc comm id: none] .
*** immediate events .
  sort IEvent .  subsort IEvent < Event .
  op rcv : KB -> IEvent [ctor] .
*** delayed events
  sort DEvent .  subsort DEvent < Event .
  op _@_ : Task Time -> DEvent .
  op _@_;_ : Action Time Time -> DEvent .
endfm
```

There are two kinds of event, immediate events (sort `IEvent`) and delayed events (sort `DEvent`). `rcv(kb)` is an immediate event, representing notification of newly arrived shared knowledge. Delayed events are built from tasks (sort `Task`) and actions (sort `Action`). Tasks are used to trigger agents to process information and schedule actions to be carried out. `task @ delay` is a delayed event specifying that `task` should be done after `delay` time units have passed. The generic task `tick` is provided by the framework. A specific agent model can introduce additional tasks as needed. `act @ delay ; duration` is a delayed event specifying that the action `act` is to be enacted starting after `delay` time units and lasting `duration` time units. For example in response to a `tick`, a

patrol bot with identifier `b(0)` might schedule `mv(b(0),E) @ 0 ; 1`— move east, now, for one time unit. It might also schedule `tick @ 2`, to repeat the processing again in 2 time units. If the patrol bot is one square north of a charging station it could schedule

```
(mv(b(0),S) @ 0 ; 1) (charge(b(0)) @ 1 ; 2) tick @ 3
```

to move to the charging station, charge for two time units and then repeat the task processing. We note that action terms must have an identifier, since ready actions from all agents are collected and executed concurrently. Thus it is necessary to know who is doing the action.

3.3 Agents and Configurations

Now we can formalize the structure of a soft agent (module `AGENTS`) and of agent systems (module `AGENT-CONF`).

```
fmod AGENTS is
  inc LOCATION-KNOWLEDGE .  inc CLOCK-KNOWLEDGE .
  inc EVENTS .

  sorts Attribute AttributeSet .
  subsort Attribute < AttributeSet .
  op none : -> AttributeSet  [ctor] .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
           [ctor assoc comm id: none] .

  sort Agent  .
  op [_:_|_] : Id Class AttributeSet -> Agent [ctor] .
  op lkb`:_ : KB -> Attribute [ctor] .
  op ckb`:_ : KB -> Attribute [ctor] .
  op evs`:_ : EventSet -> Attribute [ctor] .
endfm
```

In the spirit of object oriented specification, an agent has an identifier (recall the module `IDSET`), a class, and a set of attributes. Three attributes are essential: `lkb` (the local knowledge base); `ckb` (the knowledge cache for sharing); and `evs` (pending events for this agent). A specific agent model might introduce additional attributes if needed. The initial state of a patrol bot might look like the following

```
[ b(0) : Bot |
  lkb: class(b(0),Bot) (myDir(b(0),E) @ 0) (myY(b(0),0) @ 0)
       class(s(0),Station) (atloc(s(0),pt(2,1)) @ 0),
  ckb: none, evs: tick @ 1]
```

where `Station` is the class of charging stations, `myY(b(0),0)` says this bot should travel along $y = 0$ and `myDir(b(0),E)` says the bot is currently traveling east. In this case the local knowledge base does not store location and energy level since the corresponding sensors are read each time the bot processes a task.

A soft agent system configuration (sort `Conf`) is a multiset of configuration elements (sort `ConfElt`). The sort `ASystem` is used for top-level configurations.

```
fmod AGENT-CONF is
  inc AGENTS .
  sorts  ConfElt Conf .
  subsorts Agent  < ConfElt < Conf .
  op none : -> Conf .
  op __ : Conf Conf -> Conf [ctor assoc comm id: none] .

  sort Env . subsort Env < ConfElt .
  op [_|_] : Id KB -> Env [ctor] .

  op restrictKB : Id KB -> KB .
  ...

  sort ASystem .
  op `{_`} : Conf -> ASystem .
  op mte : Conf -> TimeInf .
endfm
```

An agent is a configuration element (`subsort Agent < ConfElt .`), and environment objects (sort `Env` are also configuration elements (`subsort Env < ConfElt .`). An environment object has an identifier and a knowledge base. A top-level system should have a unique environment object. The environment knowledge base includes the (physical) state of each agent as well as environment knowledge such as terrain, obstacles, or weather conditions. For example an environment knowledge base for a single patrol bot scenario could be the following

```
clock(3)
class(b(0),Bot) (atloc(b(0),pt(2,0)) @ 3) (energy(b(0),10)) @ 3)
class(s(0),Station) (atloc(s(0),pt(2,1)) @ 0)
wind(N,pt(1,2)) @ 0)
```

where 3 time units have passed and the bot is now at grid location $(2, 0)$ with energy 10. There is northerly wind at grid location $(1, 2)$.

The function `restrictKB(id,kb)` selects the knowledge items in `kb` with identifier `id`. For example, restricting the above knowledge base to `b(0)` results in the following.

```
class(b(0),Bot) (atloc(b(0),pt(2,0)) @ 3) (energy(b(0),10)) @ 3)
```

The operator $\{\_\}$ is used to collect configuration elements to define a top-level system state of sort `ASystem`. The function `mte` computes how much time can elapse before some task is ready (0 delay).

Specific agent models can introduce new configuration elements to facilitate analysis. For example, simple flags can be defined that indicate a goal has been reached or an invariant has failed. Metadata elements can be defined that monitor execution state and collect information.

3.4 Rules

There are two rewrite rules: `doTask` that controls carrying out agent tasks, and `timeStep`, that carries out actions concurrently, propagates any sharable knowledge, and increments time.

The `doTask` rule uses the `doTask` function to determine how to update the agents state.

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

The function `getTime(lkb)` gets the local time from the `clock` knowledge of a knowledge base, in this case the agents view of the current time. `splitEvents` splits the input event set into a pair consisting of the immediate events (`ievs`), and the delayed events (`devs`). Immediate events are to be processed by the agent. The only immediate events provided by the framework are `rcv` events, holding knowledge received from other agents by sharing. Delayed events are actions to be executed by the environment. The agent is free to add to or delete/cancel elements of its delayed events. The `doTask` function is declared as part of the framework,

```
op doTask : Class Id Task EventSet EventSet KB KB
          -> KBEventsKBSet.
```

but equations defining this function must be provided by each model. `doTask` is given the agents class (`cl`), its identifier (`id`), and the task to be addressed (`task`).[3] In addition it is given unprocessed knowledge acquired by knowledge sharing (`ievs`), the agents current scheduled tasks and actions (`devs`, delayed events), the agents local knowledge base (`lkb`), and the environment knowledge base from which it can extract the agents current sensor readings (`ekb`) and other local information. `doTask` returns a triple: the updated local knowledge base (`lkb'`), updated scheduled tasks and actions (`evs'`), and information items to be shared with other agents (`kb`), which is added to the agents cache to produce the updated cache knowledge base (`ckb'`).

The `timeStep` rule concurrently performs actions using the function `doEnvAct`. It updates delays and durations in the event sets (counting down) using the function `timeEffect`. It propagates sharable knowledge using the

[3] In fact, we use a single task, tick, to schedule the next invocation of an agents cyberactivity. Information about what to do is kept in the local knowledge base.

function shareKnowledge. The function updateAConf is an auxiliary function used during model analysis to collect and update metadata (stored as configuration elements that are ignored by other framework functions that operate on configurations).

