Soft Agents: Exploring Soft Constraints to Model Robust Adaptive Distributed Cyber-Physical Agent Systems

Carolyn Talcott[1], Farhad Arbab[2], and Maneesh Yadav[1]

[1] SRI International, Menlo Park, CA 94025, USA
{carolyn.talcott,maneesh.yadav}@sri.com
[2] CWI Amsterdam, The Netherlands
Farhad.Arbab@cwi.nl

Abstract. We are interested in principles for designing and building open distributed systems consisting of multiple cyber-physical agents, specifically, where a coherent global view is unattainable and timely consensus is impossible. Such agents attempt to contribute to a system goal by making local decisions to sense and effect their environment based on local information. In this paper we propose a model, formalized in the Maude rewriting logic system, that allows experimenting with and reasoning about designs of such systems. Features of the model include communication via sharing of partially ordered knowledge, making explicit the physical state as well as the cyber perception of this state, and the use of a notion of soft constraints developed by Martin Wirsing and his team to specify agent behavior. The paper begins with a discussion of desiderata for such models and concludes with a small case study to illustrate the use of the modeling framework.

With Best Wishes to Martin Wirsing

This work has roots in the joint work with Martin on the PAGODA project and soft constraint solving[1], carried out when Martin spent a sabbatical at SRI in 2005. This work was continued by Martin, Max Meier and Matthias Hölzl [2,3], leading to a new notion of soft constraints that seems very well suited to the world of cyber-physical agents. The present work also builds on many discussions over the last few years on the challenges of cyber-physical systems, including several Interlink workshops [4] lead by Martin.

It has been a great privilege and pleasure to know Martin, to have the opportunity to work together from time to time, and to exchange ideas as our paths have crossed over the years. I look forward to many more years of exchanging ideas and tackling challenging problems.

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

R. De Nicola and R. Hennicker (Eds.): Wirsing Festschrift, LNCS 8950, pp. 273–290, 2015.

such as large collections of automated vehicles, but also harnessing the underlying communication 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). At the time of writing, 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. The increasingly impressive capabilities of such platforms have led to autonomous cyber-physical systems that implement realtime methods for computer vision[5], machine learning[6] and computational aerodynamics[7]. Technology has progressed to the point that many of these capabilities can be easily applied by non-specialists.

The disparity between the growing sophistication of cyber-physical agents and practical, scalable methods that autonomously coordinate agent collectives (without central control) is clear from the lack of widely available frameworks. There are very few practical tools available to non-specialists that would enable them to specify joint goals across a cyber-physical agent collectives in a way that is robust to a dynamic environment.

The specification of goals and constraints for agent collectives broadly falls under a number of problem representations that have been long explored, these include Distributed Constraint Satisfaction Problems [8], Distributed Control Optimization Problems [9], Distributed Continual Planning [10], Multi-Agent Planning [11] and multi-agent coordination[12]. Not all methods that we have surveyed in the literature have proven correctness or bounds, but amongst those that have, coordination is clearly difficult since all methods that we are aware of are exponential in the number of messages sent (with the exception of DPOP[13], where the number of messages is kept linear at the cost of exponential memory usage).

Most methods focus on *distributed* systems, but practical distributed systems are often further complicated from unexpected changes in goals, (partial) agent failure and delays in agent communication. In the context of such complicating factors, some coordination methods will suffer (unbounded) delays in performing actions towards a specified goal (e.g., halt during consensus formation), when it would make eminent sense for the agents to begin *doing something* towards achieving their goal. While many of the problem representations that have explored distributed systems have carefully considered dynamic environments and failure, we use the term *fractionated* to emphasize these aspects.

Our intent is to address the design, prototyping and eventually implementation of systems which we call *Fractionated Cyber-Physical Systems* (FCPS) [14,15]. FCPS are distributed systems composed of many small cyber-physical agents that must act using only local knowledge in the context of an uncertain environment, communication delays as well as agent failure/replacement/addition. Agents in FCPS interact by sharing knowledge that is gained by sensing and by reasoning. We are particularly interested in principles for designing FCPS in which desired cooperation/coordination emerges from local behaviors, under practical conditions.

FCPS promise robustness and fault tolerance using many small entities such that no specific individual is critical. Entities can come and go without disrupting the system, as long as the needed functionality is sufficiently represented. Defective, worn out, or out-of-date entities can be easily replaced by fresh, improved versions. The term "fractionated" was originally coined to describe replacing small sets of multifunctional space satellites with an collective of smaller more specialized "nanosats" that could be launched cheaply and easily replaced, providing a resilient system capable of complex functionality at a lower overall cost[16]. We suggest that this notion has become much more relevant with the advent of ubiquitous mobile computing and applicable to "down to earth" problems such as package delivery.

Towards these goals, we propose a framework we call *Soft Agents* and describe a prototype implementation in the Maude rewriting logic system [17] along with a small package delivery case-study to illustrate the ideas.

