Reconfigurable Autonomy: Architecture and Configuration Language

Louise A. Dennis

September 17, 2018

1 Introduction

During the course of the EPSRC funded Reconfigurable Autonomy Project a number of technical notes were generated outlining a potential architecture and configuration language. None of this work was published. This technical report compiles the contents of those notes for future reference.

2 Architecture

A **rational agent** is responsible for taking high level decisions about how to proceed. We distinguish a *rational agent* from other sorts of agent as one which takes these decisions based on explicit goals and beliefs and can therefore potentially explain its decisions naturally in a high-level way easily comprehensible to a human[1]. Goals will be derived from the mission goals, while beliefs will depend upon information from sensors and advisory systems and some world model. A decision, in this context, means selecting a plan for execution from some *plan library* and instantiating it where necessary (e.g., stating which rock a rover is to explore in some `explore_rock` plan). The plan library contains both plans provided by human programmers and, potentially, plans provided by other parts of the system (e.g., a planning adviser). The rational agent does not, itself, generate plans it simply makes decisions about which plan is to be executed when. We anticipate that most operation will take place *without* the intervention of the agent. The agent may make the decision to move to point B, but it will then request an appropriate path (if needed) from an advisory system, and trust the underlying control system to follow the path without intervention. It will only be if something changes, or fails, while the path is being followed that the agent will need to perform additional reasoning.

As hardware components degrade, or are changed, the rational agent, and adviser sub-systems (and possibly the control systems) can adapt to these changes. Such a change will be signalled to most sub-systems via changes in the knowledge base (or world model) that describes the global system. It should also be possible to change the adviser sub-systems available to the agent, this may include co-opting routines from other pieces of hardware and integrating them via wireless communications.

The shared information necessary for all these systems to collaborate is stored in the **knowledge base** (or world model). This contains several sort of information: the existing components of the system, their *capabilities*, *data streams* and composition (we call this the **configuration**); current ground facts about the world generated from sensors and diagnostics (we call these the **percepts**); models, route plans, maps, metrics, and other miscellaneous data in program specific formats that must be shared by components (we call these the **program_data**); rules for abstracting information and establishing a shared understanding of concepts (we tentatively refer to this as the **ontology**). At the moment we have no section of the knowledge base dedicated to the modelling of continuous information about the system and its environment in a consistent fashion (it may be that SENGLISH [13] and the ontology can do this). We have yet to make any decisions about the underlying implementation of the knowledge base (or world model), but it should be possible for the rational agent to view the information about the configuration, percepts and ontology as statements in first order logic.

Stream transformers *transform* data content and rate – e.g., continuous data can be made discrete and data can be sampled in various ways to prevent overloading of other components. The rational agent (and possibly other sub-systems) has the power to create and destroy the stream transformers between components.

Only the rational agent and the knowledge base (or world model) will be constant across all instantiations of this architecture. Each individual application will need a different set of control and adviser systems.

Open Questions

1. How easy is it for systems such as a learning systems, planning systems, SLAM algorithms, etc, to work with external world models? not to mention external vision classification systems, and prediction systems?

[1] It is also generally believed in the declarative programming communities that programming such agents is less error-prone than other programming methodologies and that the resulting programs are more amenable to verification and analysis.

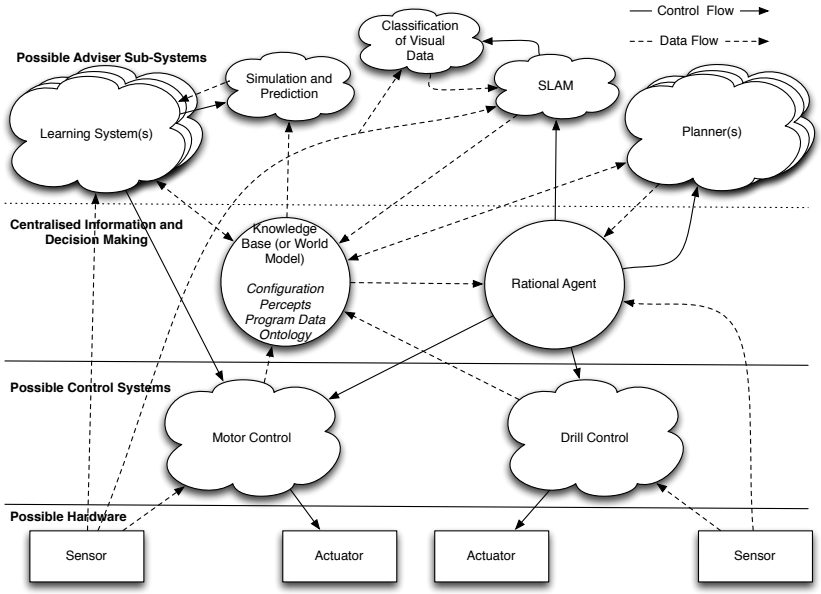

Figure 1: Overview of the Architecture. A centralised rational agent and knowledge base (or world model) connect together various control systems and software sub-systems. There may be multiple versions of some types of sub-system (e.g., learning systems for anomaly detection and other learning systems for classification and feature detection). Some sub-systems may communicate directly with each other, e.g., a learning system may use an external simulation engine to generate predictions of behaviour, and may have a direct connection to the motor control system in order to invoke particular actions as part of learning. Common software sub-systems such as SLAM localisation may also be treated as separate entities, invoked by the agent and communicating information to the knowledge base (or world model). These may make use of systems for classifying data (such as vision systems) which may, in turn, have been spawned by a learning system. Connections and components are dynamic and may change during the lifetime of the system.

2. What is the appropriate terminology for informally discussing parts of the system? (e.g., component versus module versus sub-system).

3. Do we need a section of the knowledge base dedicated to the storing of continuous information and equations about the system and the world?

Figure 1 shows an overview of the proposed architecture for reconfigurable systems. Such an architecture will consist of a rational agent; a knowledge base (or world model); a number of hardware components and their control systems; and a number (possibly zero) of supporting software sub-systems such as learning systems, planning systems, localisation modules and so forth. In the diagram possible flows of control are shown by solid arrows while possible flows of data are shown by dotted arrows.

2.1 Main Sub-Systems

2.1.1 The Rational Agent

We want the rational agent to be written in a BDI style language. In particular we want to it select plans for execution based upon its beliefs and mission goals. We believe that the information in the knowledge base (or world model) will be fundamental to the agent checking its beliefs. We want the agent to be able to:

- select plans based on the belief that certain capabilities, and data streams are available, or capabilities and data streams that satisfy certain constraints;
- react to changes in the knowledge base (or world model) – in particular, the loss, change or addition of capabilities;
- Understand that some capabilities depend upon other sub-system capabilities (e.g., learning will depend upon access to sensors) and be able to create links between relevant components and install the correct data transformers on the links – again this to be managed by plans.
- Reason about beliefs – e.g. by using Prolog to reason about the contents of the knowledge base (or world model).

- Have plans which involve configuring the system, and instructing sub-systems (e.g., learning, localisation, planning) to run, as well as plans focused directly on the achievement of mission goals.

The connection between the rational agent and the knowledge base (or world model) will be managed via two mechanisms. Firstly the agent can register an "interest" in particular ground facts, (e.g., **collision**) and will be notified on a "push" basis whenever those facts change. Secondly it can query the knowledge base (or world model) for additional information. This avoids over-loading the agent with information about every change that takes place in the knowledge base (or world model) while making sure that it can access any information it needs, and that it learns information of critical importance as soon as possible.

We assume that the knowledge base (or world model) does not contain an explicit description of the agent's own plans and abilities since other components do not need to reason about these.

2.1.2 Software Adviser Sub-systems

There are a wealth of systems that can be covered by adviser sub-systems. We are particularly interested (given the project deliverables) in systems that can *learn* and systems that can *plan*. I think, given the hardware, expertise and software we also have available that we should also look at systems that can *simulate*, *classify visual data* and *simultaneous localization and mapping (SLAM)* systems.

Classification of Visual Data These will, I think, be the easiest sub-systems to integrate since they will take in a stream of pixel data and output classifications for particular groups of pixels which can be stored in the knowledge base (or world model) potentially related to geographical positions in a world map. They do not need access to any other sub-systems in order to function. We will probably reconfigure them (e.g. if a camera is damaged) by replacing them by a new learned classification system rather than reconfiguring them internally. Our interest in these is primarily how they are produced by a learning sub-system, and how they are utilised as external systems from path planning and SLAM.

Planning Traditional goal-directed planning systems also appear straightforward to integrate into the proposed framework. Abstract descriptions of such planning problems are well-studied, as evidenced by the existence of the generic PDDL language [5], which appears to have a fairly good fit to the language of capabilities we are discussing. Similarly the output of such planning systems are plans that achieve some goal and these, again, appear to have a good fit with the internal language of the rational agent.

A second type of planning system is an ad hoc planner, particularly those concerned with route or path planning. We have a couple of examples (two involving UAVs and one involving a robot arm) in which utilise such path planning type algorithms.

Simulation/Prediction We already have some experience integrating simulation systems with rational agents. Up until this point we have used the simulation to either; answer yes/no questions such as "will there be a collision in the next 10 seconds"; or to produce information to be passed to the control system in the form of a thruster activation "plan"[2]. Both these capabilities have a fairly natural fit into our proposed architecture. However, to be reconfigurable, such a simulator would need to be able to change the underlying model it was working with. We haven't looked into how such a model could be described or transferred, though sEnglish style executable MatLab specifications are a step in this direction. We also haven't looked into using a simulation system which simply produces a stream of predictive sensor data rather than some yes/no answer or a plan, although that was discussed a number of times. We assume that physical models needed for simulation are either hard coded into the simulators or (preferably) available in the knowledge base (possibly described using SENGLISH) and can then be passed to simulators on demand.

SLAM SLAM systems require continuous access to sensor data and some model of the kinematics (and/or dynamics?) of the robot. To be reconfigurable, therefore, such a system needs to be able to adapt to different models and data streams. Output from a SLAM system is a continuously changing map. In the examples we have considered so far we've deemed it preferable if the SLAM system stores its map locally and instead outputs higher level information about location and objects. However the map could also be stored in the knowledge base (or world model) itself. If we are creating a learning system that can classify input from a vision system as (e.g, rocks) then a SLAM component would need to access the resulting classification system as an external sub-system. This might be challenging.

[2]Not to be confused with the plans used by the rational agent.

Learning Learning systems present many challenges. Not only are there several different types of learning system we want to consider (such as feature classification, and anomaly detection) and several different kinds of outcome. In some cases the learning system may spawn a new sub-system for some specific task.

Classification Systems seem to be the simplest to integrate. We can provide them with appropriate learning data via an input stream, and they will spawn a new classification sub-system which can then be used by, for instance, potential path-planning sub-systems or SLAM systems. The main challenge to answer here is where the learning data comes from, and how spawning works and is described.

Anomaly Detection Systems An anomaly detection system needs to compare the actual outcome of actions with simulated outcomes (ideally in the form of simulated versions of the actual sensor data). We are particularly interested in anomaly detection systems that generate error matrices which can then be used by a **capability modifier** to amend the system description in the knowledge base. An example of this is discussed later. Ideally, we would like a learning system to diagnose problems with the physical model being used (i.e, to realise that motion capabilities have changed because the vehicle's front right tyre is flat), but that is a far more challenging task.