```
crl[timeStep]:
{ aconf }
=>
{ aconf2 }
if nzt := mte(aconf)
/\ t := getTime(envKB(aconf))
/\ ekb' := doEnvAct(t, nzt, envKB(aconf), effActs(aconf))
/\ aconf0 := updateEnv(ekb',timeEffect(aconf,nzt))
/\ aconf1 := shareKnowledge(aconf0)
/\ aconf2 := updateConf(aconf1)
 [print "eAct:" ekb' "\ntimeStep:" t "++" nzt] .
```

Note that both rules have a print attribute. The print attribute specifies a string to be printed (if the print option is turned on) each time the rule is applied. The keyword print is followed by one or more string literals and/or variables bound by rule matching. This is a handy way to produce an execution trace with just information you care about.

The function doEnvAct determines the effects of agents actions on the environment, including the agents physical state and interactions with neighbors. The function is given the current time (t), the maximum time that can elapse (nzt), the current situation (represented by the environment knowledge base, envKB(aconf)), the actions to be executed (effActs(aconf), actions, collected from all agents, that are no longer delayed), and the amount of time that has elapsed in the current execution round. doEnvAct returns the environment knowledge base resulting from (concurrent) execution of these actions along with the amount of time passed (t') which is as most nzt. This is used in case execution fails and less than the allotted time has passed.

doEnvAct proceeds by time increments (in the discrete case 1 time unit). The ready actions are concurrently executed for 1 time increment, each in the same initial environment using the function doUnitEnvAct. The combined environment resulting from the concurrent effects is used for the next time increment.[4]

Specifically, doEnvAct is defined by the conditional equations

```
ceq doEnvAct(t, nzt, ekb, evs, t')
  = doEnvAct(s(t), nzt monus 1, ekb', timeEffect(evs,1), s(t'))
  if ekb' := doUnitEnvAct(t, ekb, evs, none)
  /\ isOk(ekb).
```

[4] It is possible that the concurrent actions are in conflict. In the current framework, we have two options. One is that time stops, and the user can investigate what went wrong. Or some actions are arbitrarily chosen to succeed and others fail. This is not unreasonable, since true simultaneity is rare.

```
eq doEnvAct(t, t'', ekb, evs, t') = {ekb,t'} [owise]
   --- applies if t'' is 0 or ekb' is not Ok

eq doUnitEnvAct(t, ekb, (act @ 0 ; nzt) evs, ekb')
  = doUnitEnvAct(t, ekb, evs,
     addK(ekb',doUnitEnvOneAct(act,ekb,t))).
```

`isOk` recognizes impossible environments (like two agents in the same location). The function `doUnitEnvAct` simply iterates over the events, accumulating environment updates returned by `doUnitEnvOneAct(act,ekb,t)`.

`doUnitEnvOneAct` is the heart of the matter. It models the physics of the action, `act`. Recall that actions are required to have an identifier component that determines the agent doing the action.

As an example, a move action `mv(id,dir)` will change the location of the agent identified by `id`, according to environmental conditions such as boundaries, terrain slope, or wind. The default is to move one unit in the specified direction `dir`.

4 Case Studies

In this section we present two soft agent case studies. The first is the simple Patrol Bot mentioned in Sect. 3 that we now formalize in some detail. The second is a more complex surveillance drone case study that we summarize briefly focusing on a set of formal analysis results. We begin by defining an extension of the soft agent framework to support use of soft constraints to specify agent behavior that is robust to modest disruptions.

4.1 Soft Constraints

Although the framework does not enforce a particular mechanism for defining the `doTask` function, we provide an example template for this function to facilitate use of soft constraint problem solving to determine what actions an agent might consider in a given situation.

The idea of soft constraints is to constrain possible values of a set of variables by mapping such assignments to a partially ordered domain and then selecting assignments with maximal value. Traditionally soft constraints use an algebraic structure such as c-Semirings as the valuation domain. Such structures have good properties with respect to different forms of composition [17,18,24]. In our examples, we do not need all the machinery of c-Semirings, so to simplify the formalization so we defined a theory `VALUATION` that captures what we do require of a valuation domain for our SCPs.

```
fth VALUATION is
  pr BOOL .  inc SOFT-AGENTS .
  sort Grade .
  op equivZero : Grade -> Bool .
```

```
  op _<_ : Grade  Grade -> Bool .
  op val : Id KB Action  -> Grade .
endfth
```

Specifically, there is a sort `Grade` (the values), a partial order < on `Grade`, a predicate, `equivZero`, that recognizes the minimal element(s) of `Grade`, and a valuation function `val` that evaluates an action from the point of view of the identified agent in the context of a knowledge base. Typically a specific instance of `val` will measure the effect of an action, carried out in a situation represented by the knowledge base–is it safe?, is it progressing towards some goal?, and so on. For example the evaluation of a patrol bots action with respect to energy (it should not run out of energy), might return 0 if the action leads to a state in which there is not sufficient energy to get to a charging station and 1 otherwise. This could be refined to return 1/2 if the energy is sufficient to get to a charging station, but with less than 1/4 the energy capacity to spare (formalizing a notion of caution).

The module `SOLVE-SCP` provides a simple mechanism for solving a soft constraint problem by finding the maximal solutions, i.e. the maximally graded actions. It is parameterized by a module `Z` that satisfies the theory `VALUATION`.