The notion of fractionated cyber-physical systems is very similar to the notion of *ensemble* that emerged from the Interlink project [4,18] and that has been a central theme of the ASCENS (Autonomic Service-Component Ensembles) project [19]. In [20] a mathematical system model for ensembles is presented. Similar to FCPS and 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 approached are both compatible and complementary and joining them could lead to a very expressive framework supporting both high-level specification and concrete design methodologies.

The soft agents framework combines ideas from several previous works: the use of soft constraints [1,2,3] and soft constraint automata [21] for specifying robust and adaptive behavior; partially ordered knowledge sharing for communication in disrupted environments [14,22,23,15], and the Real-time Maude approach to modeling timed systems [24].

Soft constraints allow composition in multiple dimensions, including different concerns for a given agent, and composition of constraints for multiple agents. In [2] 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 [3] partially ordered valuation structures are explored to provide operators for combination of constraints for different features that respects the relative importance of the features.

Given local constraints, a global theoretical model can be formed as a cross product, that considers all inter-leavings of actions of individual agents. This is generally an infeasibly complex problem to solve. We propose solving the complex problem by concurrent distributed solution of simpler local problems. This leads us to study questions such as

- 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?
- What frequency of decision making is needed so that local solutions are safe and effective?

Plan. Section 2 discusses desiderata for a framework for FCPS. Section 3 presents the proposed framework and its formalization in Maude. Section 4 illustrates the application of soft-agents to a simple autonomous drone packet deliver system. Section 5 summarizes and discusses future directions.

2 Desiderata for Soft Agents

CPS 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/tools for system designs that achieve adaptive, robust functionality using diversity, redundancy and probabilistic behavior.

The primary desiderata for our FCPS 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 an FCPS agent 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 *satisficing consensus*: for any agent, consensus is satisficed 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 POKS knowledge dissemination framework underlying an FCPS takes care of agreeing on state to the degree possible. In a quiescent connected situation all agents will eventually have the same knowledge base. As communication improves, an FCPS 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 of an FCPS to consider concerns safety and liveness: 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.

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

– 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 [2] 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. Rather than looking for 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.

3 Soft Agent Model Formalized in Maude

3.1 Networked Cyber-Physical Systems and Partially-Ordered Knowledge Sharing

To motivate the formal model we give a brief overview of our Networked cyber-physical systems (NCPS) framework. The theoretical foundation is a distributed computing model based on sharing knowledge that is partially ordered (POKS) [22,14,15]. An NCPS is a collection of cyber-physical agents (CPAs or simply agents) with diverse capabilities and resource limitations. They may cooperate and/or compete to achieve some global or local goal. An agent can communicate directly with connected peers to share knowledge. Information propagates opportunistically when connections arise. Communication, as well as an agent's own sensors, may update the agent's local knowledge base. A CPA must function somewhat autonomously, making decisions based on local information. It should function safely even in absence of communication and should be capable of a wide spectrum of operation between autonomy and consensus/cooperation to adapt to resource constraints and disruptions in communication. Interpreting knowledge items as facts or goals enables declarative specification of sensing and control. Conflicts between information received from various peers, through various channels, and/or local sensors, acquired/received at different times, need to be resolved. The partial order on

knowledge provides a mechanism for revision, control, conflict resolution and consensus under suitable conditions.

Soft agents are based on the NCPS framework with two additional features: formulation of an agent's goals as soft constraint problems and representation of an agent's physical state separated from its knowledge. Actions are carried out based on the physical state, and an agent makes decisions based on its local knowledge. In the following we describe the formalization of Soft Agents in the Maude rewriting logic language [17].

3.2 Soft Agents in Maude

A Maude cyber-physical soft-constraint system is an executable model with two parts: (1) the application independent modules (SOFT-AGENTS, SOFT-AGENT-RULES) and (2) modules specifying capabilities and behavior of the agents by defining the abstract functions used in the rewrite rules.

The following is a brief summary of the data structures and behavior rules used to specify the framework, including the functions that must be defined for each application. We present fragments of Maude code followed by informal discussion to clarify and augment the fragments. [2]

Knowledge. We use knowledge to represent properties of the world (the environment perspective) as well as an agents local view. An agent can receive or post knowledge. The environment, represented by rewrite rules, provides sensor information to agents.

```
sorts PKItem TKItem KItem  Info .
subsort PKItem TKItem < KItem .
op _@_ : Info Nat -> TKItem .
```

Knowledge is represented as a set of knowledge items (sort `KItem`), and comes in two flavors: persistent (sort `PKItem`) and ephemeral/temporary (sort `TKItem`). Persistent knowledge is knowledge that doesn't change over time, such as the class of an agent, size of a packet or capacity of a drone. Ephemeral knowledge concerns things that are expected to change over time such as sensor information, delivered by the environment, (received) logical and shared state knowledge (received and/or posted). Ephemeral knowledge is constructed from the underlying information (sort `Info`) and a timestamp (`info @ t`).

```
op _<<_ : KItem KItem -> Bool .  *** knowledge partial order
eq kitem << kitem' = false [owise] . *** not ordered default
```

Knowledge is partially ordered (<<) providing a mechanism to discard knowledge that is no longer valid/relevant. This allows an agent to deal with situations where, for example, newer state knowledge is available, or knowledge representing a goal is no longer valid because the goal has been satisfied or timed out. The equation with the `[owise]`

[2] Maude specifications consist of sort and subsort declarations, specifying the data type hierarchy, function and constant declarations (keyword `op`) giving argument and result sorts, equations (keyword `eq`) used to define functions, and rewrite rules `rl[rulename]: lhs => rhs.` `---` and `***` precede comments.

label says that the default is that two items are not ordered. It will be used if no other equation for the relation << matches.