Open Questions

1. Do we want to be able to generate sub-systems "on the fly" (e.g., new classifier systems). If so, how is this done and how is it described?

2. How do we understand and describe the limitations of, say, an anomaly detection system? (e.g., that it only detects anomalies in wheeled vehicles).

3. If we have an anomaly detection system that responds to a flat tyre by amending a capability, will the fact that we never deduce that the *actual* problem is a flat tyre cause knock-on problems throughout the system? Could the fact that we have compensated for the error without deducing the actual cause introduce inconsistencies into our world model? Even if not, is it likely to cause confusion when communicating with human operators? If so, how do we handle this?

2.1.3 Control Systems

Control systems consist of low level code based on control theory. They occupy an uneasy place between the hardware and the higher software layers. They will offer a range of capabilities from the very simple (e.g. **open_valve1**, **turn_wheel_180**) to more complex (e.g. **maintain_position**, **follow_path**, **increase_flow**). To do this the control systems will need access to both sensors and actuators. It doesn't seem implausible that some pieces of hardware (possibly just sensors) will be accessed by several control systems.

Control systems can share sensors (it will just be a matter of subscribing to the same data stream). We have not yet needed to represent actual actuators in the system, any actuator being strongly associated with some control software. In principle actuators can be shared, but this is to be discouraged.

Open Questions

1. How reconfigurable can control software be? Should they allow reconfiguration of sensors but not actuators? Does that work via data connections and the knowledge base (or world model) (like the rest of the architecture) or is there a different mechanism for reconfiguring control. Do we consider a control system that is changed because of hardware change or failure as a *new* control system or as a reconfigured existing control system?

2.1.4 The Knowledge Base (or World Model)

We discuss the knowledge base (or world model) in more detail later. It is a central database which stores the information that needs to be shared by the various sub-systems. The knowledge base should not be something that changes rapidly and so sensor data should not be stored directly. There will be at least two different views on the underlying knowledge base (or world model). SENGLISH will be used by programmers to describe shared understandings between parts of the system, including dynamics and kinematics that can be used by simulators etc., meanwhile at the highest level the rational agent will interpret the information in the knowledge base (or world model) as statements in first order logic. A language for configurations is described in section 3, which we hope will mediate between the first order logic demanded by the agent, the temporal logic for streams used in stream transformers and PDDL used by planners, this has yet to be fully set out. We anticipate that SENGLISH will also be used to generate statements in this intermediate language for configurations. We may also want a view suitable for modelling dynamic and kinematic information for use in simulators and other components that make use of continuous modelling data. Possibly SENGLISH can supply this.

Planning sub-systems use the knowledge base (or world model) to access the capabilities of other parts of the system. Learning, simulation and SLAM sub-systems access models about the real world and the anticipated effect of actuators upon it. The rational agent accesses high level abstract descriptions of the world and the current system configuration.

The knowledge base (or world model) is both modular (with sections dedicated to abstract *percepts*, the configuration, program specific data, and logical rules but possibly also with modularity within those sections, e.g., so "drilling" and "moving", say, can be accessed separately) and hierarchical in order to describe things at different levels of abstraction.

We want the knowledge base (or world model) to explain both the composition of the hardware system it is a part of, and the beliefs/assumptions about the real world. So, for instance, we want to store both the fact that **sensor1** outputs a stream of real numbers and that we believe these real numbers reflect the exterior temperature.

Open Questions

1. How do we handle capabilities which synthesise new capabilities and components?
2. Should capabilities be able to specify time outs? after which other parts of the system may assume the capability has failed and attempt something else? Should the implicit *eventually* be an explicit *within time t*.

2.1.5 Stream Transformers

The system will be able to connect inputs and output of various sub-systems together, on the fly if necessary. These connections will not be "dumb" connections but will be able to transform data (e.g. a stream of real numbers may be transformed into a stream of predicates "temperature(high)" or "temperature(low)" for instance. Similarly the streams may tag data, and sample data)[3].

At the moment we propose using Alexei Lisitsa's temporal logic based stream transformation language [8] for these connections. We hope, among other things, that this language will make it possible to synthesize transformers on the fly.

Brief Overview of Data transformation Language This language treats all data streams as a stream of tuples $\langle \mathcal{T}, \tau \rangle$.

- $\mathcal{T}$ is a set of *tagged data* of the form $\{(w_1, d_1), \ldots, (w_n, d_n)\}$ were w_i is a tag and d_i is an item of data.
- τ is a *time stamp*.

Data streams from sensors are expected to be of the form $\langle \{(s, d)\}, \tau \rangle$ were s is the "name" of the sensor and d is the sensor reading at time τ.

Transformers transform one kind of data stream into another. For instance a transformer could take a stream of the form $\langle \{(accel, a)\}, \tau \rangle$ of acceleration values from some sensor and perform a transformation that considers all the data in the stream in a 10 second window and then outputs a stream of data of the form $\langle \{(max\_accel, a')\}, \tau \rangle$ which is the maximum acceleration over the 10 second interval before τ. Similarly a stream can average or minimize data.

Transformations on a stream can also be conditional and perform simple feature detection. For instance a sub-system may only be interested in temperatures over a certain threshold, a data transformer can be constructed in this language which transforms a stream of temperature data into a stream of temperature warnings $\langle \{(temperature, p)\}, \tau \rangle$ were p could just be a predicate value $warning(temperature)$ which indicates the temperature has exceeded the threshold at time τ.

A sample implementation of this stream transformation has been implemented on top of Esper which is a highly efficient stream processing system. This means that the transformer itself should not get overwhelmed by the flow of data from a sensor. Because the language outputs a stream of data it is obviously possible to string these data transformers together to provide, for instance, a hierarchy of abstractions on a low-level data stream – it is not clear at this point whether we want to use this facility in our system.

The language for writing transformations is based on linear temporal logic. A transformer has a tag (e.g., `High Average Speed`), a conditional statement (e.g., $average_{[0,-5]}(speed) > 30$ – meaning the average over input streams tagged `speed` in the last 5 seconds is greater than thirty), and tagged terms for new data items (e.g. $average_{[0,-5]}(speed)$ – the average over the last 5 seconds of the input tagged speed). So an example transformer is:

$$(\texttt{High Average Speed}, average_{[0,-5]}(speed) > 30, \{\langle \texttt{av\_speed}, average_{[0,-5]}(speed) \rangle\})$$

This transformer will act on a stream of data tagged $\langle \{(\texttt{speed}, d)\}, \tau \rangle$. Every time the average of `speed` over a five second interval is greater than 30 the transformer places $\langle \{(\texttt{High Average Speed.av\_speed}, sp)\}, \tau \rangle$ were sp is the average speed over the last five seconds. This can be interpreted as meaning that `High Average Speed` holds as witnessed by `av_speed`, sp. It does not put anything on the stream if the average is under thirty. As well as specific aggregation operators (like average) temporal logic statements can be used such as $\Box^{10} speed > 30$ which will return true if all the data for speed arriving in the last 10 seconds has been over 30.

[3]This means that practically speaking, in terms of say a ROS implementation, a connection will b e a transformation node sitting between two data streams. Only at our architectural level do we consider it as a single intelligent data connection.

Open Questions.

1. For streams of commands we are probably going to want a transformer to be able to transform a single data item (e.g., `change_fuel_line`) into a sequence of data (e.g, `close_valve1;open_valve2`). This is currently listed as an extension to the formalism and implemented system.

2. What sort of sampling protocols are needed? Can they be described in our transformation language?

2.1.6 World Knowledge

We will need **modules** so that we can isolate and quickly access all the data about, say, drilling, or moving or mapping. **modules** should be tightly coupled with individual components and should encapsulate the data associated with that component. The draft interim configuration language uses an object-oriented style syntax to associate details of the configuration with specific components.

Information will also need to be **hierarchical** so that we can access facts at different levels of abstraction so that, for instance, we can reason just about paths on the world map, or in more detail about the interactions of an agent's hardware with the external terrain.

Where information is stored in some program specific format (e.g., MATLAB mfiles) then it will be kept in the **program_data** part of the knowledge base (or world model) and a tagging scheme can be used to store relevant meta-data (e.g., the language).

2.1.7 Logical Reasoning with the Knowledge Base (or World Model)

There is a lot of potential for duplication in the knowledge base (or world model) - particularly in where we store information about the real world. In particular the **stream ontology** that is used to generate stream transformers is probably (but not necessarily) a restricted sub-set of a more general **ontology**. However if it is completely separate then there is a risk that information about what it means to be "too hot", for instance, could get stored in both places.

We have a rather large candidate set of descriptive mechanisms – Prolog like predicates, precondition/postcondition style triples, temporal logic for stream transformers, continuous modelling formalisms, and low level imperative language code. We need to minimize this, where possible. Since many aspects of the system need to be able to query the knowledge base (or world model) we need to think of it as more than just a set of data structures for describing the system and the world, but as a language over which we can perform reasoning, in which case we need to provide a semantics for the language, a reasoning algorithm over statements in that language *and ideally* some idea of the complexity, soundness and completeness of such reasoning. The beginnings of this is present in section 3 though it only focuses on the highest level where, it is hoped, reasoning can be reduced to first order logic.

As well as reasoning over the language we are going to have to deal with issues of consistency, *especially* where hardware is degrading or becoming broken and the system is compensating for the degradation without diagnosing the root cause. We need to prevent inconsistencies in the description of capabilities and the world model from destroying the ability to reason reliably over the ontology. We may need some way to partition the knowledge base (or world model) into reliable and unreliable information, or we will need truth maintenance and belief revision processes of some kind which can be used to manage the information within the ontology.

Open Questions

1. What do queries over facts in the knowledge base (or world model) look like?

2. Is reasoning sound? (it had better be, but we need to prove that).

3. Is reasoning complete? (I'm guessing unlikely if low-level code appears in places).

4. What is the complexity of reasoning? Can we find restricted fragments with good complexity results?

5. How will consistency checking work?

2.2 Thoughts on using a Layered Architecture

[4] proposes an architecture for self-reconfiguring systems which is based upon the traditional three-layer architectures common in robotics. This architecture can be viewed in the abstract as a sequence of layers each consisting of three general components one of which *senses* information about the layer below, one of which *analyses* that information and then one of which *administrates* the layer below.

So the base layer is the traditional robotics control systems layer with sensors, feedback loops and actuators, then you can have reactive planning systems which sense information about goals, and work with more abstract information, and then failure analysis layers on top of that whose admin components reconfigure the layers below.

It is superficially tempting to split our architecture up into these layers within the agent and the knowledge base and to assign components to each layer. So for instance the lowest layer sits on top of the control systems layer in the traditional layered architecture and consists of reactive plans, which monitor external events (e.g., changes in perception) and send commands to components which control actuators. One of the main advantages of the three layer architecture is that it allows "gearing". At each layer the processes are more computationally expensive and can be managed to make sure that "quick" responses occur in a timely fashion. So on top of our reactive planning layer we have a layer which consists of planners and simulators. Many of these components use information about the configuration of the system (which are not used by the reactive plans) and the outputs from these components become new reactive plans at the layer below. Plans in the middle layer are activated by monitoring and analysing the agents' goals. At the top layer we have tools which react to problems with the configuration (either the actual configuration or the information about them on the layer below). This layer monitors the behaviour of the reactive plans (when they are selected, whether they succeed etc.), and has components which may monitor the system performance and change the system configuration or its description. A version of this layering is shown in figure 2.