```
fmod SOLVE-SCP{Z :: VALUATION} is
   inc SOFT-AGENTS .

  sorts RankEle{Z} RankSet{Z} .
  op {_,_} : Z.Grade ActSet -> RankEle{Z} .

  subsort RankEle{Z} < RankSet{Z} .

  op updateRks : RankSet{Z} Action Z.Grade -> RankSet{Z} .
  op solveSCP : Id KB ActSet -> ActSet .
  eq solveSCP(id,kb,acts) = solveSCP!(id,kb,acts,none) .

  op solveSCP! : Id KB ActSet RankSet{Z} -> ActSet .
  eq solveSCP!(id,kb,none,rks) = getAct(rks) .
  ceq solveSCP!(id,kb,act actset,rks) =
        solveSCP!(id,kb,actset,rks1)
   if v0 := val(id,kb,act)
   /\ rks1 := updateRks(rks,act,v0) .
endfm
```

The function `solveSCP` (solve Soft Constraint Problem) takes an agent identifier, a knowledge base and an action set and returns the actions in the input action set with maximal grade according to the parameter theory. This function maintains a ranked action set where the ranks/grades are non zero and maximal among the actions considered so far. It uses the auxiliary function `updateRks` to update this set given the grade of an action. The function `equivZero` is used to ensure that the returned actions have non-zero grade. This method of solving soft constraint problems works well when there are only a few actions

to consider. More efficient methods will be needed when situations get more complex.

To the degree possible we would like to derive valuation functions for agents from valuations with respect to different concerns. The choice of combination operation is not a simple task. Lexicographic combination of partial orders works in some cases [18], however it is not always possible to use a lexicographic ordering to obtain the desired combination. Initial steps towards a theory of composition of soft constraint problems/automata is presented in [24]. For the present the soft agent framework does not provide builtin compositions operations. Each agent model will need to develop its own valuation functions and combination methods. We will see examples in the case studies below.

4.2 doTask Template

We now introduce the specialization of the `doTask` function that uses `solveSCP` to determine the actions for an agent to consider. Recall that `doTask` returns a triple consisting of an update for the agents local knowledge base, an update for the agents event set, and knowledge to share. Then `doTask` is specified by the following conditional equation, that formalized the task process outlined in beginning of Sect. 3.

```
ceq doTask(cl,id,tick,ievs,devs,ekb,lkb) =
  if acts == none then
     {lkb2, devs (tick @ botDelay), none }
  else
     selector(doTask!(id,lkb2,devs (tick @ botDelay),acts))
   fi
 if lkb0 := handleS(cl,id,lkb,ievs)
 /\ lkb1 := getSensors(id,ekb)
 /\ lkb2 := proSensors(id,lkb0,lkb1)
 /\ acts0 := myActs(cl,id,lkb2)
 /\ acts := solveSCP(id,lkb2,acts0) .
```

The functions `handleS`, `getSensors`, `proSensors`, and `myActs` are declared by the framework, but must be defined by each specific agent model. The function `handleS` processes the new shared knowledge (in `ievs`) producing an updated local knowledge base (`lkb0`). It could simply add the new knowledge, or could be more selective, do simple reasoning, or aggregate incoming data. The function `getSensors` reads relevant sensors (and local environment information) from the environment knowledge base (`ekb`), and the function `proSensors` processes the result, `lkb1`, updating `lkb0` to produce the updated local knowledge `lkb2`. The function `myActs` returns a list of actions that are possible given the current situation, represented by `lkb2`. For example a `charge` action is only possible if the agent is at a charging station, and move actions may be constrained by boundaries or known obstacles.

The function `doTask!` constructs a result triple for each of the actions, `act` in `acts`, returned by `solveSCP`.

```
op doTask! : Id KB EventSet ActSet -> KBEventsKBSet .
ceq doTask!(id,lkb2,devs,act acts) =
    {lkb3,devs (act @ 0 ; 1),kbp} doTask!(id,lkb2,devs,acts)
 if kbp := tell(id,act,lkb2)
 /\ lkb3 := remember(id,act,lkb2) .
eq doTask!(id,lkb2,devs,none) = none .
```

`doTask!` uses `remember(id,act,lkb2)` to compute the local knowledge update, and `tell(id,act,lkb2)` to compute what new knowledge to share. By default, the updated events consist of `devs`, `(tick @ botDelay)` (to schedule the next execution of `doTask`), and `act @ 0 ; 1` (scheduling the action to happen immediately for one time unit). This can be overridden as needed. The `selector` function, by default, returns all the result triples. Alternatively, one result could be picked arbitrarily (by Maude or by tossing a coin).

4.3 The Patrol Bot Case Study

The idea of patrol bots was introduced at the beginning of Sect. 3. Now we address the problem of specifying a patrol bot system, including knowledge, the possible actions of a patrol bot, the effects of these actions, and the soft constraint problem to be solved for deciding actions. Once individual patrol bots have been specified we define some scenarios to illustrate the use of the specifications to study possible system behaviors.

Recall that a patrol bot is moving on a grid, along some track (fixed y) continually going from one side to the other. Moving uses energy, so the patrol bot must recharge to avoid dying (so there must be a charging station somewhere on the grid). The patrol bot may drift off its path (faulty motor, or wind, ...). It prefers to move along the assigned track so it should correct when it discovers it has drifted.

In addition to class, clock and location knowledge, patrol bot knowledge includes `energy` (a sensor reading), `caution` (how much reserve energy to keep), and its patrolling parameters `myY,myDir`.

```
**** Info constructors
  op energy : Id FiniteFloat -> Info [ctor] .
  op caution : Id FiniteFloat -> Info [ctor] .
  op myY : Id Nat -> Info [ctor] .    *** the bot's track
  op myDir : Id Dir -> Info [ctor] .  *** current direction
**** partial order--new information superceeds old
  eq energy(id,e0) @ t0 << energy(id,e1) @ t1 =  t0 < t1 .
  eq caution(id,e0) @ t0 << caution(id,e1) @ t1 =  t0 < t1 .
  eq myY(id,y0) @ t0 << myY(id,y1) @ t1 = t0 < t1 .
  eq myDir(id,dir0) @ t0 << myDir(id,dir1) @ t1 = t0 < t1 .
```

A patrol bot has two kinds of action: `charge`, to restore energy; and `mv`, to move one step in the given direction. The directions `E,W,N,S` stand for East, West North, and South (or left, right, up, and down).

```
*** actions
  sort Dir .   ops E W N S : -> Dir [ctor] .
  op mv : Id Dir -> Action [ctor] .
  op charge : Id -> Action [ctor] .
```

The effects of patrol bot actions depend on several global parameters, listed below.