An agent is modeled by a structure of the form

```
[id : class | envkb | localkb | cachedkb | events ]
```

where `id` is the agent's identity and `class` is the agent's class. The terms `envkb`, `localkb`, and `cachekb` denote knowledge bases—sets of knowledge items, where `envkb` contains facts representing an agent's physical state. In mobile settings it will include location and may include energy or fuel level, load, weight, etc.. The agent doesn't see `envkb` directly, only via sensor readings which maybe posted automatically or upon request. The knowledge items in `localkb` contain the agent's perception of its state and other knowledge which could include mode, plans, and knowledge posted by other agents. The knowledge items in `cachedkb` form the local cache of knowledge, used by the environment to implement knowledge sharing. It is not directly visible by the agent, although the agent has had the opportunity to process the knowledge in the cache. An agent system evolves by application of event handling rules. Rules for different types of event use information from different knowledge bases as appropriate.

Events, actions and tasks. *Events* include those that an agent handles and those handled by the environment. There are four classes of event:

- received knowledge such as local sensor information or shared knowledge, handled by the agent
- tasks scheduled by the agent, possibly with delay, to be handled by the agent
- actions to execute posted by agents, possibly with delay, handled by the environment (using rewrite rules that reflect the model physics)
- knowledge posted by an agent, handled by the communication system rewrite rules

```
sorts IEvent DEvent Event .
subsort  IEvent DEvent < Event .  --- immediate/delayed events
sorts Action Task ActTask .       --- delayed event body
subsort Action Task < ActTask .
```

Events are classified as immediate (sort `IEvent`) or delayed (sort `DEvent`). Delayed events are actions or tasks with a timestamp. The following are the event constructors.

```
op _@_ : ActTask Nat -> DEvent .
op rcv : KB -> IEvent [ctor] .        *** receive event
op post : InfoSet -> IEvent  [ctor] . *** posting information
op done : Action Bool -> Info  [ctor] . *** action status
op tick : -> Task .                    *** built in task
```

Time For initial studies we assume a global clock. Agents can run at different speeds by scheduling actions with different delays. In the current simple model only tasks and actions cause time to pass. Other events are handled instantaneously. An agent's behavior rules can cause delay in handling received knowledge by posting tasks.

3.3 Rules

An agent system has the form

```
{ aconf clock(t) }
```

where `aconf` is a multiset of agents, assumed to have distinct identifiers, and `clock(t)` is the global clock. There are five rewrite rules for describing how an agent system evolves over time. An agent may be reactive, and only respond when new knowledge is received, or agents may be proactive, scheduling tasks rather than waiting for input.

Rules for communication, posting and receiving knowledge, and handling scheduled tasks operate on a local part of the agent configuration. The rule for executing actions and the rule for passing time must capture the whole system.

Knowledge sharing. Notation convention: In the following we use `id` (and decorated variants) to range over agent identifiers, `cl` to range over agent classes, `ekb` for environment knowledge bases, `lkb` for local knowledge bases, `ckb` for cache knowledge bases, `evs` for event sets, and `rcvk` for received knowledge bases. For example, in the following code `ekb1` is the environment knowledge base of the agent with identity `id1`, and `evs1` is the set of pending events for this agent.

```
crl[KnowledgeSharing]:
[id1 : cl1 | ekb1 | lkb1 | ckb1 | evs1 ]
[id2 : cl2 | ekb2 | lkb2 | ckb2 | evs2]
=>
[id1 : cl1 | ekb1 | lkb1 | ckb11 |
       evs1 (if rcvk1 == none then none else rcv(rcvk1) fi)]
[id2 : cl2 | ekb2 | lkb2 | ckb21 |
       evs2 (if rcvk2 == none then none else rcv(rcvk2) fi)]
if inContact(id1,ekb1,id2,ekb2)
/\ {ckb11, rcvk1} := share(ckb2,ckb1,none) --- ckb2 to ckb1
/\ {ckb21, rcvk2} := share(ckb1,ckb2,none) --- ckb1 to ckb2
/\ rcvk2 rcvk1 =/= none .
```

Knowledge is propagated to neighbors upon contact. The definition of contact is part of each specific model and a model could have several different forms of contact. This is formalized by declaring a function `inContact(id1,ekb1,id2,ekb2)` that takes two agent identifiers and their corresponding environment knowledge bases and returns a boolean. A simple notion of contact is for two agents to be within a given distance of each other. Upon contact two agents fully exchange knowledge in their cache knowledge base and each agent is notified (via a receive event) of any new knowledge obtained. The function `share(ckb2,ckb1,none)` returns `ckb11,rcvk1`, where `ckb11` is `ckb1` updated with knowledge from `ckb2` that is not present or subsumed/replaced by knowledge already in `ckb1`, and `rcvk1` is the set of knowledge items newly added, used to notify the agent via the `rcv(rcvk1)` event.