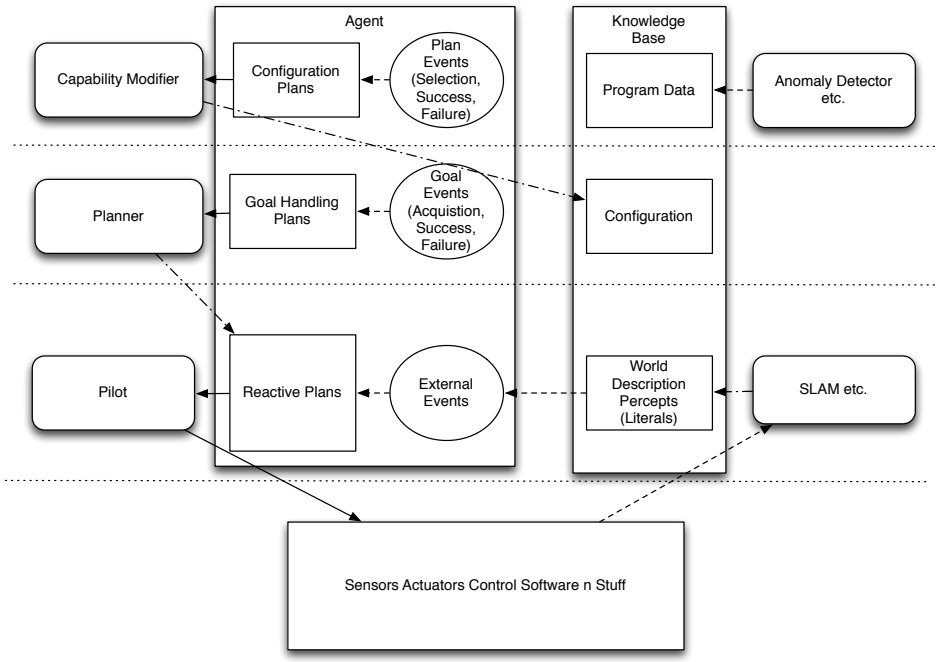

Figure 2: A Layered View of the Architecture

The problem with this viewpoint is that it starts to fall apart when more detail is included. For instance the anomaly detection system (see section 4.2) actually monitors the real sensor output from either two or three layers below, not just goal behaviour. Similarly the capability modifier and anomaly detection systems can both be linked and unlinked by configuration changes, even though they are at the configuration layer, not those below it. Lastly since we allow compositional components, these can, internally, consist of sub-components that span several layers.

Nevertheless I think there is some value to layering within the agent on the level of plans which react to external events and result in actions or subgoals, plans which react to goal behaviour and result in new plans (or new instantiations for plans), and plans which react to plan and system performance and result in new configurations. We do need to think harder about how components and connections might fit into such layering, and what the restrictions on each layer may be.

3 Configuration Language

This section presents a series of examples in order to design a language for describing the configuration of hybrid autonomous systems. The purpose of this language is to allow autonomous reasoning about configurations. This means that the language does not need to describe every detail of an autonomous system's configuration, nor does it necessarily have to explicitly mention aspects which do not change.

We assume that such systems are constructed according to the architecture out-lined in section 2. We assume, therefore, that all the systems described contain an agent and a knowledge base[4] within one software/hardware entity. The knowledge base receives information about the environment from other components and passes it on to the agent where relevant. At the moment we are assuming that components are connected by streams of data. At some point we may wish to view streams as a form of component. Alternatively we may want a wider variety of connectors (c.f. [9]). The knowledge base also has a query interface, which allows components to query it for specific information.

The knowledge base is modular with several distinct sections. These include a section for *perception* (percepts with an input steam literals) which consists of ground literals describing the external world, and a *stream ontology* which is expressed in a variety of temporal logic similar to that introduced in [8]. The *configuration* is expressed in the language we develop here but we assume configuration descriptions can also be reduced to literals (so configuration also has an input stream literals). Lastly we assume a module which consists of arbitrary program-specific data shared between specific sub-components (e.g., MatLab mfiles, or map data). We don't, at present, concern ourselves with program-specific data but we do assume that components can query the module to extract the data they are interested in.

At the moment we are assuming that all components "mean" the same thing by predicates such as $speed(S)$ and so forth. At some point we will need to extend our language to include shared ontologies and/or mediators between ontologies (e.g., as discussed in [11]). These may need to be expressed in a richer language than that used by the stream ontology. We do not consider this explicitly here but merely observe there may be a separate *shared ontology* which will include hierarchical data abstractions, as well as the existing stream ontology.

Ultimately we anticipate that sENGLISH will be used to construct and manage the knowledge base. In particular it will interface between the shared ontologies, the stream ontology, program data, configuration and percepts in a coherent fashion. As a result, the syntax used here should not be considered definitive. However it seemed desirable to have a semi-formal representation and, in particular, a representation that would interface well with the logical reasoning with which this note is primarily concerned.

As well as being able to query the knowledge base for specific information, we also assume that the agent can register an interest in specific predicates and will be informed of changes in those predicates as events. At the moment we assume these predicates must be in percepts or configuration. The agent does not get informed of changes that can only be discovered by applying deduction using shared ontologies to the percepts or other categories of data.

Key parts of component descriptions are the *capabilities* and *streams*. Capabilities are called by external systems via a dedicated input stream, called command. A capability executes one command and may return a value. It's execution is described in the configuration language via logical pre- and post-conditions. Since we anticipate that we may wish to apply forward planning to the computational resources, and actuators in our system we want the stored descriptions of these capabilities to be close to the language of PDDL [7] in the anticipation that we may want to connect to PDDL based planning systems. Some streams are used by capabilities or query interfaces in the knowledge base and so values are placed on them in response to stimuli. But components may also have streams which are always active, (or always active after being switched on). We also use logical property descriptions to describe what it means when a value is placed on one of these streams.

We use a series of examples to introduce the key aspects of this language.

4 Examples

4.1 Example 1: A Lego Robot with a Single Sensor

In our first example we consider a Lego robot which can move around a space and detect obstacles with a single forward facing sensor, either a touch sensor which registers when a button is depressed or an ultrasonic sensor which returns integers representing the distance to the nearest object. In the anticipated scenario we imagine switching one sensor for another and allowing the robot to reconfigure its internal systems appropriately.

4.1.1 Structural View

Figure 3 shows a structural view of the main components in the Lego rover. We have five components, the agent and knowledge base are always present while the pilot, sensor and sensor wrapper are specific to this example. The knowledge base receives the information from the sensor (either the touch sensor or the ultrasonic sensor). Because the actual sensor interfaces allow only for function calls which return a single data point we have our own "wrapper" around the sensor which continuously calls the function and transforms the results into a stream of data. The agent receives this data from the knowledge base and, based on its internal deliberations, sends commands to the pilot which interfaces to the robot's motors. The stream of data from the sensor is transformed using a definition for obstacle that is drawn from the stream ontology.

We start with a high-level description of the rover system shown in listing 1. We use the name:type notation to name our components. This description is for a Lego rover with an ultrasonic sensor. This describes three components: the sensor, u, the wrapper uw and the pilot, p. The wrapper component is actually a compound component which contains u.

[4]Or World Model. I'm not particularly fussed what this is called, but we should agree terminology soon.

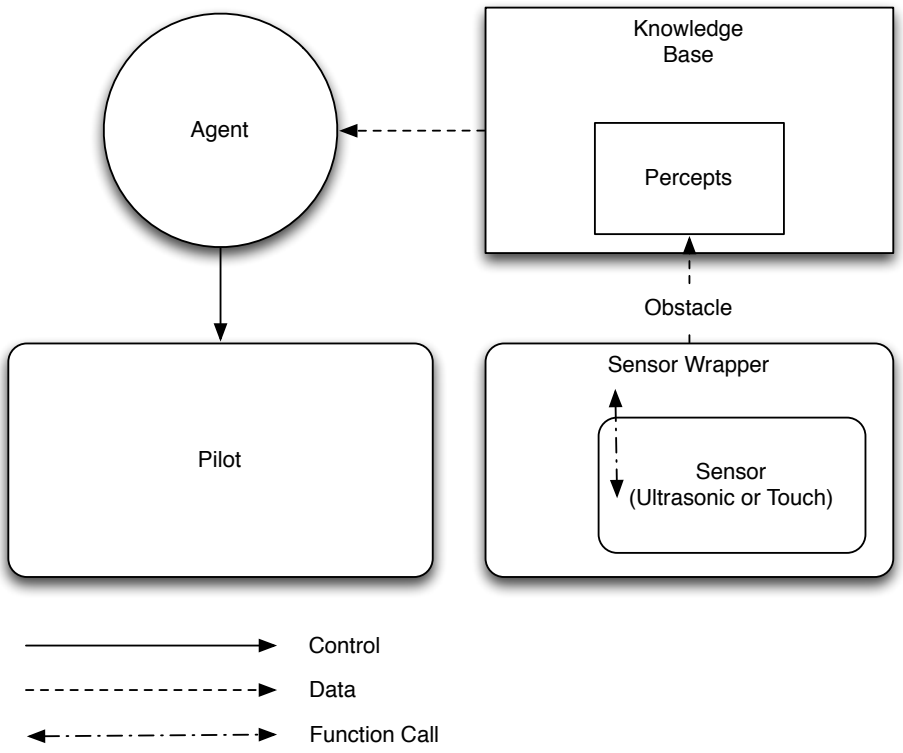

Figure 3: A Lego Rover

We do not need to completely describe the agent and knowledge base since they are always present and most of their capabilities and streams can be assumed to be static. However both the stream ontology and the interests the agent has registered do need to be described since they are application specific.

We use the stream connections to show how we connect the in and out streams of the components. We assume that all streams are named. We shall see in listing 6 that the ultrasonic sensor wrapper has an output stream called distance. This stream is connected to an input stream in percepts called literals . The connection between the sensor wrapper and the percepts is transformed by a *stream transformer* which is derived from the stream ontology formula by recognising from the description of the distance stream that it satisfies the preconditions for $obstacle$[5] (See section 4.1.4). Similarly we assume that the agent component has an array of output streams upon which it puts external action calls. The first of these is connected to the command input stream of the pilot.

When we describe the pilot in listing 2 we do not show this command input stream. We assume that all components have a distinguished input stream, called command, which is used to invoke specific capabilities.

Lastly there is a component which states which parts of the knowledge base are to be pushed to the the agent as events. In this case it is changes in the $obstacle$ predicate that is located in percepts

4.1.2 Capabilities

Listing 2 is a specification of the capabilities of the differential pilot. We have defined two capabilities, forward and rotate. rotate takes an argument D which defines the angle of rotation. We adopt Prolog/BDI conventions so that capitalised letters indicate universally quantified variables that appear in both pre- and post-conditions. Any instantiation (provided by either forward or backward reasoning) having to be consistent in both parts for any one call of the capability. In the actual function call we indicate parameters that must be provided as input with a + and those which are returned as output with a $-$.