```
*** global parameters
  ops gridX gridY : -> Nat [ctor] .          *** grid dimensions
  op chargeUnit : -> FiniteFloat [ctor] .    *** energy gained
  op maxCharge : -> FiniteFloat [ctor] .     *** energy capacity
  op botDelay : -> Nat [ctor] .              *** time between ticks
  op costMv : -> FiniteFloat [ctor] .        *** cost of moving
```

An agent's model of the effects of actions is given by the functions `doMv` and `doAct`.

```
  op doMv : Loc Dir -> Loc .
  op doAct : Id KB Action -> KB .
```

`doMv` simply returns the location after the move. In case the result would be off the grid, the initial location is returned. `doAct` updates the given knowledge base with the result of the action. Energy will be decreased in the case of a move action and increased by `chargeUnit` in the case of a charge action. The model assumes actions are carried out for one time unit, thus the new information is time stamped with a time that is one time unit in the future, the time the action completes.

The physical model of the effect of actions is given by `doUnitEnvOneAct` which is used in the `timeStep` rule. For actions that operate as expected, given the physical state (as reflected by the environment knowledge base), the `doUnitEnvOneAct` result is the same as `doAct`. This is the ideal case. In other cases `doUnitEnvOneAct` will differ. For example, charging stops when the `maxCharge` is reached. If there is an obstacle in the target of a move, then the agent doesn't move, but it does use energy trying. If there is wind, the final location may be different depending on the strength and direction of the wind.

doTask Auxiliary Functions. Recall that an agents behavior is specified by the function `doTask` that is defined in terms of auxiliary functions for handling shared knowledge and processing sensors to define the soft constraint problem to be solved. The function `handleS` processes newly received shared knowledge by simply adding it to the local knowledge base.

```
  op handleS : Class Id KB EventSet -> KB .
  eq handleS(cl,id,lkb,rcv(kb) ievs) = addK(lkb,kb) .
```

Sensors are read by restricting the environment knowledge base to the patrol bots identity.

```
op getSensors : Id KB -> KB .
eq getSensors(id,ekb) = restrictKB(id,ekb) .
```

Sensor processing adds sensed location and energy information to the local knowledge base. It also reverses the direction (`reverseDir(dir)`) if the current location is at one of the vertical edges (`atVEdge(loc)`).

```
op proSensors : Id KB KB -> KB .
ceq proSensors(id,lkb,ekb) = lkb1
 if ((myDir(id,dir) @ t0) clock(t) lkb0) := lkb
 /\ (atloc(id,loc) @ t1) (energy(id,e) @ t2) ekb0 := ekb
 /\ dir1 := (if atVEdge(loc) then reverseDir(dir) else dir fi)
 /\ lkb1 := addK(lkb0 (myDir(id,dir1) @ t) clock(t),
                 (atloc(id,loc) @ t1) (energy(id,e) @ t2)) .
```

By reversing the direction when the patrol bot reaches an edge it will always be able to move in the 'current' direction.

The possible actions in a given situation (`myMvs(Bot,id,lkb)`) include charging, if current location is that of a station and not fully charged, and moves in any direction that do not lead to an occupied location or a location that is off the grid (`myMvs(Bot,id,lkb)`).

```
op myActs : Class Id KB -> ActSet .
eq myActs(Bot, id,lkb) =
(myMvs(Bot,id,lkb)
(if canCharge(Bot,id,lkb) then charge(id) else none fi)) .
```

The functions `remember` and `tell` are used by the auxiliary `doTask!` to assemble triples from the set of actions returned by `solveSCP`.

```
op remember : Id Action KB -> KB .
eq remember(id,act,lkb) = lkb .
op tell : Id Action KB -> KB .
eq tell(id,act,(atloc(id,loc) @ t) lkb) = (atloc(id,loc) @ t) .
```

A patrol bot remembers all the information gathered to define the soft constraint problem. In fact, the location and energy could be forgotten as they are simply overridden during the next task processing. Only the location is shared. This is useful when there are several patrol bots, to avoid potential collisions or blocking.

Valuation Functions. What remains is to specify the valuation function used by a patrol bot to choose among available actions. The valuation function should ensure that the patrol bot does not run out of energy (assuming sufficient charge capacity and accessible charging stations). It should also ensure that the patrol bot continually patrols, i.e. it reaches one side, turns, reaches the other side, turns .... Given the two different *concerns* we define two valuation functions: `val-energy` for assuring the energy requirement and `val-patrol` for maximizing patrolling progress. `val-energy` is overloaded in that it can evaluate a

knowledge base (`val-energy(id,kb)`) or an action in the context of a knowledge base (`val-energy(id,kb,act)`). In either case the valuation is relative to a specific patrol bot, identified by `id`. `val-energy` returns an element of `TriVal` which has three elements ordered by `bot < mid < top`.

```
op val-energy : Id KB Action -> TriVal .
eq val-energy(id,
       (energy(id,e) @ t0) (atloc(id,loc0) @ t)
       (atloc(st,loc1) @ t1) (class(st,Station)) kb,
        charge(id)) =
       (if (loc0 == loc1)
        then (if (e >= maxCharge) then bot else top fi)
        else bot fi) .
```

For a charge action, the value is `top` if the patrol bot is at a charging station and is not fully charged. Otherwise the value is `bot`. In the case of move actions, the resulting knowledge base is evaluated for energy safety.

```
ceq val-energy(id,kb,mv(id,dir)) =
        val-energy(id,doAct(id,kb,mv(id,dir)))
if not (val-energy(id,doAct(id,kb,mv(id,dir))) == mid) .
```

If the result of knowledge base valuation is not `mid` then that result is returned.

```
ceq val-energy(id,kb,mv(id,dir)) =
      if towards(dir,loc,locb) then  mid else bot fi
 if val-energy(id,doAct(id,kb,mv(id,dir))) == mid
 /\ (atloc(id,loc) @ t) (atloc(st,locb) @ t1)
    (class(st,Station)) kb' := kb .
```

If the result of knowledge base valuation is `mid` then energy valuation prefers a move in the direction of a charging station.

```
op val-energy : Id KB  -> TriVal .
eq val-energy(id,(energy(id,e) @ t0) (atloc(id,loc) @ t)
   (atloc(st,locb) @ t1) (class(st,Station)) kb) =
 if e > cost2loc(loc,locb) + caution then top else
  (if e > cost2loc(loc,locb) then mid else bot fi) fi .
```

Energy valuation of a knowledge base uses the `caution` parameter that determines how much reserve energy it prefers. If the situation allows the agent to reach a charging station with out running out of energy, but the reserve energy is less than the caution parameter the value returned is `mid`. Otherwise the value is `top` if it is energy safe and `bot` if unsafe.

The valuation from a patrolling perspective, computed by `val-patrol`, has a range of 0.0 to 1.0. The value for charging is 1.0 assuming it will not be asked if charging is not feasible.