Future work could limit the number of knowledge items that can be exchanged upon each contact, controlled by a policy or by the physics. Other properties of channels could be modeled as well, such as one way channels.

Posted Knowledge.

```
crl[post]:
[a : cl | ekb | lkb | ckb | post(iset) evs ] clock(t)
=>
[a : cl | ekb | lkb' | ckb' | evs ]   clock(t)
if kb := tstamp(iset,t)
/\ ckb' := addK(ckb,kb)
/\ lkb' := addK(lkb,kb) .
```

Posted knowledge is time stamped (`tstamp(iset,t)`) with the current time and added to the cached knowledge base and the local knowledge base. The function `addK` adds knowledge items from its second argument, `kb1`, to its first argument, `kb0`, that are not less than in the partial order to knowledge already present in `kb0`. It also removes elements of `kb0` that are less in the partial order than a newly added item.

Receiving Knowledge.

```
crl[rcv]:
[id : cl | ekb | lkb | ckb | rcv(rcvk) evs ] clock(t)
=>
[id : cl | ekb | lkb' | ckb | evs fixTime(evs',t) ]   clock(t)
if {lkb', evs'} := handle(cl,id, lkb,rcvk) .
```

Agents have class specific procedures for handling new knowledge. These procedures specify how the local knowledge base is updated, possibly raising new events to schedule. This is formalized by the function

```
handle(class,id,lkb,rcvkb) = {lkb1,evs1}
```

where `lkb` is the current local knowledge base, `lkb1` is the updated local knowledge base, and `evs1` is the possibly empty set of events to schedule.

Tasks. Tasks provide a mechanism for an agent to control scheduling its activity.

```
crl[doTask]:
[id : cl | ekb | lkb | ckb | (task @ t) evs ] clock(t')
=>
[id : cl | ekb | lkb | ckb | evs fixTime(ev,t')]  clock(t')
if t <= t'
/\ ev evs' := doTask(cl, id, task,lkb) .
```

The event `task @ t` expresses that the agent plans to carry out `task` at time `t`. The task can be carried out if its time stamp is not greater than the current time. The task handling function, `doTask(class,id,lkb,task)`, returns a set of alternative actions. `task` is the task to be carried out, `lkb` is the agents local knowledge base. One of the alternative actions is chosen non deterministically and added to the agents event set. The pattern `ev evs'` models the selection of an event from a non-empty set of events.

The framework provides a generic task `tick` which an agent can use as a mechanism to periodically check the local state and decide on possible actions, if any. In our applications the agent solves a soft constraint problem for this purpose, but the framework allows other methods of deciding on actions.

Actions. For each agent class there is a (possibly empty) set of actions that its instances can perform. The event `act @ t` expresses that the agent intends to execute the action `act` at time `t`. For example the agent could move to a new location, press a button, pickup or drop an object.

```
crl[doAct]:
{[id : cl | ekb | lkb | ckb | (act @ t') evs ] clock(t) aconf}
=>
{ updateAConf([id : cl | ekb | lkb | ckb | evs ],{id,ekb',evs'})
  updateAConf( aconf,idkbevset)  clock(t) }
if t' <= t
/\ {id,ekb',evs'} idkbevset := doAction(cl,id,act,t,ekb,aconf) .
```

The environment has an action execution function, `doAction`, parameterized by agent class, formalizing the physics of the model. The physics may involve the state of the local environment, which is represented in the environment knowledge base of nearby agents. This is formalized by the abstract equation

```
doAction(class,id,act,t,ekb,aconf) = idekbevents
```

where `act` is the action to be executed, `ekb` is the agent's local environment knowledge base, and `aconf` contains the nearby agents. The variable `idekbevents` stands for a set of updates of the form `id,ekb',evs'`, one for the agent doing the action and one for any nearby agent affected by the action. `ekb'` is the new environment knowledge base and `evs'` is added to the existing event set.

Advancing time.

```
crl[timeStep]:
{ aconf clock(t) } => { aconf clock(ni) }
if ni := minTime(aconf,t)
/\ ni :: Nat  .
```

Following the real-time Maude approach [24], time passes if there is nothing else to do at the current time. The function `minTime` returns infinity (which will not satisfy the membership `ni :: Nat`) if there are any immediate events or any knowledge sharing enabled. Otherwise, it returns the smallest timestamp of a delayed action or task.