If we intend our capability descriptions to be compatible with PDDL [7], then the preconditions can be written in full first order logic, while the post-conditions are restricted to the fragment that does not contain disjunctions or existential quantifiers. If we want to incorporate the idea that the execution of capabilities can take time, and specifically to capture notions such as eventually then this isn't expressive enough for us. However one of the earliest extensions to PDDL, PDDL 2.1 [6] allows *durative* actions and the use of the keywords `at start`, `at end` and `over all` in both preconditions and post-conditions. All the IPC planning competitions since 2002 have used PDDL 2.1 or an extension of it, so it seems like

[5]We'll need to work on the transformation a bit, specifically converting between pairs of data+tags into predicates.

```
rover {
  Components {
     p: differential_pilot
     u: ultrasonic_sensor
     uw: ultra_wrapper(u)
  }

  Stream Ontology {
     obstacle ↔ obstacle_at(X, Y) ∧ forward_distance_to(X, Y) < 30
  }

  Stream_Connections {
     uw.distance −− | obstacle if distance < 30 | −−> percepts.literals
     agent.act[1] −− −−> p.command
  }

  Agent_Interest {
    percepts.obstacle
  }
}
```

Listing 1: A Lego Rover with an Ultrasonic Sensor

a good compromise between core PDDL and allowing some description of time and continuous change. We therefore have capabilities that can be expressed in a sub-set of the action description syntax for PDDL 2.1. Capabilities can have a duration (left unstated in these examples) and the distinguished symbol $\#t$ is used to indicate time since the capability was invoked. I use the subscripts $_s$, $_e$ and $_d$ to indicate "at start", "at end" and "over all" (during) respectively.

```
Differential_pilot {

  Capabilities {
     forward:
       duration{}
       pre{speed(S)_d ∧ at(vehicle, X, Y)_s ∧ orientation(A)_d}
       post{at(vehicle, f_x(A, X) * S * #t, f_y(A, Y) * S * #t)_d}

     rotate(+D:double):
        duration{}
        pre{orientation(vehicle, A)_s}
        post{orientation(vehicle, A + D)_e}
  }

}
```

Listing 2: The Pilot of a Lego Rover

The pilot has two capabilities, forward and rotate. NB. I'm using $f_x(A, X)$ as shorthand for $X/sin(A)$ and $f_y(A, Y)$ as shorthand for $Y/cos(A)$.

rotate also takes an argument (a double, D, for the angle of rotation). +D states that D must be supplied as input to the capability.

The capabilities in the differential pilot have effects in the real world that are detected through perception and act over time, but the capabilities in the two sensors operate as functions and return values which are placed on named output streams. The descriptions of these are shown in listings 3 and 4. Here we use the notation −D to indicate a value that is returned by the capability and we state explicitly which output stream it is placed on. I think we have some redundancy in the notation here, but I'm leaving it as it is at the moment. Since these capabilities are not durative in the sense that the differential pilots were, we don't indicate duration or the time at which the pre and post conditions occur. The postcondition for bump states that if the output is true then there is an obstacle at the vehicle's position and if it is false then the position is clear. The postcondition for the ultrasonic sensor indicates the position of the obstacle in relation to the vehicle based on the distance returned and the

orientation of the sensor.

```
touch_sensor() {
  out–bumped: bool

  Capabilities {
    bump(–B:bool): bumped<B>
      pre{at(vehicle, X, Y)}
      post{B → at(obstacle, X, Y) ∧ ¬B → clear(X, Y)}
  }
}
```

Listing 3: A Lego Touch Sensor

```
ultra_sensor() {
  out–distance : int

  Capabilities {
    getDistance(–D:int): distance<D>
      pre{at(vehicle, X, Y) ∧ orientation(this, A)}
      post{at(obstacle, X + f_x(A, X) * D, Y + f_y(A, Y) * D)}
  }
}
```

Listing 4: A Lego Ultrasonic Sensor

4.1.3 Composing Components and Stream Properties

The output from the two sensors are converted into streams by the wrappers. We are unconcerned about the internal implementation and how it achieves this, but at a logical level we want to know that the behaviour of these streams is related to the behaviour of the internal component. The wrappers, therefore, incorporate the sensors' pre- and post-conditions into their own descriptions. We pull the pre- and post-conditions from the sub-components so that the wrapper description will change simply if the description of the underlying component changes (even though we are not explicitly considering that scenario). Listings 5 and 6 show the wrapper components

```
touch_wrapper(X:touch_sensor): {
  out–bump : bool

  Stream_Properties {
      rates:
          bump: [1s, 5s]
      output properties:
          bump⟨B⟩ : pre(X.bump(B)) → post(X.bump(B))
  }

  Components {
    X: touch_sensor
  }
}
```

Listing 5: A Lego Touch Wrapper

These components are also described primarily in terms of the properties of their streams, *not* their capabilities. Where we expect capabilities to be used primarily by planning components, we expect stream properties to be used during reconfiguration to both identify streams that can be substituted based on the requirements in the stream ontology and to control sampling rates between inputs and outputs.

```
ultra_wrapper(U:ultra_sensor): {
  out-distance : int

  Stream_Properties {
    rates:
      distance: [1s, 5s]
    output properties:
      distance⟨D⟩: pre(U.getDistance(D))) → post(U.getDistance(D))}
  }

  Components {
    U : ultra_sensor
  }
}
```

Listing 6: A Lego Ultrasonic Wrapper

So the stream properties fall into two categories, *data rates* and *output properties*. Output properties provide logical specifications for the data on the streams. For instance in listings 5 and 6 these show that when a value is placed on the output stream, then if the preconditions for the sub-component capability hold, then the post-conditions also hold for that value. As usual, I'm using capital letters to implicitly indicate universally quantified variables. At the moment I'm assuming that the property holds *at the moment when* the data is put on the output stream.

4.1.4 Stream Transformers

Lastly we consider the Stream transformers which are generated from the ontology. In the description using the Ultrasonic sensor the formula in the ontology is obstacle if distance < 30. This translates, I think, into the stream query:

$$\{obstacle : \top, value : distance < 30\}$$

This places the output $\langle obstacle, (value, \top), \tau\rangle$ onto the stream every time $\langle (distance, d), \tau\rangle$ appears on the input stream (where d is an integer less than 30 and τ is a time stamp). In our system we assume that we have $(distance, d)$ if the value d is placed on a stream named $distance$. And places $\langle obstacle, (value, \bot), \tau\rangle$ on the stream when d is greater than 30. $obstacle, (value, \top)$ then needs to be interpreted as the literal $obstacle$ and $obstacle, (value, \bot)$ as the literal $\neg obstacle$.

Our intention is that the agent (or some dedicated stream reasoning subsystem) can take the stream ontology definition of obstacle:

$$obstacle \leftrightarrow obstacle\_at(X, Y) \wedge forward\_distance\_to(X, Y) < 30$$

And the output property derived from the distance sensor pre- and post-conditions:

$$at(vehicle, X, Y) \wedge orientation(this, A)$$

$$at(obstacle, X + f_x(A, X) * D, Y + f_y(A, Y) * D)$$

and deduce that $obstacle$ is satisfied if a D less than 30 is placed on the distance sensor wrapper's distance output stream. At the moment the details of how this are achieved are left unclear but they will depend upon reasoning using the shared ontology, allowing the distance in front of the robot to be related to it's orientation etc.

Similarly, if the distance sensor is switched for the bump sensor then the agent should be able to reason that now $obstacle$ is satisfied if $\top$ is placed on the bump output stream so the stream transformer will be modified to obstacle if bump.

I've not considered stream transformers that are used to sample streams to control "gearing" between components. Providing the data rates are given in component descriptions it should be possible to construct stream transformers with the appropriate gearing. However, particularly if stream transformers are not described explicitly, we would need to decide whether and, if so how, such gearing is recorded in the knowledge base.

4.2 Example 2: Anomaly Detection System

The anomaly detection system is based on the system proposed in [12]. The system has two states. The first is very similar to the Lego Rover, the robot is navigating using a SLAM system that is connected to a saliency tracking based vision system. The Rover is also using a route planning system which draws information on the vehicle's movement capabilities from the knowledge base. We assume that the planner can query the knowledge base for the vehicle's capabilities.

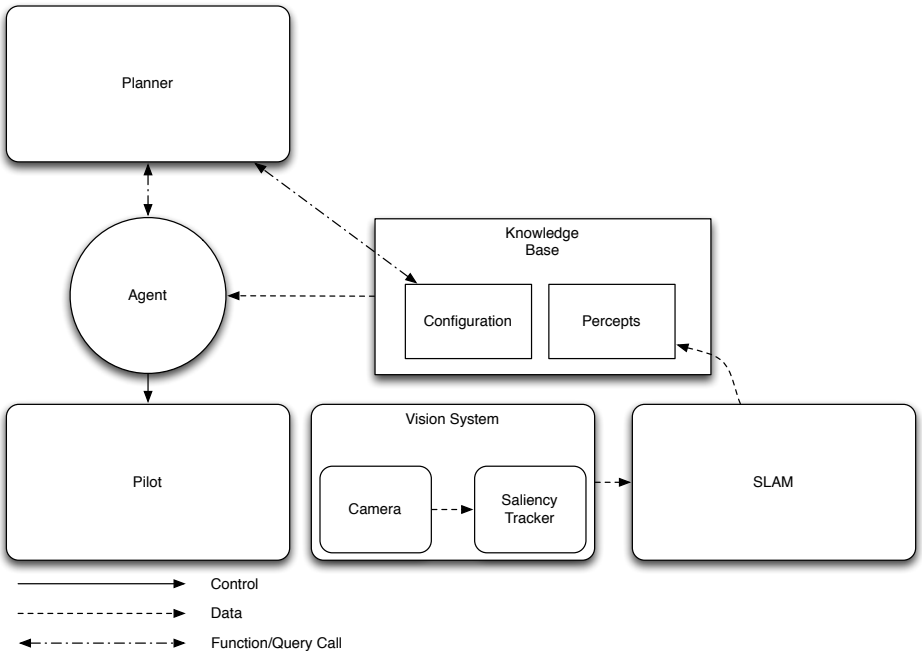

Figure 4: A SLAM based Rover with Planning

An anomaly arises meaning that the positioning information coming from the SLAM system isn't consistent with the projected position information (i.e., the agent isn't achieving its goal positions and is doing a lot of re-planning in order to reach its target locations). The agent reconfigures the system to bring an anomaly detection system, a simulator and a capability modifier online. The commands being sent to the pilot are also sent to the anomaly detection system and the simulator. The simulator outputs a stream of data that would be produced if the pilot were acting as advertised. The anomaly detection system compares the streams of data and eventually outputs an error matrix for some (set of) capabilities in the pilot. The capability modifier then updates the internal representation of these capabilities so that planning will be correct in future.

4.2.1 Structural View

Block diagrams for these two configurations are shown in figures 4 and 5.

The composition of the original system is shown in listing 7. While the agent, in this system, has the simulator etc., available to it they are not connected into any part of the system.