```
op val-patrol : Id KB Action -> Float .
eq val-patrol(id,
       (atloc(id,pt(x,y)) @ t) class(id,Bot)
       (myDir(id,dir) @ t0) (myY(id,y0) @ t1) kb,
       mv(id,dir1)) =
    (if (y0 < y)
     then (if  (dir1 == S) then 0.9 else 0.0 fi)
     else (if (y < y0)
           then (if  (dir1 == N) then 0.9 else 0.0 fi)
           else (if  (dir == dir1) then 0.9 else 0.0 fi) fi) fi).
eq val-patrol(id, kb, charge(id))  = 1.0 .
```

A move is given value 0.9 if the agent is off track and the move corrects, or if the move is in the current direction. From a patrolling perspective, charging is preferred if it is possible, otherwise moves that correct or move towards the current goal are preferred.

The combined valuation function `val` returns a pair: the first component is the energy valuation and the second component is the patrol valuation.

```
sort BUVal .
op {_,_} : TriVal Float -> BUVal .
op val : Id KB Action -> BUVal .
eq val(id,kb,act) =
   {val-energy(id,kb,act),val-patrol(id,kb,act)} .
op _<_ : BUVal BUVal -> Bool .
op equivZero : BUVal -> Bool .
eq {b1,u1} < {b2,u2} = (b1 < b2) or (b1 == b2 and u1 < u2) .
eq equivZero({b1,u1}) = (equivZero(b1)) .
```

The partial order on these pairs is the lexicographic order [18] with energy valuation (safety) given preference. A value pair is equivalent to zero just if the energy component is equivalent to zero. Thus energy consideration alone can veto an action. But an action with non-zero energy value is acceptable even if the patrol value is equivalent to zero (i.e., is 0.0).

Here are a few examples. We assume a knowledge base with a patrol bot moving east, along $y = 0$, in a 5×3 grid.

```
loc   energy caution act   val          comment
(2,0) 5.0     1.0     mv(E) {top, 0.9}  min caution move E
                      mv(N) {top, 0.0}
(2,0) 5.0     4.0     mv(E) {bot, 0.9}   more caution, N wins
                      mv(N) {mid, 0.0}
(2,1) 10.0    1.0     charge {top,1.0}
(2,1) 25.0    1.0     charge {bot,1.0} fully charged
```

With caution 1.0 and energy 5.0 moving east is preferred, but then the patrol bot will need to backtrack. With caution 4.0 and energy 5.0 moving north to the charging station is preferred over moving east in the patrolling direction since `{bot, n} < {mid,n'} = true`. Also, when at the charging station and fully charged, a charge action will not be considered since `not(equivZero({mid,0}) = true`.

4.4 Patrol Bot Scenarios

In this section we show how rewriting and search can be used to gain understanding of agent system behavior, the reasons for failures, and the effects of changing parameters. To illustrate how environment effects can be introduced we add potential for wind to blow a patrol bot off course. It is not intended to be particularly realistic as a model of wind, but it does allow us to see how robust the patrol bots can be by just varying the frequency of disruption and the level of caution.

```
op wind : Dir Nat -> Info .
op windEffect : Loc KB -> Loc .
ceq windEffect(l0, (clock(t)) (wind(dir,n) @ t0) ekb)
  = doMv(l0,dir)
 if t rem n == 0 .
eq windEffect(l0,ekb) = l0 [owise] .
```

`wind(dir,n) @ t0` specifies wind effect in direction `dir` to be applied if the current time is divisible by `n`. The point is that we don't want continual wind. Periodic wind for different periods can test robustness without needing to introduce machinery for probability distributions. The function `windEffect` is applied to the result of a move in the function `doUnitEnvOneAct` that is the core of the `doEnvAct` function used in the `timeStep` rule.

A Patrol bot system consists of one or more patrol bots each with their own track and current direction, together with an environment. A system can be instrumented for analysis in various ways. For example, to bound an execution we define a new configuration element.

```
op bound : Nat -> ConfElt .
```

The auxiliary function `updateConf` that is applied during the `timeStep` rule is used to decrement the bound each time step. When the bound reaches 0 the configuration is replaced by a constant `goalConf` for which there are no rewrite rules, so execution/search must stop. To analyze a system we define a notion of *critical configuration* and carry out bounded search for such configurations.

```
ops criticalConf goalConf : -> ConfElt .
op critical : Conf -> Bool .

ceq critical([eId | (energy(id,ff) @ t) kb ] aconf) = true
 if equivZero(val(id,(energy(id,ff) @ t) kb)) .
eq critical(aconf) = false [owise] .

eq updateConf(bound(n) aconf) =
  if critical(aconf) then criticalConf aconf
  else (if (n == 0) then goalConf
        else bound(monus(n)) aconf fi) fi .
```

When a critical configuration is reached the constant `criticalConf` is added to the configuration by the function `updateConf` to simplify specifying the search command. For our examples, we define a critical configuration to be one in which the valuation for some agent is equivalent to zero using the boolean function `critical`.

We begin with a one patrol bot configuration, watch it run and search for critical configurations in a family of configurations parameterized by the level of caution and the frequency of wind. Then we will look at what happens when a second patrol bot is added. In all cases we will use the same global parameters:

```
eq gridZ = 5 .
eq gridY = 3 .
eq chargeUnit = 5.0 .
eq maxCharge = 20.0 .
```

To simplify notation we define an agent template

```
 B(I,X,Y,Z,D,C) =
  [b(i) : Bot |
    lkb : (clock(0) class(b(i), Bot)
           class(st(0), Station) (atloc(st(0), pt(2, 1)) @ 0)
           (atloc(b(I), pt(X,Y)) @ 0) (energy(b(I), 1.5e+1) @ 0)
           (myY(b(I),Z) @ 0) (myDir(b(I),D) @ 0)
 caution(b(I),C) @ 0),
     ckb : none,
     evs : (tick @ 1) ]
```

Thus `B(I,X,Y,Z,D,C)` is the state of a patrol bot at time 0 with identifier `I`, location `pt(X,Y)`, track `Z`, direction `D`, and caution `C`. The family of agent systems with one patrol bot parameterized by level of caution, C, and Wind, `Asys1(C,Wind)`, is given by

```
Asys1(C,Wind) =
{bound(200)
[eI | clock(0) class(b(0), Bot)
      (atloc(b(0), pt(0, 0)) @  0)  (energy(b(0), 15) @ 0)
      class(st(0), Station) (atloc(st(0), pt(2, 1)) @ 0)
      Wind  ]
 B(0,0,0,0,W,C)
}
```

The system `Asys1(C,Wind)` has a single patrol bot, with identifier `b(0)`, location `pt(0,0)`, moving west, with energy `15` and caution `C`. In the environment there is wind specification `Wind`. Note that the patrol bot will immediately turn and head east since it is at the western edge of the grid.

To watch the system run, we turn on printing of print attributes and rewrite for 20 steps. The following is a simplified version of what is printed. Following the `eAct` tag is the patrol bots location and energy at the end of the timeStep. Following the `doTask` tag is the patrol bot identifier, task, current time, and event set produced.