4 A Simple Packet Delivery System

The drones and packets simple package delivery problem was inspired by a small startup in San Francisco developing an instant package delivery/courier service. There is a service area; packets at various locations in the service area, that want to be at another location (called the destination); and drones (mobile agents capable of moving around) that can transport packets. Drones use energy when moving and there are charging stations where a drone can recharge. We start with a simple case where there is one kind of packet and one kind of drone. The service area is a grid, locations are points on the grid, some of these locations are charging stations.

Knowledge partial order. The partially ordered knowledge for drones and packets adds three kinds of knowledge items for packet state

```
dest(idp,loc)           *** the destination of packet pid
pickupOk(pid,id,b) @ t *** models a pickup light indicating
                       *** permission for drone id to pickup pid
delivered(pid) @ t        *** packet pid has been delivered
```

and one item for drone state

```
energy(id,e) @ t   *** drone id has energy reserve e
```

where we use `idp` and `id` as variables for packet and drone identifiers; `t, t0, t1` range over natural numbers representing discrete time; and `loc,l0,l1` range over locations. The partial order is given by the following equations.

```
ceq (atloc(id,l0) @ t0) << (atloc(id,l1) @ t1)
      = true if t0 < t1 .
ceq (energy(id,n0) @ t0) << (energy(id,n1) @ t1)
      = true if t0 < t1 .

*** once delivered, packet info disappears.
eq (atloc(idp,l0) @ t0) << (delivered(idp) @ t1) = true .
eq dest(idp,l0) << (delivered(idp) @ t1) = true .
eq (pickupOk(idp,id,b0) @ t0) << (delivered(idp) @ t1) = true .
eq class(idp,packet) << (delivered(idp) @ t1) = true .
```

The first two equations say that new information about energy or location of an agent replaces older information. The last four formalize the policy that once a packet is delivered, it is no longer part of the system.

Actions. Drones are proactive, periodically choosing an action and executing it. The possible drone actions are

- `mv(dir)`: moving in direction `dir` to an adjacent grid point, where `dir` is one of `N,S,E,W`.
- `charge`: charge one charge unit, if located at a charging station
- `pickup(idp)`: picking up packet `idp`, if permitted
- `drop(idp)`: dropping packet `idp`, if carrying `idp`

It is permitted for a drone to pickup a packet if they are co-located and the packet agrees to ride (it may prefer another drone). Packets are largely passive, they react to sensing the presence of a drone by indicating whether they will accept a ride from that drone. This is done by action `setPickUpOk(id,b)`, where `b` is a boolean. If `b` is *true* the pickup light should turn on, otherwise it turn off.

The system objective is that packets get delivered with minimal delay and minimal cost (drone energy). Less delay gives more satisfaction as does less energy consumption.

Choosing actions. As a first step, drones operate independently. A drone repeatedly chooses an allowed action to execute. The choice is formulated as a family of soft-constraint problems parameterized by the drone's local knowledge. This pro-activity of individual drones is insufficient by itself to ensure liveness of the system, there are two local criteria to consider, in designing the soft constraints.

(Safety) The drone should not run out of energy.
(Benefit) If there are packets needing transport, the drone should pick a packet according to some notion of benefit gained for work done and transport that packet.

We start with a simple drone behavior in which the drone will try to pick a packet that requires the least work to deliver, where work to deliver is a function of the distance to the packets destination, going via the packet's current location.

We use the definition of Soft Constraint Problem (SCP) presented in [2]. An SCP over a set of variables V with values in domain D consists of a tuple of grading functions

$$[(G_i, c_i)|1 \leq i \leq k]$$

together with a ranking structure

$$(R, [p_i|1 \leq i \leq k], I)$$

where each G_i is a monoid, c_i maps variable value tuples $\bar{d}$ to the domain of G_i, and p_i is an action of G_i on R ($p_i(g_i) : R \to R$). The solution is the set of variable value tuples with maximal rank.

$$maxS(SCP) = \{(\bar{d}, S(\bar{d})|S(\bar{d}) \text{ is maximal over } D^V\}$$

where $S(\bar{d}) = \bigcirc_{1 \leq i \leq k}(p_i(c_i((\bar{d}))(I)))$. Here $\bigcirc$ is the binary operation in the ranking structure, $*$ in our case.

In the independent drone problem there is one variable, the action, and actions are graded based on the state predicted to result from executing the action. Thus the drone soft constraint problem *DSCP* is a family of SCPs parameterized by the drone identifier and local knowledge base (the drones view of the system state). We use two grading functions one for the Benefit criteria, one for the Safety constraint. Safety is a crisp constraint – the value of an action is 0 if it leads to an unsafe situation and 1 otherwise. An action is deemed safe if in the resulting state the drone has sufficient energy to reach a charging station. Thus it should be provable that if a drone is initially in a safe condition, it will remain safe. (Although it may not be able to move.)

Benefit ranges from 0 to `maxBenefit`. The ranking function combines Safety and Benefit by multiplication, thus ensuring an unsafe action has rank 0.

$$\begin{aligned} &DCP(id, lkb) = \\ &\quad [(NatMax, Benefit(id, lkb)), \\ &\quad (ZeroOne, Safety(id, lkb)), \\ &\quad (NatMult, *, 1)] \end{aligned}$$

where $NatMax$ is the natural numbers ordered as usual taking max as the binary operator, and `maxBenefit` as the identity. $NatMult$ is the natural numbers with multiplication as the binary operator and 1 as the identity. $ZeroOne$ is like the boolean ordered monoid using $0, 1$ rather than booleans to support the multiplicative action.