The agent sends commands to the pilot and the planner. It is receiving plans from the planner (we assume in a known, always present input stream like percepts. literals). Meanwhile the knowledge base is getting position information from the SLAM system which, in turn, is getting tracking information from the vision system. It should be noted that I'm assuming that plans are stored in the agent and not in the knowledge base. The planner can request configuration data from the knowledge base via the knowledge base's query interface (represented by $<--\ -->$ in the configuration file).

Listing 8 shows the description of the system after it is reconfigured to detect anomalies. The commands that the agent sends to the pilot are now also going to the simulator and the anomaly detector, meanwhile the agent is also sending commands of its own to the simulator and capability modifier. The SLAM system is sending its output to the anomaly detector as well as to the percepts. Both the anomaly detector and the capability editor also send information to the knowledge base. The capability modifier can change the description of the agent configuration. The error matrix (or possibly matrices if several capabilities are effected - which they would be if, for instance, a wheel was damaged) is stored in the program_data part of the knowledge base which is used for non-logical information such as MatLab files.

In [12], Shaukat and Gao suggest that the Vision System (as well as the SLAM system) need access to the agent/knowledge base. I wasn't quite sure if that was necessary for this example or just a more generic possibility. At present I'm assuming that the vision system doesn't need any interaction. It simply passes its information on to SLAM. Similarly I'm assuming that the SLAM system is maintaining its own internal map and not using the knowledge base to store this information, only passing on information at a much more abstract level to the knowledge base. But we should consider a situation where the SLAM system map is stored in program_data and predicates about positioning are inferred from that using a shared ontology, possibly based on SENGLISH.

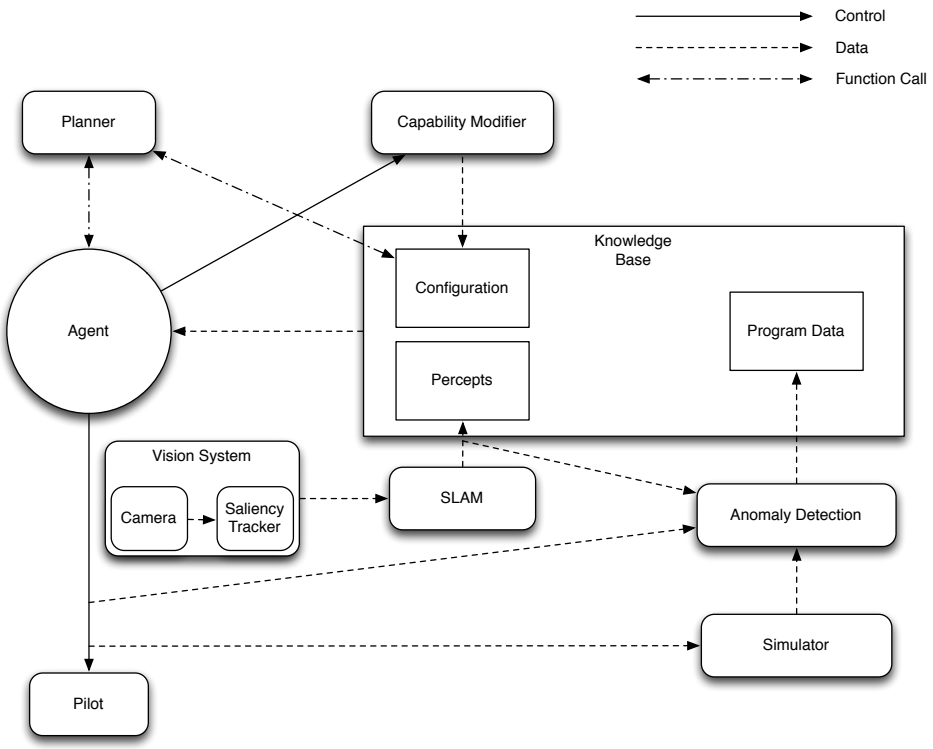

Figure 5: The Anomaly Detection System

```
rover {
   Components {
     p: pilot
     sim: simulator
     c: camera
     v: vision(c, s)
     s: saliency_tracker
     ad: anomaly_detector
     cm: capability_modifier
     sl: slam
     pl: planner
   }

   Stream_Connections {
      agent.act[1] --> p.command
      agent.act[2] --> pl.command
      pl.config <--> configuration.query
      pl.plans --> agent.plans
      v.see --> sl.annotated_images
      sl.pos --> percepts.literals
   }
}
```

Listing 7: Structural Composition

```
rover {
   Components {
     p: pilot
     sim: simulator
     c: camera
     v: vision(c, s)
     s: saliency_tracker
     ad: anomaly_detector
     cm: capability_modifier
     sl: slam
     pl: planner
   }

   Stream_Connections {
      agent.act[1] --—> p.command
      agent.act[1] --—> sim.simulate
      agent.act[1] --—> ad.error_commands
      agent.act[2] --—> pl.command
      agent.act[3] --—> sim.command
      agent.act[4] --—> cm.modify
      pl.plans --—> agent.plans
      pl.config <—-—> configuration.query
      v.see --—> sl.annotated_images
      sl.pos --—> percepts.literals
      sl.pos --—> ad.error_real_input
      sim.output --—> ad.error_sim_input
      sim.config <—-—> program_data.query
      ad.error --—> program_data.error
      cm.cap_edit --—> configuration.capabilities
   }
}
```

Listing 8: Structural Composition

4.2.2 Vision System

The vision system is another composite component. [12] describes half a dozen components that are required to build a saliency based tracking system, for the time being we will simplify this into a Camera and a saliency-based tracker.

The description of the camera is straightforward and show in 9. It outputs a stream of bitmaps at a particular data rate, let us say 1 per second. We don't need any semantic information (i.e., stream output properties or capabilities to describe this information) beyond the type of the output stream.

```
camera() {
  out-images: bitmap

  Stream_Properties {
     rates:
       images: [1s]
  }
}
```

Listing 9: Camera

4.2.3 Tracking Systems

At some point we are going to need to grapple with interfaces and this seems a reasonable point to explore a simple version. Interfaces will allow systems to switch flexibility between different implementations of underlying sub-systems. They are related, though not the same, as the issue of an agent looking for a component with some particular capability, or a capability that obeys some constraints. Interfaces are very much a part of standard practice in component-based software engineering and we need to accommodate them.

Let's assume we have a tracker interface which is for something that takes in a stream of images and outputs a stream of images paired with annotations indicating a "box" in the image and an number. Shaukat and Gao's system identifies boxes by the coordinates of the top left hand corner of the box plus the width and height of the box – for simplicity we will assume this is represented as a tuple of four integers[6]. Boxes annotated with the same number in successive images are assumed to be the same object. This interface is shown in listing 10 and expresses the idea that boxes annotated with the same integer in different outputs mark the same object.

```
interface tracker() {
  in-images: bitmap
  out-boxes: (set(int, (int, int, int, int)), bitmap)

  Stream_Properties {
     output properties:
       boxes⟨(S, IM)⟩, boxes⟨(S, IM′)⟩ : {(I, B) ∈ S ∧ (I, B′) ∈ S′
                                          → object_at(B, IM) = object_at(B′, IM′)}
  }
}
```

Listing 10: Tracker Interface

The actual saliency tracker that implements this interface is described in listing 11. NB. the description is basically an interface itself so I'm using the keyword extends rather than implements. The saliency tracker imposes a constraint on the input stream (that data must come in at a rate of no more than one image per second) and also states that the output stream will work at the same rate as the input steam. It also modifies the description of the output stream from the tracker interface to indicate that the property only holds 75% of the time.

NB. The notation $[1s, \infty]$ indicates that the component produces data *no faster* than once per second, but that it may never produce data at all.

The vision component (Listing 12) then composes a camera and a tracker and constructs the properties of its own output stream from those of the tracker's output stream.

[6]Shaukat and Gao's system also identifies the centroid of the object of interest inside each box – I'm ignoring that for now.

```
saliency_tracker extends tracker {
  Stream_Properties {
     rates:
        images: [1s, ∞]

     output properties:
        tracker.boxes(prob:75%)
  }
}
```

Listing 11: Saliency Tracker

```
vision(C:camera, T:tracker): {
  out-boxes : (set(int, (int, int, int, int)), bitmap)

  Stream_Properties {
     rates:
        boxes : rate(T.boxes)

     properties:
        boxes : T.boxes
  }

  Stream_Connections {
     C.images --—> T.images
  }
}
```

Listing 12: Vision

4.2.4 SLAM

The SLAM system maintains an internal map of the area but I'm not sure it is necessary for the knowledge base to be aware of this (at least not in this example, I can imagine examples where other components need direct access to this map). In fact, to the knowledge base the SLAM system looks a bit like a stream transformer (i.e., it converts the output from the tracking system into a set of predicates). We may want to discuss the idea that components can act as stream transformers.

```
slam {
  in-annotated_images : (set(int, (int, int, int, int), bitmap)
  out-predicates : position_info
}
```

Listing 13: SLAM

I've also shown the SLAM system using a specialised type, position_info , which I'm thinking of as a particular set of predicates about the positions of the rover and any obstacles in some coordinate system. I assume the nature of position_info would be described by the shared ontology. This could be explored when Sheffield looks into the shared ontology building side of this.

4.2.5 The Pilot

We borrow the pilot from the Lego rover. It was shown in listing 1.

4.2.6 The Planner

Lastly the original configuration features a planner which will use descriptions of capabilities from the knowledge base in order to generate plans to meet the agent's goals. In all likelihood any planner will expect a problem description in its own input language (e.g., PDDL 2.1) but at present I'm assuming the planner can synthesize such a description from a goal and

set of capabilities taken from the knowledge base. The planner acquires it's capabilities by querying the knowledge base's configuration module. We indicate the that it has a client-server based connection to the knowledge base with the stream config which is shown as being both an input and an output stream. In reality this would obviously actually be two streams, an output stream upon which requests for information about capabilities would be placed and an input stream on which the information would be returned. A precondition for planning to work is that the planner has this connection. This is indicated by $\exists_c config$.

```
planner: {
  out-plans: plan
  in-out-config: set(capability)

  Capabilities {
    plan(+G:goal,-P:plan): plan<P>
     pre{∃_c config}
     post{plan_to_achieve(G,P)}
  }
}
```

Listing 14: Planner

The description is shown in listing 14.

4.2.7 Components of the Reconfigured System

When the system is reconfigured three new components are brought online. In our description of the simulator (Listing 15) I'm using some implicit notion of state - i.e., the post-condition states that $simulating$ becomes true. I then use this notion of simulating as a precondition to the data rate on the output stream. I don't provide any properties for the output stream position_info , more or less assuming that anything the agent needs to know can be deduced from the type. Note that the capability requires a set of mfiles as input – we assume that these are stored in the program_data part of the knowledge base and that the agent extracts these with a query to the knowledge base and passes them on to the simulator.

```
simulator {
  in-simc: commands
  out-pos: position_info

  Capabilities {
    simulate(+Dyn:set(mfile)):
       pre{∃_c simc}
       post{simulating}
  }

  Stream_Properties {
    rates:
      pos: simulating → [6s, ∞]
  }
}
```

Listing 15: Simulator

Listing 16 describes the anomaly detection component. This requires two sets of position_info in order to work correctly. It also needs an input list of commands. It can then compares the expected results of each command from the simulator with the actual results achieved by executing the command. The component generates a stream of tuples of a capability and the error matrix for that capability.