```
set print attribute on .
Maude> rew [20] updAkb(asys(200),b(0),caution(b(0),1.0) @ 0) .

eAct: clock(0) (atloc(b(0),pt(0,0)) @ 0) (energy(b(0),15) @ 0
timeStep: 0 ++ 1
doTask: b(0) ! tick time: 1 !! (tick @ 1) mv(b(0),E) @ 0 ; 1
eAct: clock(1) (atloc(b(0),pt(1,0)) @ 2) energy(b(0),14) @ 2
timeStep: 1 ++ 1
....
doTask: b(0) ! tick time: 4 !! (tick @ 1) mv(b(0),E) @ 0 ; 1
eAct: clock(4) (atloc(b(0),pt(4,0)) @ 5) (energy(b(0),11) @ 5)
timeStep: 4 ++ 1
**** the bot reversed direction
doTask: b(0) ! tick time: 5 !! (tick @ 1) mv(b(0),W) @ 0 ; 1
eAct: clock(5) (atloc(b(0),pt(3,0)) @ 6) (energy(b(0),10) @ 6)
timeStep: 5 ++ 1
.....
```

Initially there is nothing to do since the patrol bot has a task with delay 1. Then `doTask` and `timeStep` alternate, with the patrol bot moving east until it reaches the edge. At time 5 it reverses direction and starts moving west. By adjusting what is printed one can observe just variables of interest and look for unexpected behavior.

Now we look for critical configurations starting with instances of the one patrol bot family using the search command

```
search [1] Asys1(C,Wind) =>+ {criticalConf aconf} .
```

for different values of Wind and C.

Table 1 summarizes the search results. We see that in ideal conditions, minimal caution seems good enough to ensure no critical configurations are reached. Minimal caution works for 'modest' wind conditions (`N 17, S 13` or `N 11, S 7`).

Table 1. The columns `Wind` and `C` are values of the corresponding parameters. `Found?` indicates whether a critical configuration was found (in less that 200 time units). `States` is the number of states visited in the search, `Rewrites` is the number of rewrites, and `Duratin` is the search time in milliseconds. `N n, S m` stands for the wind items `(wind(N,n) @ 0) (wind(S,m) @ 0)`.

Wind	C	Found?	States	Rewrites	Duration (ms)
none	1.0	no	406	59948	184
none	4.0	no	406	59948	184
N 17, S 13	1.0	no	1049	460717	474
N 11, S 7	1.0	no	1195	524411	564
N 7, S 5	1.0	yes (1)	2216	926342	935
N 7, S 5	4.0	no	6605	3022120	3207

A critical configuration is found at state 2215 with Wind `N 7, S 5` and caution `1.0`. The environment component of the found critical configuration is the following:

```
[eI | clock(191) class(b(0), Bot) class(st(0), Station)
      (atloc(b(0), pt(2,0)) @ 191) (energy(b(0), 1.0) @ 191)
      (atloc(st(0), pt(2, 1)) @ 0)
(wind(N,7) @ 0) wind(S, 5) @ 0]
```

Thus more caution (`4.0`) is needed when wind effects are more frequent. Recall that the effect specified by `wind(dir,n) @ 0` is wind blowing in direction `dir` every `n` time units. Smaller values of `n` mean wind blowing more often and thus a greater chance to interfere with a patrol bots progress.

Caveat. The above searches were limited to time less than 200. This is already useful to find problems. In practice we may only need a given patrol instance to operate for a limited time, for example overnight or weekends. In this case bounded search is sufficient. If not, there are several ways to consider to extend the analysis. These will be discussed in Sect. 5.

The family of agent systems with two patrol bots parameterized by Wind and level of caution is given by

```
Asys2(C,Wind) =
{bound(200)
[eI | clock(0) class(b(0), Bot) class(b(1), Bot)
     (atloc(b(0), pt(0, 0)) @  0)  (energy(b(0), 15) @ 0)
     (atloc(b(1), pt(4, 2)) @  0)  (energy(b(1), 15) @ 0)
     class(st(0), Station) (atloc(st(0), pt(2, 1)) @ 0)
     Wind  ]
 B(0,0,0,0,W,C)
 B(1,4,2,2,E,C)
}
```

The system `Asys2(C,Wind)` extends `Asys1(C,Wind)` with a second patrol bot, with identifier `b(1)`, location `pt(4,2)`, moving east, with energy `15` and caution `C`. The environment is also extended with class, location and energy knowledge about the second patrol bot.

The Table 2 summarizes results of searching for critical configurations in instances of the two patrol bot system using the search command

```
search [1] Asys2(C,Wind) =>+ {criticalConf aconf},
```

for different values of Wind and C.

The critical configuration (`N 7, S 5, 4.0`) is found at state 113. The environment component of the found critical configuration is the following:

```
[eI | clock(15) class(b(0), Bot) class(b(1), Bot)
  (atloc(b(0), pt(2, 0)) @ 15) (energy(b(0), 19) @ 15)
  (atloc(b(1), pt(2, 2)) @ 15) (energy(b(1), 1.0) @ 15)
  class(st(0), Station) (atloc(st(0), pt(2, 1)) @ 0)
```

We can get an idea of how the critical configuration arises in the two patrol bot scenario by using the command

```
show path 113 .
```

which shows the sequence of states and rules applied leading to the critical configuration (state 113). The following shows the clock and patrol bot location and energy information in environment components of the last three states.

```
state 89, ASystem: {bound(187)
[eI
| clock(13)
(atloc(b(0), pt(2, 1)) @ 11) (energy(b(0), 15) @ 13)
(atloc(b(1), pt(3, 1)) @ 13) (energy(b(1), 3.0) @ 13) ]
```

Patrol bot `b(0)` is at the station, charging, and `b(1)` is next to the station, presumably intending to enter.