Only moves perceived to be possible are considered. Thus `drop` won't be considered if the drone is not carrying a packet, and moves off the grid or to a point thought to be

occupied by another drone will not be considered. This is done just to simplify the grading functions, as such moves will be given a 0 grade.

The *Benefit* function is the key. It depends on the drone's mode: searching (for a packet) or carrying (a packet). In searching mode, if the drone is co-located with a packet the only action with a non-zero grade is to pickup the packet. Otherwise the grade is the max of 1 and the gain for delivering known packets, where the gain is the max possible cost minus the actual cost. The actual cost for delivering a given packet is the distance to the packets destination going via the packet's current location.

In carrying mode if the drone's location is the packet's destination then the only action with a non-zero grade is dropping the packet. Otherwise moves that decrease distance to destination are preferred. Charging gets a maximal grade if the drone is at a charging station and its energy is below the upper bound.

The `doTask` function for drones formalizes the above.

```
ceq doTask(drone,id,tick,lkb) = tstampA(bestActs,0)
if acts := droneActs(id,lkb)
/\ bestActs := selectMax(rankDroneActs(id,lkb,acts),1,none) .
```

The function `droneActs(id,lkb)` enumerates the allowed actions, and the function `tstampA(acts,n)` stamps each action in `acts` with delay `n` (`0` in the above equation). The function `selectMax(rankDroneActs(...)...)` selects the maximal solutions.

4.1 The doAction and Handle Functions

The `doAction` function specifies the physical effects of an action as updates on local environment knowledge bases and the perceived/sensed effects of the action by information in a `rcv` event. Part of the sensed effects of the agent executing an action `act` is the information item `done(act,b) @ t` where `b` is a boolean indicating success or failure, and `t` is the time at which the action was executed.

Drone actions. The drone move and charge actions effect only the drone state. For example the equation defining a move is

```
ceq doAction(drone,id,mv(dir),t,ekb,aconf) =
 {id, addK(ekb,mvkb), rcv((done(mv(dir),b) @ t) mvkb)}
if l0 := getLoc(id,ekb)
/\ l1 := doMv(l0,dir)
/\  e := getEnergy(id,ekb)
/\  b := not(occupied(l1,aconf)) and e > 1
/\ mvkb := (if b
            then mvInfo(id,l1,sd(e,1),t)
            else mvInfo(id,l0,e,t) fi) .
```

where `mvInfo` computes the new location and energy information items for the drone. The charging action succeeds if the drone is located at a charging station and the new information is the new energy level.

The `handle` function for drones for response to receiving action information adds the new information to the local knowledge base of the drone, schedules a `tick` at delay `droneDelay` and posts new location information if any.

The drone `pickup(idp)/drop(idp)` actions, if successful also affect the packet `idp`. In the case of a successful `pickup(idp)/drop(idp)` action the packet is added/removed from the drone's environment knowledge base. The packet environment knowledge base is updated with new packet location and the packet is notified of its new location. When the packet handles the new location information it will post this information, thus the drone local knowledge base will be updated as well. A `pickup(idp)` by a drone will fail if not co-located or if the packet environment knowledge base has the pickupOk light off (explicitly or by default). A `drop(idp)` fails if the packet is not carried by the drone (which should not happen in our simple setting). If `pickup` or `drop` fails then nothing changes (except that the time stamp on packet location may be updated).

Drones respond to received information that is not an action report by adding the new information to their local knowledge bases.

Packet actions. The only packet action is `setPickupOk(id,b)`. If the boolean `b` is `true`, the action turns on the pickupOk light for `id` in the packet's environment knowledge base. This is represented by the knowledge item `pickupOk(id,true) @ t` in the environment and local knowledge bases. When the boolean is `false`, the pickupOk light is turned off. This is represented by `pickupOk(id,false)` in the knowledge base, or by absence of a `pickupOk` fact. The packet is also notified of the `pickupOk` information. This action should succeed (unless the light is broken).

A packet that is delivered, `delivered(idp) @ t` in the local knowledge base, handles new information by ignoring it. A packet that is still active handles new self location information `atloc(idp,loc) @ t` according to whether `loc` is its destination or not. In the first case, the packet posts `delivered(idp)` which will remove other information about the packet as it propagates using the partial ordering. Otherwise the packet remembers the new location (adds it to the local knowledge bases) and also posts it.

A packet responds to new location for drone `id` by scheduling a `setPickupOk` action for `id` with boolean `true` if the drone becomes co-located. In either case it also remembers the new location. In other cases a packet simply remembers the new information.