Lastly listing 17 describes the capability modifier which returns a new capability description given an existing description and an error matrix.

```
anomaly_detection {
  in-commands : command
  in-real_pos : position_info
  in-sim_pos : position_info
  out-error : (command, error_matrix)

  Stream_Properties {
     output properties:
       error<(C, E)> : {∃_c commands ∧ ∃_c real_pos ∧ ∃_c sim_pos →
                error_matrix_for(C, E)}
  }
}
```

Listing 16: Anomaly Detector

```
capability_modifier {
  out-capability: capability

  Capabilities {
    modify(+C:capability , +E:error_matrix , -C1:capability): capability<C1>
  }

}
```

Listing 17: Capability Modifier

4.3 Example 3: Intruder Avoidance in UAVs

This example considers the effect of two possible reconfigurations of an intruder avoidance system for UAVs. The initial configuration of the system is shown in Figure 6

The knowledge base gets information from GPS, Radar and FLIR. The Radar and FLIR data is filtered through a combination of stream transformers, the first of which determines if a collision is possible in the current environment, and the second of which determines if a collision is imminent and evasive action is required.

The agent has plans for reaching its destination and making maneuvers which involve calling a dedicated route planning system. This planning system returns routes that obey the Rules of the Air and may be used by the Auto Pilot.

The two reconfigurations we consider are, firstly, the effect of the Radar being taken offline. We assume, for this example, that an instruction comes from some trusted external source that the radar data is unreliable and not to be used. i.e., we are not concerned with how a decision is reached that the radar is unreliable. In this case the radar needs to be unlinked from the collision stream transformer and the transformer needs to be updated accordingly.

The second configuration is the situation in which the agent can not find a route for collision avoidance which obeys the rules of the air. As discussed in [2], we can reconfigure the system in order to select the "least bad" of the plans that break the rules of the air. This involves firstly reconfiguring the route planner so that it will output routes that break the rules of the air and then filtering these routes through something I've called the "Ethical Translator". This uses data about the agent's environment (presumably a map stored in program_data) to place ethical annotations on these routes indicating which, as a side effect, risk endangering life, or damaging property, or other concerns supplied to it by the agent. These annotations allow the agent then to select the least bad of the routes generated by the planner. This configuration is shown in figure 7

I don't intend to discuss this example in as much detail as the previous ones but to focus on the aspects that bring new features. Listing 18 shows the general configuration for the first case.

In the first configuration the system is using the first disjunct from the stream ontology definition of $collision$ to transform a merged stream coming from the radar and flir components (not described). $working(radar)$ and $working(flir)$ are used to indicate that the system considers those components to be functional. I've indicated the conditions for collision detection on these sensors as $p_r(radar)$ and $p_f(flir)$. This transformed stream, called collision is then *both* sent directly to the knowledge base *and* fed into the imminent_collision stream transformer. I'm not 100% convinced by the notation I'm using here. The definition of $imminent\_collision$ uses $\blacksquare^{ns}$ from [8] which means "all values on the stream in the last n seconds".

In the first reconfiguration, when the radar stops working the system can reconfigure itself by replacing the transformer generated from the first disjunct of collision to the second disjunct.

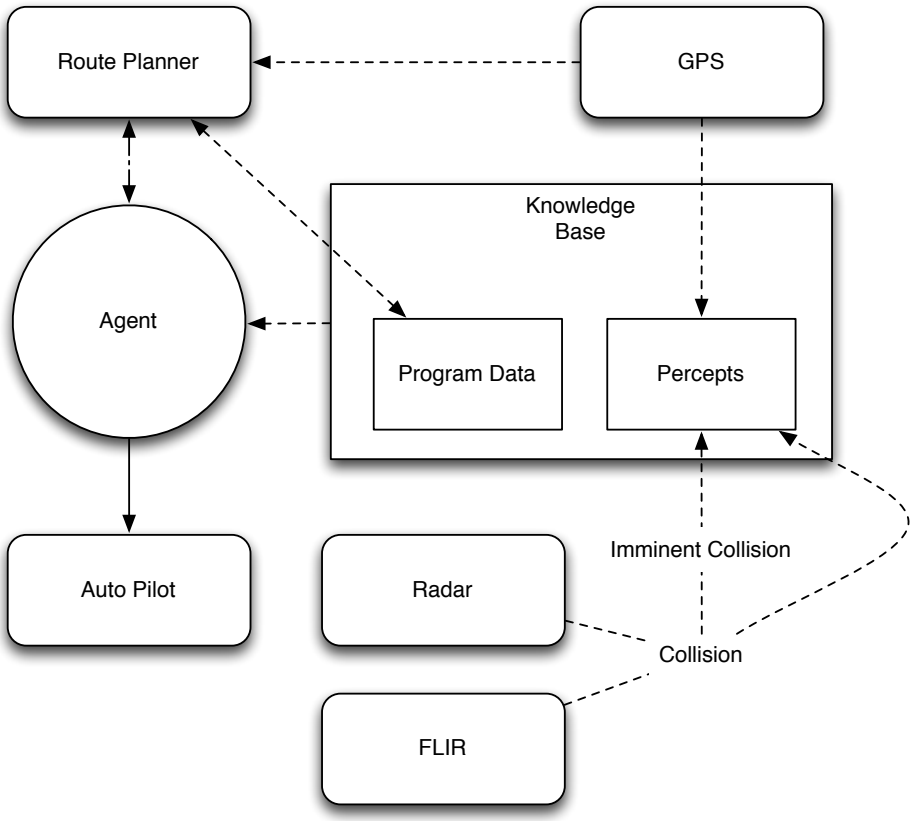

Figure 6: A UAV in its Default Configuration

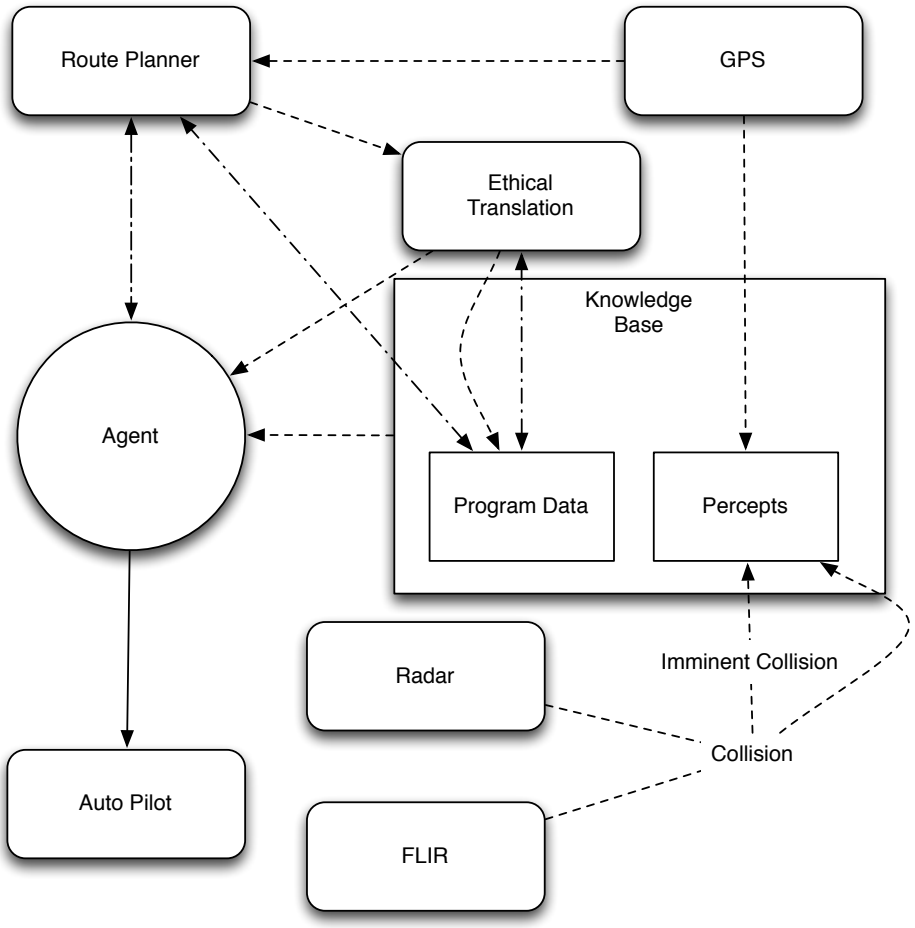

Figure 7: A UAV reconfigured with an “Ethical Planning System”

```
uav {
  Components {
    a: auto_pilot
    rp: route_planner
    et: ethical_translator
    gps: gps
    r: radar
    f: flir
  }

  Stream Ontology {
    collision: (collision ↔ working(radar) ∧ p_r(radar) ∧ working(flir) ∧ p_f(flir)) ⋁
               (working(radar) ∧ ¬working(flir) ∧ p'_r(radar))
    imminent_collision: imminent_collision ↔ ■^20s collision
  }

  Stream_Connections {
    r.radar -- --> merge_radar_flir
    f.flir -- --> merge_radar_flir
    merge_radar_flir -- | collision if p_r(radar) and p_r(flir) | --> collision
    collision -- --> percepts.literals
    collision -- | imminent_collision if imminent_collision | --> percepts.literals
    gps.gps -- --> percepts.literals
    gps.gps -- --> rp.gps
    rp.map <-- --> program_data.query
    rp.route -- --> program_data.routes
    rp.route_data -- --> percepts.literals
    agent.act[1] -- --> rp.command
    agent.act[2] -- --> a.command
    a.route <-- --> program_data.query
  }

  Agent_Interest {
    percepts.collision
    percepts.imminent_collision
    percepts.route_data
  }
}
```

Listing 18: Initial UAV Configuration

4.3.1 Route Planner

The route planner description is shown in listing 19. The route planner outputs tuples of route information together with declarative information about the route (e.g., an identifier for the route and its destination). We want the route to be stored in program_data while we want the information about it to go to the agent. However it seems plausible that the route data also needs to go to program_data - so that the route can be looked up by its identifier, so the stream routes outputs tuples of the route information together with the logical data, while the route_data stream just outputs route_data.