```
state 101, ASystem: {bound(186)
[eI
| clock(14)
(atloc(b(0), pt(2, 1)) @ 11) (energy(b(0), 20) @ 14)
(atloc(b(1), pt(3, 1)) @ 14) (energy(b(1), 2.0) @ 14) ]
```

Now `b(0)` is fully charged, and `b(1)` is still waiting, but it has used one energy unit trying to enter the station.

```
state 113, ASystem: {criticalConf
[eI
| clock(15)
(atloc(b(0), pt(2, 0)) @ 15) (energy(b(0), 19) @ 15)
(atloc(b(1), pt(2, 2)) @ 15) (energy(b(1), 1.0) @ 15) ]
```

`b(0)` has left the station. `b(1)` could enter at the next time, but it has used up its energy. It seems that the patrol bot is not paying attention to the fact that the station is occupied. Perhaps it should just wait until the station is free before trying to move there.

Table 2. The table columns are the same as Table 1 for one patrol bot.

Wind	C	Found?	States	Rewrites	Duration (ms)
none	4.0	no	853	669244	684
N 11, S 7	4.0	no	64154	51976162	65039
N 7, S 5	4.0	yes (1)	114	101739	122
N 7, S 5	6.0	no	243096	209875128	504088

4.5 Surveillance Drone Case Study

Now we look at a more complex case study involving a surveillance problem: there are P points of interest and we need to continually have recent pictures of the area around each point. This case study is inspired by a project to develop drones with the ability to monitor health in agricultural fields. In this project formal models are being used to explore different strategies for robustly meeting the recency requirement we formalize an abstract version of the problem as follows. The points are distributed on a grid with dimensions $x_{max} \times y_{max}$. N drones are deployed to take pictures. As for patrol bots, the drones use energy to fly from one place to another, and to take pictures, and they have maximum energy of e_{max}. There is a charging station in the center of the grid. In addition to its charging service, the station serves as a knowledge exchange cache, so drones can share information with each other by sharing with the station. Drones use soft-constraints, which take into account the drone's position, energy, and picture status of the points, to rank their actions. They may perform any one of the best ranked actions. We use M to denote the maximal acceptable age of a picture. Thus a critical configuration is one in which a drone runs out of energy, or the latest picture at some point is older than M.

As for the patrol bot system, we searched for critical configurations. Here we used a time bound of $n = 4 \times M$. The search results are summarized in Table 3. We varied M and the maximum energy capacity of drones e_{max} (instead of varying caution).

Note that even when considering a large grid (20×20) and three drone, search finds critical configurations or covers the full bounded search space quite quickly (less than a minute). As expected the number of states and time to

Table 3. N is the number of drones, P the number of points of interest, $x_{max} \times y_{max}$ the size of the grid, M the time limit for photos, and e_{max} the maximum energy capacity of each drone. We measured st and t, which are, respectively, the number of states and time in seconds until finding a critical configuration if F (for fail), or until searching all traces with exactly $4 \times M$ time steps if S (for success, no critical configuration before the time bound is reached).

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

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

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

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

search increases moderately with the increase of the number of drones and size of grid.

Although abstract, the surveillance drone model can help specifiers to decide how many drones to use and with which energy capacities. For example, in Exp 3, drones required a great deal of energy, namely 500 energy units. Adding an additional drone, Exp 4, reduced the energy needed to 150 energy units.

5 Conclusion and Future Perspectives

We have described a framework for modeling and reasoning about cyber-physical agent systems using executable models specified in the Maude rewriting logic language. Agents coordinate by sharing knowledge, and their behavior is specified by soft constraint problems (SCPs). Physical state and an agents perception of the state are modeled separately. These features are intended to help specify agents with some robustness, and to allow reasoning about agents that have only partial information about the system state, and about operation in an unpredictable environment. The soft agent framework does not provide methods for deciding what valuation domains and functions to use in defining SCPs. It does provide tools for formal exploration of parameter settings, weighting of valuation functions, degrees of caution, and so on.

The notion of soft-agent system is very similar to the notion of *ensemble* that emerged from the Interlink project [29] and that has been a central theme of the ASCENS (Autonomic Service-Component Ensembles) project [5]. In [30] a mathematical system model for ensembles is presented. Similar to soft agents, the mathematical model treats both cyber and physical aspects of a system. A notion of fitness is defined that supports reasoning about level of satisfaction. Adaptability is also treated. In contrast to the soft-agent framework which provides an executable model, the system model for ensembles is denotational. The two approaches are both compatible and complementary and there is intriguing potential for future extensions that could lead to a very expressive framework supporting both high-level specification and concrete design methodologies.

Soft Constraint Automata (SCA) is another approach to specifying soft agent systems [24]. It is an inherently compositional approach. Agents are composed from SCA for different aspects of their behavior, systems are composed from agent SCAs. The environment is treated as an agent, and it too is composed from smaller parts. Future plans include defining maps between SCAs and agent systems specified in the soft agent framework to be able take advantage of the benefits of each approach.

The soft agent framework presented here is just the beginning. In the following we discuss some of the remaining challenges and future directions.

We proposed agents that decide what to do by locally solving soft constraint problems (SCPs) and showed that this works in some simple cases. But, how much can be done usefully with local SCPs? When is higher level planning and coordination needed? Some specific questions to study include:

- Under what conditions are the local solutions good enough?
- Under what conditions would it not be possible?
- How much knowledge is needed for satisfactory solution/behavior? For example, for sufficient consensus.
- What frequency of decision making is needed so that local solutions are safe and effective?

Another challenge is compositional reasoning. Can we reason separately about different concerns by abstracting the rest of the system? Methods are needed to derive suitable decompositions and abstractions. We expect compositional approach based on Soft Constraint Automata [24] to lead to methods for compositional reasoning.