4.2 Experiments

We now describe the results of a few simple experiments in the setting of the drones and packets model. The are number of parameters to be set. For this set of experiments, these are fixed as follows.

```
eq commDistance = 2 .        *** upper bound on contact distance
eq gridX = 10 .              *** grid dimensions
eq gridY = 10 .
eq chargeLocs = pt(5,5) .    *** one charging station at 5,5
eq maxBenefit = 20 .
```

```
eq maxCharge = 25 .          *** stop charging when full
eq costMv = 1 .              *** energy used in a move
eq chargeUnit = 5 .          *** energy gained per charge action
eq droneDelay = 2 .          *** delay on tick action
```

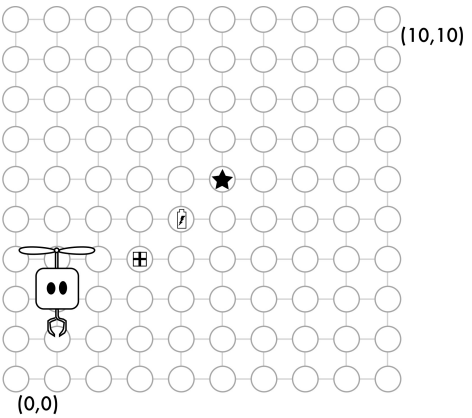

Fig. 1. A depiction of the location grid with the drone at (2,2). packet at (4,4), charging station at (5,5) and destination at (6,6).

Here is a simple ASystem, AS0, with one drone, d(0), at (2,2) with 10 energy units, and one packet `p(0)` at (4,4) with destination (6,6).

```
AS0 =
{clock(1)
[d(0) : drone
    | (atloc(d(0),pt(2,2)) @ 0)(energy(d(0),10) @ 0)class(d(0),drone)
    | (atloc(d(0), pt(2, 2)) @ 0) (atloc(p(0), pt(4, 4)) @ 0)
    (energy(d(0),10) @ 0) class(d(0),drone) class(p(0), packet)
    dest(p(0), pt(6, 6))
    | none
    | tick @ 1]
[p(0) : packet
    | (atloc(p(0),pt(4,4)) @ 0) class(p(0),packet)
    | (atloc(p(0),pt(4,4)) @ 0) class(d(0),drone) class(p(0),packet)
    dest(p(0),pt(6,6))
    | none
    | none]}
```

Rewriting this configuration with a limit of 100 results in a system state in which `p(0)` was delivered at time 25, `(delivered(p(0)) @ 25)` is in the local KB of `p(0)`. Searching starting with AS0 for a state with `p(0)` delivered at some time using the search pattern

```
{[ p(0) : packet | ekb:KB | lkb:KB  (delivered(p(0)) @ t:Nat)
                 | ckb:KB | evs:EventSet] ac:Conf }
```

finds a solution at state 201. If the drones initial energy is only 5, it will not move, as it is in an unsafe situation. If the initial energy is 8, the drone will recharge before it drops the packet.

As a next example we use an initial state, AS1, with one drone, `d(0)`, at (2,2) with 10 energy units, and two packets `p(0)` at (2,6) with destination (4,6) and `p(1)` at (6,2) with destination (6,6). Searching from AS1 for a state in which `p(0)` is delivered finds a solution at state 66, and searching for a state in which `p(1)` is delivered finds a solution at state 459. It the latter case `p(0)` has also been delivered and the drone has recharged.

Finally, we consider an initial state, AS2, with two drones, `d(0)`, at (2,2) with 10 energy units and `d(1)`, at (7,7) with 10 energy units, and two packets `p(0)` at (2,6) with destination (4,6) and `p(1)` at (6,2) with destination (6,6). Rewriting leads to a state with both packets delivered: `p(0)` at time 17, `p(1)` at 47. Searching from AS2 for a state in which `p(0)` is delivered finds a solution at state 21276. Searching from AS2 for a state in which `p(1)` is delivered finds a solution at state 576171. In this solution `p(0)` was delivered at time 15 by `d(0)` and `p(1)` was delivered at time 41 by `d(1)`.

Note that it is possible, given an upper bound on the energy capacity that the drone can not reach a packet to deliver it. This is a design flaw that could hopefully be discovered by formal modeling. This is a topic for future investigation.

5 Conclusions and Future Directions

We have presented a framework for modeling fractionated cyber-physical systems called Soft Agents, and a prototype using the Maude rewriting logic system. Three key features of the framework are the partially ordered knowledge model of communication; soft constraint problems for specifying robust, adaptive agent behavior; and explicit representation of the physical state of an agent as well as its knowledge state. Use of the framework was illustrated with a simple drones and packets case study, showing how soft constraints can be used, and the use of partial ordering on knowledge to keep the knowledge state relevant and simple.

The soft agents framework supports, but does not enforce, liveness as discussed in section 2. Agents can control how often to check whether some action is needed, and agents can be notified about change in their local state. It is up to the system designer to use these capabilities to achieve the desired/necessary liveness.

The present work is just a first step. It lays a foundation for further exploration and experimentation to understand the tradeoffs and the nature of emerging behavior. One point of being able to experiment is to guide attempts to prove general properties.