"Reconfiguration" of the route planner is achieved simply by calling different capabilities for route generation, one of which guarantees that all the generated routes will be compliant with the Rules of the Air. We use the notation $inR(S)$ to indicate the set formed of all the elements in the right-hand sides of the tuples in S.

```
route_planner {
  out-routes: set((route_info, route_data))
  out-route_data: set(route_data)
  in-out-map: map
  in-gps: gps

  Capabilities {
     roa-route(+D:coords) : routes<RS> route_data<RDS>
         pre{∃_c map ∧ ∃_c gps}
         post{∀(R, RD) ∈ RS. route_to(D, R) ∧ RoAcompliant(R) ∧ route_data_for(R, RD) ∧ RDS = inR(RS)}

    route(+D:coords) : routes<RS> route_data<RDS>
         pre{∃_c map ∧ ∃_c gps}
         post{∀(R, RD) ∈ RS. route_to(D, R) ∧ route_data_for(R, RD) ∧ RDS = inR(RS)}
  }
}
```

Listing 19: Route Planner

The ethical translator doesn't have capabilities. Every time it receives a set of routes it annotates them all against the current set of ethical concerns (which it can get by querying the knowledge base). This means we end up with output properties for the a_routes stream that reference input on other streams – e.g. that information has come in on routes and on concerns. I use the notation $stream[In]$ to represent input In arriving on stream, $stream$. The ethical translator is shown in listing 20. The output properties are here very opaque [7]. I also don't like the way we talk about the concerns on the concerns stream, though possibly I needn't worry about that and should just state that something is an ethical concern and ignore how the agent learns that fact.

```
ethical_translator {
  in-routes: set((route_info, predicate))
  in-out-concerns: set(ethical_concerns)
  out-a_routes: set((route_info, predicate))
  out-route_data: set(predicate)

  Stream_Properties {
      output properties:
          a_routes⟨RDS'⟩: ∀(R, RD) ∈ routes[RDS], ∃(R, RD') ∈ RDS'. RD' = ethics(RD, CA)∧
                          ∀C ∈ CA. concerns[CS] → C ∈ CS ∧ ethical_concern(R, C)
          route_data⟨RDS'⟩ : ∀(R, RD) ∈ routes[RDS], ∃RD' ∈ RDS'. RD' = ethics(RD, CA)∧
                             ∀C ∈ CA. concerns[CS] → C ∈ CS ∧ ethical_concern(R, C)
  }

}
```

Listing 20: Ethical Translator

[7] and incorrect since I don't say that all members of the output sets have to be annotated versions of the input.

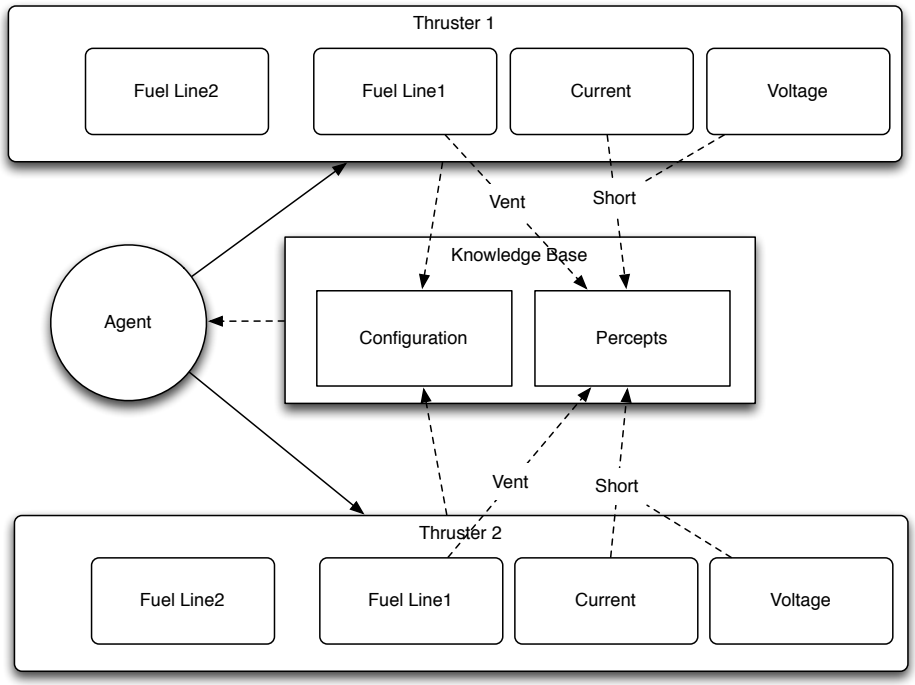

Figure 8: Satellite Thrusters

4.4 Example 4: Fuel Line Reconfiguration In Satellites

This example is based on a demonstrator we created for the Engineering Autonomous Space Software project which is discussed in [3]. We had a number of examples based around the diagnosis and repair of a thruster malfunction.

A structural view of a system with two thrusters [8] is shown in figure 8. Satellite thrusters can fail in two ways. Firstly one of their input fuel lines may break in which case the fuel pressure measured on that line will drop close to zero. This can be fixed by switching to a second, back-up, fuel line. Secondly there may be a short across the satellite grid which will result in the current increasing and the voltage decreasing to zero. This can generally be fixed by simply restarting the thruster. Figure 8 shows a configuration with two thrusters. Since their configuration is identical we will require parameterised stream transformers in the description. When a fuel line is switched the output stream from Fuel Line 2 must be connected to the knowledge base and the configuration will need to be updated accordingly. In this setting the responsibility for this lies with the thruster which internally reconfigures which internal component connects to its own pressure output stream, and then the thruster sends an update of its own description to the knowledge base.

Listing 21 shows the top level configuration for the system. The definitions for $vent$ and $short$ in the ontology are used twice, one for each thruster so the definition is parameterised by the name of the thruster. The streams coming out of the sensors are not parameterised by names, since these may come from third party vendors, but the predicate put on the stream leading out of the connection will have the thruster name.

The configuration for an individual thruster is shown in listing 22. The thruster is composed of sub-components representing the various sensors. It inherits its steam properties and capabilities from its sub-components.

We show a sample configuration for one of the sensors (a current sensor) in 23 this states that if the sensor is associated with thruster, T, then the output placed on the current stream is the current across T.

When the fuel line is switched, then the overall configuration of the system changes, and this need to be reflected in an update of configuration. This will probably consist of a completely new description for the thruster component. But I abbreviate this by noting that FB is swapped for FA in the **Stream_Properties** of the component description.

4.5 Example 5: 6DOF Robot Arm with a Range of Tools

This example is based on discussion with National Nuclear Labs aspects of which are discussed in [1]. The proposed description is for a 6DOF fixed robot arm. The arm can use a number of tools, in this example a scanner and a cutting tool. The arm has an internal control system which allows it to execute commands such as "move joint X to angle Y" and an internal planning system which can produce plans of joint activations that will move the head of the arm to some specific

[8] In reality the system should have a minimum of three thrusters but I think two is sufficient for this example.

```
satellite {
  Components {
     f1a: pressure_sensor
     f2a: pressure_sensor
     f1b: pressure_sensor
     f2b: pressure_sensor
     c1: current_sensor
     c2: current_sensor
     v1: voltage_sensor
     v2: voltage_sensor
     tc1: thruster_control
     tc2: thruster_control
     t1: thruster(f1a, f1b, c1, v1, tc1)
     t2: thruster(f2a, f2b, c2, v2, tc2)
  }

  Stream Ontology {
     vent(T): vent(T) ↔ fuel_pressure(T) < 1
     short(T): short(T) ↔ current(T) > 50 ∧ voltage(T) = 0
  }

  Stream_Connections {
    t1.fuel_pressure -- | vent(t1) if fuel_pressure < 1 | --> percepts.literals
    t2.fuel_pressure -- | vent(t2) if fuel_pressure < 1 | --> percepts.literals
    t1.current -- --> merge_t1_current_voltage
    t1.voltage -- --> merge_t1_current_voltage
    merge_t1_current_voltage --
              | short(t1) if current > 50 & voltage = 0 | --> percepts.literals
    t2.current -- --> merge_t2_current_voltage
    t2.voltage -- --> merge_t2_current_voltage
    merge_t2_current_voltage --
              | short(t2) if current > 50 & voltage = 0 | --> percepts.literals
    t1.config_update -- --> configuration.literals
    t2.config_update -- --> configuration.literals
    agent.act[1] -- --> t1.commands
    agent.act[2] -- --> t2.commands
  }

  Agent_Interest {
    percepts.vent(X)
    percepts.short(X)
  }
}
```

Listing 21: Satellite Configuration

```
thruster(FA: pressure_sensor, FB: pressure_sensor,
         C: current_sensor, V:voltage_sensor) {
  out-pressure: int
  out-current: int
  out-voltage: int
  out-config-update: thruster_description

  Components {
     FA, FB, C, B
  }

  Stream_Properties {
     rates:
        pressure : rate(FA.pressure)
        current : rate(C.current)
        voltage : rate(V.voltage)
        config-update : rate(TC.config-update)
     properties:
        pressure : FA.pressure
        current : C.current
        voltage : V.voltage
  }

  Capabilities {
    switch_fuel_line(-U): config-update<U>
      pre{vent(this)}
      post{¬vent(this) ∧ U = this.Stream_Properties(FA\FB)}

    restart:
      pre{short(this)}
      post{¬short(this)}
  }
}
```

Listing 22: Thruster

```
current_sensor {
  out-current: int

  Stream_Properties {
     properties:
       current<Val> : thruster(this, T) → current(T) = Val
  }
```

Listing 23: Current Sensor

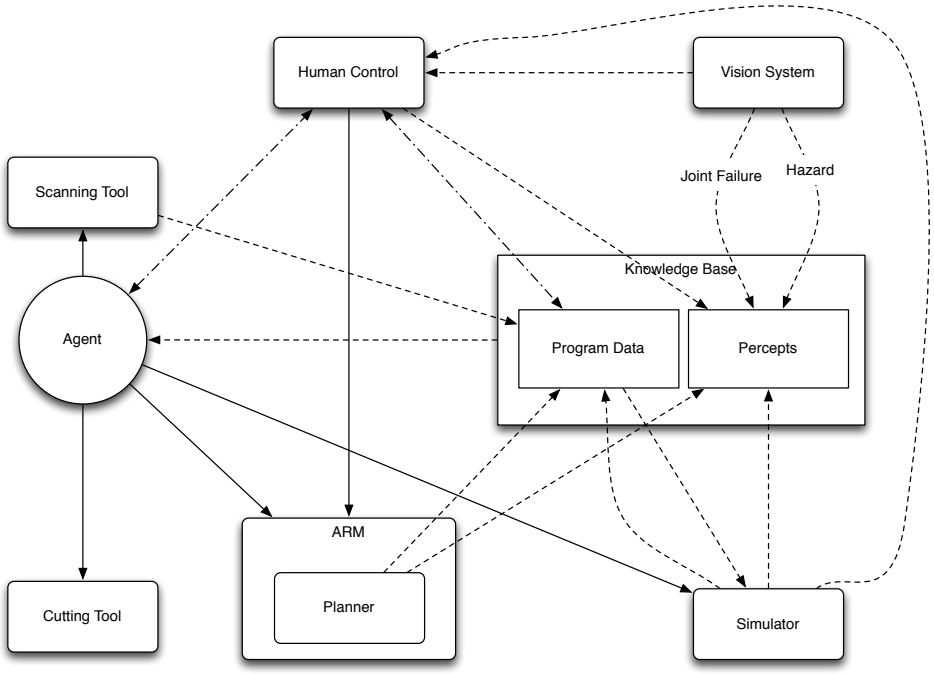

Figure 9: Robot Arm

position in space. The control system is capable of following these plans. We assume, as with the UAV example, that these plans are annotated with some logical information (final position and an identifier, for instance) that can be stored in percepts and made accessible to the agent. The arm's built-in planning system is unaware of the details of its environment. Therefore the plans it produces may result in collisions with external objects, specifically the piece of plant that it is required to examine or cut. Therefore the system also contains a simulator equipped with a model of the environment. This model can be updated based on information from the vision system. The simulator can execute the plans generated by the arm's planning system and determine which, if any, of the suggested plans are safe in the environment in which the arm finds itself.