Since our objective is to reason about both cyber and physical aspects of a system, it is important to be able to have models that involve dense time. One of the reasons for the way time steps are formalized in the soft agent framework is to be able to specify actions using continuous functions (applied for some duration). We imagine that tasks will happen in discrete time, and that agents only observe continuously changing situations at discrete times. There are many details to worry about, including frequency of observation and frequency of adjusting controls, not to mention more complex interactions between elements of a model of the physical state. Importantly also, what are the right abstractions that manage complexity while remaining sufficiently faithful.

We showed how execution and bounded search provide simple tools for analyzing a soft agent system. But the guarantees provided are quite limited. One way to expand the analysis capability is to develop methods based on symbolic execution. Here large parts of a system are represented by variables, possibly subject to constraints. Search is carried out by unifying rules with symbolic states rather than matching rules to concrete states (this is called *narrowing*). If there are constraints, they accumulate and can be checked for satisfiability to prune impossible branches while avoiding the need to enumerate solutions. In this way whole families of systems can be analyzed at once. Backwards narrowing is another approach. Here one starts with a pattern representing a critical configuration and applies rules backwards to see if an initial state can be reached. If not, no instance of the critical configuration pattern is reachable. This approach has been used successfully in the MaudeNPA protocol analysis tool [31]. Another possibility is to identify systems where one can apply timeshift abstraction. Here one defines an equivalence relation on states at different times and attempts to show that modulo equivalence there are a finite number of states that repeat. This is a form of bounded induction. Another important direction is to develop efficient algorithms for model checking of soft constraint automata, taking advantage of compositionality.

Finally, an important issue that we have not mentioned is security. In the context of soft agents security has many aspects and subtilties. There are issues of trust or confidence of an agent in knowledge received, from another agent, or by reading its sensors. Security protocols may have space or time aspects. Like everything else, we probably want soft notions for security guarantees: trust for

some purpose, secret for some small amount of time, and so on. It is clear that the soft agent framework needs to provide support for managing and reasoning about security. What are the right mechanisms? How much security should be built into knowledge sharing? How can we balance imposition of security precautions and need for agility, and open systems.

References

1. Robots that fly and cooperate. TED talk (2015). Accessed 07 March 2016
2. Das, J., Cross, G., Qu, C., Makineni, A., Tokekar, P., Mulgaonkar, Y., Kumar, V.: Devices, systems, and methods for automated monitoring enabling precision agriculture. In: IEEE International Conference on Automation Science and Engineering (2015)
3. Vijay Kumar lab. Accessed 11 March 2016
4. Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.): Software Engineering for Collective Autonomic Systems. The ASCENS Approach. LNCS, vol. 8998. Springer, Switzerland (2015)
5. Ascens: Autonomic service-component ensembles. Accessed 15 November 2014
6. Choi, J.-S., McCarthy, T., Kim, M., Stehr, M.-O.: Adaptive wireless networks as an example of declarative fractionated systems. In: Stojmenovic, I., Cheng, Z., Guo, S. (eds.) MOBIQUITOUS 2013. LNICST, vol. 131, pp. 549–563. Springer, Heidelberg (2014)
7. Kim, M., Stehr, M.O., Talcott, C.: A distributed logic for networked cyber-physical systems. In: Arbab, F., Sirjani, M. (eds.) FSEN 2011. LNCS, vol. 7141, pp. 190–205. Springer, Heidelberg (2012)
8. Stehr, M.-O., Talcott, C., Rushby, J., Lincoln, P., Kim, M., Cheung, S., Poggio, A.: Fractionated software for networked cyber-physical systems: research directions and long-term vision. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 110–143. Springer, Heidelberg (2011)
9. Networked cyber physical systems. Accessed 11 March 2016
10. Drone swarms: The buzz of the future. Accessed 08 March 2016
11. Knightscope. Accessed 11 March 2016
12. Liquid robotics. Accessed 11 March 2016
13. Why BNSF railway is using drones to inspect thousands of miles of rail lines. Accessed 11 March 2016
14. Dantas, Y.G., Nigam, V., Fonseca, I.E.: A selective defense for application layer ddos attacks. In: SI-EISIC (2014)
15. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
16. Wirsing, M., Denker, G., Talcott, C., Poggio, A., Briesemeister, L.: A rewriting logic framework for soft constraints. In: Sixth International Workshop on Rewriting Logic and Its Applications (WRLA 2006). Electronic Notes in Theoretical Computer Science. Elsevier (2006)
17. Hölzl, M., Meier, M., Wirsing, M.: Which soft constraints do you prefer? In: Seventh International Workshop on Rewriting Logic and Its Applications (WRLA 2008). Electronic Notes in Theoretical Computer Science. Elsevier (2008)

18. Gadducci, F., Hölzl, M., Monreale, G.V., Wirsing, M.: Soft constraints for lexicographic orders. In: Castro, F., Gelbukh, A., González, M. (eds.) MICAI 2013, Part I. LNCS, vol. 8265, pp. 68–79. Springer, Heidelberg (2013)
19. Arbab, F., Santini, F.: Preference and similarity-based behavioral discovery of services. In: ter Beek, M.H., Lohmann, N. (eds.) WS-FM 2012. LNCS, vol. 7843, pp. 118–133. Springer, Heidelberg (2013)
20. Kim, M., Stehr, M.-O., Talcott, C.L.: A distributed logic for networked cyber-physical systems. Sci. Comput. Program. **78**(12), 2453–2467 (2013)
21. Choi, J.S., McCarthy, T., Yadav, M., Kim, M., Talcott, C., Gressier-Soudan, E.: Application patterns for cyber-physical systems. In: Cyber-Physical Systems Networks and Applications (2013)
22. Stehr, M.-O., Kim, M., Talcott, C.: Partially ordered knowledge sharing and fractionated systems in the context of other models for distributed computing. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 402–433. Springer, Heidelberg (2014)
23. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of real-time maude. High. Order Symbolic Comput. **20**(1–2), 161–196 (2007)
24. Kappé, T., Arbab, F., Talcott, C.: A compositional framework for preference-aware agents (March 2016, submitted)
25. Nielson, H.R., Nielson, F., Vigo, R.: A calculus for quality. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 188–204. Springer, Heidelberg (2013)
26. Nielson, H.R., Nielson, F.: Safety versus security in the quality calculus. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 285–303. Springer, Heidelberg (2013)
27. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoret. Comput. Sci. **96**(1), 73–155 (1992)
28. The maude system. Accessed 15 November 2014
29. Hölzl, M., Rauschmayer, A., Wirsing, M.: Engineering of software-intensive systems: state of the art and research challenges. In: Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.) Software-Intensive Systems. LNCS, vol. 5380, pp. 1–44. Springer, Heidelberg (2008)
30. Hölzl, M., Wirsing, M.: Towards a system model for ensembles. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 241–261. Springer, Heidelberg (2011)
31. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: cryptographic protocol analysis modulo equational properties. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2009. LNCS, vol. 5705, pp. 1–50. Springer, Heidelberg (2009)
32. Dantas, Y.G., Lemos, M.O.O., Fonseca, I.E., Nigam, V.: Formal specification and verification of a selective defense for TDoS attacks. In: Lucanu, D. (ed.) Workshop on Rewriting Logic and Applications (2016)