The drones and packets case study can be complicated in many ways to explore principles for defining soft constraints. For example, packets can have different weights, require special accommodation such as refrigeration, and drones can have different load capacity (with/without refrigeration), speed, energy consumption, etc. New packets can appear as the system evolves. Packets can post ratings of their ride — giving other packets a reason to refuse a ride from a low rated drone, and giving drones motivation to provide better service. In addition, one can also consider local clocks, rather than a global clock, to study different forms of synchronization.

In the presented case study, drones just planned one step ahead and didn't try to account for actions of other drones. A next challenge is to consider multistep planning

both independently and with knowledge of what other agents are planning. An agent would make a plan (a set of reasonable plans) to achieve some objective, and start executing a plan that it (locally) deems best. It would re-evaluate under certain conditions, for example at every step, or when the next step is not possible. One possible approach would be the reflecting planning approach that was used in a disruption tolerant networking project [27]. The idea is to reflect a model of the system based on the local knowledge base to the meta-level, search for paths to a goal or subgoal, and use results as the space of solutions for soft constraint solving. This would not generally be efficient but might lead to useful insights.

In our simple example with one variable, finding the maximal solutions to a constraint problem is simple. With multi-step plans and consideration of multiple agents methods for efficient solution will be needed. There are some suggestions in [2]. Methods to reuse partial plans could also be useful.

As indicated by the number of states to be searched to find a solution for both packets being delivered starting from the initial state, AS2, with 2 drones and 2 packets, work is needed to scale the analysis. Interestingly, simple rewriting finds a solution, although it may not be the best one, and it is very fast even for AS2. Methods to reduce the interleaving need to be investigated. Another direction is to move to a probabilistic model and use Monte-Carlo like simulation and statistical model checking.

The Fractionated CPS model assumes honest, non-malicious agents and provides only weak guarantees about knowledge dissemination. The latter is to provide a general and realistic model. Furthermore agents are anonymous, identities are not revealed by by the system. A challenging problem is to identify primitives that, under suitable conditions, support stronger communication guarantees, and primitives that support building of trust both in the knowledge posted by other agents and in the capabilities and stated intents of other agents. It is important to have a balance between strong guarantees and the robustness enabled by weaker guarantees.

References

1. 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)
2. 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)
3. 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)
4. Interlink project (last accessed November 15, 2014)
5. Bristeau, P.J., Callou, F., Vissiére, D., Petit, N., et al.: The navigation and control technology inside the ar. drone micro uav. In: 18th IFAC world congress, vol. 18, pp. 1477–1484 (2011)
6. Krajník, T., Vonásek, V., Fišer, D., Faigl, J.: AR-Drone as a Platform for Robotic Research and Education. In: Obdržálek, D., Gottscheber, A. (eds.) EUROBOT 2011. CCIS, vol. 161, pp. 172–186. Springer, Heidelberg (2011)
7. Meng, L., Li, L., Veres, S.: Aerodynamic parameter estimation of an unmanned aerial vehicle based on extended kalman filter and its higher order approach. In: 2010 2nd International Conference on Advanced Computer Control (ICACC), vol. 5, pp. 526–531. IEEE (2010)

8. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction problem: Formalization and algorithms. IEEE Transactions on Formalization and algorithms. Knowledge and Data Engineering 10(5), 673–685 (1998)
9. Modi, P.J., Shen, W.M., Tambe, M., Yokoo, M.: ADOPT: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence 161(1), 149–180 (2005)
10. des Jardins, M.E., Durfee, E.H., Charles, L., Ortiz, J., Wolverton, M.J.: A survey of research in distributed, continual planning. AI Magazine 20(4), 13 (1999)
11. de Weerdt, M., Clement, B.: Introduction to planning in multiagent systems. Multiagent Grid Syst. 5(4), 345–355 (2009)
12. Bullo, F., Cortés, J., Martínez, S.: Distributed Control of Robotic Networks. Applied Mathematics Series. Princeton University Press (2009), Electronically available at `http://coordinationbook.info`
13. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI 2005, pp. 266–271. Morgan Kaufmann Publishers Inc, San Francisco (2005)
14. 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)
15. 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)
16. Brown, O., Eremenko, P.: The value proposition for fractionated space architectures. In: Proc. of AIAA, San Jose, CA (September 2006)
17. 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)
18. 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.) Soft-Ware Intensive Systems. LNCS, vol. 5380, pp. 1–44. Springer, Heidelberg (2008)
19. Ascens: Autonomic service-component ensembles (last accessed: November 15, 2014)
20. 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)
21. Arbab, F., Santini, F.: Preference and similarity-based behavioral discovery of services. In: Formal Methods (2012)
22. Kim, M., Stehr, M.O., Talcott, C.: A distributed logic for networked cyber-physical systems. Science of Computer Programming (2012)
23. 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)
24. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of real-time maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
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. Stehr, M.O., Talcott, C.: Planning and learning algorithms for routing in disruption-tolerant networks. In: MILCOM 2008. IEEE (2008)