Once plans have been evaluated for safety the agent selects a safe plan and requests its execution. The vision system continues to monitor the environment and if a new hazard moves into the range of the robot arm it sends this information to the percepts where it can be used both to update the simulator and to cause any plan executing on the arm to be stopped while safety is re-evaluated.

At any time a human may intervene and take control of the robot arm themselves.

This system is shown in figure 9.

There are three aspects of reconfigurability we would like to take into account here.

- The reconfiguration represented by switching between the cutting and the scanning tool.
- We want to handle degradation of the robot arm itself. We will treat a simple case where one of the joints becomes completely stuck. This can be observed by the vision system. At this point the internal planner on the arm can, probably, no longer be used since it will assume that all the joints are working and it can not be reconfigured. Therefore the simulator has to be pressed into action, updated with the joint failure information and then requested to produce plans from scratch using inverse kinematics. The situation were a joint becomes "sticky" in some way is much harder since I'm guessing it will involve internal changes to the control system in the arm. Some input from NNL may be useful if we want to seriously pursue that case.
- The issue of the human assuming control of the arm itself is a fairly major reconfiguration, especially since it bypasses the agent. For the purposes of this example we will treat both the human assuming direct control and the hierarchy of autonomy outlined in [10].

4.5.1 The Arm and its Internal Planner

I'll start with providing descriptions of the Arm and the Simulator.

The Arm is a composite components that represents both a control system for the underlying actuators, but also contains a planning system. Therefore some of its capabilities (i.e., get_to) depend on the pre and post conditions of the planner. We assume that joints are numbered and referred to by their number. The arm is shown in Listing 24.

```
arm(AP:arm_planner) {
    out-plans: set((plan, plan_data))
    out-tool_status: (const, bool)

    Components {
        AP: arm_planner
    }

    Capabilities {
      get_to(+X:double, +Y:double, +Z:double,
                -Plans:set((plan, plan_data)), -PD:set(plan_data)):
                plans<Plans>, predicates <PD>
          pre{AP.pre.get_to(X, Y, Z, Plans, PD)}
          post{AP.post.get_to(X, Y, Z, Plans, PD)}

      follow_plan(+Plan:plan):
          duration{}
          pre{at(X_1, Y_1, Z_1)_s ∧ plan_for(X_1, Y_1, Z_1, X, Y, Z, Plan)_s}
          post{at(X, Y, Z)_e}

      move(+J:int, +A:double):
          pre{in_range(J, A)_s ∧ current_angle(J, A')_s}
          post{current_angle(J, A' + A)_e}

      pick-up(+Tool:const): predicates<(Tool, true)>
          pre{at(X, Y, Z) ∧ at(Tool, X, Y, Z)}
          post{hold(Tool)}

      put-down(+Tool:const): predicates<(Tool, false)>
          pre{at(X, Y, Z) ∧ hold(Tool)}
          post{¬hold(Tool) ∧ at(Tool, X, Y, Z)}
    }
}
```

Listing 24: Arm

The description of the arm's internal planner in listing 25 is similar to the description of the route planner in Example 3. It generates plans to get so some specific position, generates a description of the plan assigning (at the least) an identifier to the plan, and outputs streams of both the plan paired with the data and the plan data on its own. This allows the existence of the plan to be noted in percepts and reasoned about by the agent but, at the appropriate moment, the plan itself can be extracted from program_data by using the identifier.

```
arm_planner {
    out-plans: set((plan, plan_data))
    out-plan_data: set(plan_data)

    Capabilities {
        get_to(+X:double, +Y:double, +Z:double,
                    -Plans:set(plan, plan_data), -PD:plan_data):
                    plans<Plans>, plan_data<PD>
            pre{at(X_1, Y_1, Z_1) ∧ all_joints_working}
            post{∀(P, P_d) ∈ Plans.plan_to(X_1, Y_1, Z_1, X, Y, Z, P) ∧ plan_data_for(P, P_d) ∧ PD = inR(Plans)}
    }
}
```

Listing 25: Arm Planner

4.5.2 Reconfiguration of the Tools

Reconfiguration, at the level of access to the tools is fairly simple.

I'm assuming that the scanning and cutting tools can not be specifically activated or de-activated by the robot arm. In fact I'm assuming the cutting tool is entirely passive while the scanning tool is always on and transmits (?wirelessly) its information to the agent.

So this turns from a reconfiguration problem to one of knowing where the scanning and cutting tools are. We already see that one of the post-conditions in the arm is $hold(Tool)$ and this information is placed on the stream that goes into percepts. The description of the scanning tool is show in listing 26.

```
scanner {
    out-radiation : int

    Stream_Properties {
        properties:
            radiation<Val>: at(scanner, X, Y, Z) → radiation_level(X, Y, Z, Val)
    }


}
```

Listing 26: Scanning Tool

4.5.3 Reconfiguration of the Arm Planners

The simulator's job is to check the safety of plans provided by the arm's planner and, if a joint is stuck in the arm to generate plans itself. Therefore it has a capability get_to which is almost the same as the arm planner's, except that it only outputs safe plans. It's other capability outputs a set of safe plans and we assume it does this by querying program_data for plans. The description of the simulator is given in 27. We don't indicate anywhere that the simulator is considered less accurate/efficient than the one shipped with the arm, but we should probably show this somehow.

4.5.4 The Human assumes Control

[10] recommends four levels for varying autonomy:

- Computer does everything autonomously

```
arm_simulator {
    out-safeplan: set(plan_data)
    in-out-current_plans: set(plan_data)
    in-out-jointinfo: joint_info

    Capabilities {
        check_safe(-P: set(plan_data)): safeplan<Plans>
            pre{at(X_1, Y_1, Z_1) ∧ connected(current_plans)}
            post{ ∀(P, P_d) ∈ Plans. plan_for(X_1, Y_1, Z_1, X, Y, Z, P) ∧ plan_data_for(P, P_d) ∧ safe(P_d)
                                                                          ∧PD = inR(Plans) }

      get_to(+X:double, +Y:double, +Z:double, +J:joint_info,
                        -Plans:set(plan, plan_data), -PD(plan_data)):
                        plans<Plans>, plan_data<PD>
          pre{at(X_1, Y_1, Z_1)}
          post{ ∀(P, P_d) ∈ Plans. plan_for(X_1, Y_1, Z_1, X, Y, Z, P) ∧ plan_data_for(P, P_d) ∧ safe(P_d)
                                                                       ∧PD = inR(Plans) }
    }
}
```

Listing 27: Arm Simulator

- Computer chooses action, performs it and informs human
- Computer chooses action and performs it unless human disapproves (timeout)
- Computer chooses action and performs it if human approves

To this we need to add full human control. At some point we need to consider the autonomous switching between at least the four levels in [10], but for the time being we will assume that the human can set the level via some switch and this is recorded in percepts.

This means that most of the time the interface a human uses will be monitoring input streams from the vision system and simulator. It can query the program data for planned routes. Depending upon the autonomy level the agent can report its actions to the human control or query the human control for approval. Only when the human switched to fully human operation will the agent disconnect its own connections to the command interfaces of the arm and simulator and replace those connections with connections to the control. We show the human interface in listing 28. This has no stream properties or capabilities since it is all under the control of the human.

```
human {
  in-images: bitmap
  in-simulation: bitmap
  in-out-plan_info: program_data
  in-out-agent: command
  out-arm-command: command
  out-simulator-command: command
  out-setlevel: int
}
```

Listing 28: Human Control Interface

A human taking control, under this scheme, does not result in any change of internal set-up or descriptions, though it does make a difference to the behaviour of the agent.

4.6 Example 7: Fuel System Health Management in UAVs

In this example we consider reconfiguration of the level of abstraction of information that is being passed around the system. We consider the example of the health management of the fuel system within a UAV. The fuel health management system has no knowledge of mission objectives, routes, flight phase or the external environment. It monitors fuel levels, pressure and other sensors and uses this firstly, to transmit warnings that there are issues within the fuel system and secondly to suggest

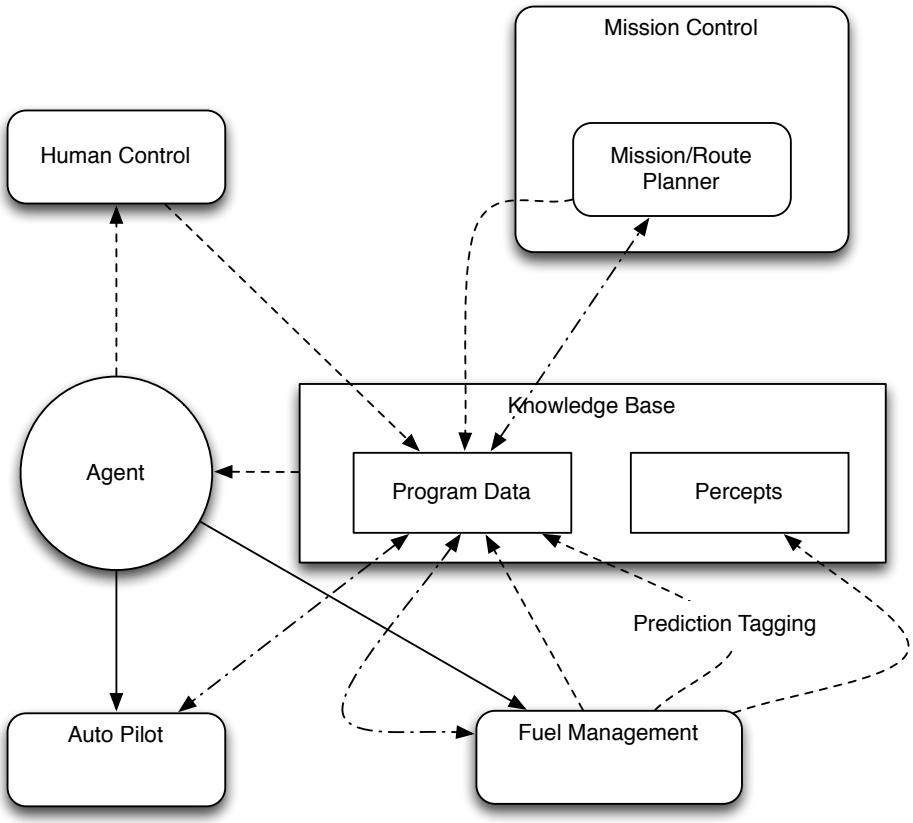

Figure 10: Fuel Health Management

measures that can be taken to counter any problems. These suggestions come with predictions about the behaviour of fuel and vehicle balance over time if the suggestion is followed. The predictions offered by the fuel health management system are dependent upon the current mission plan which tends to make everything a little circular, since they may need to be revised if the mission control revises the plan based on the management system's predictions. The impact of faults in the fuel management system are evaluated as they occur. Depending upon the flight phase, there may be a necessity for rapid action, or it may be possible to make a more considered response. Whichever subsystems are responsible for mission planning and route planning will need to be notified of the changed environment. The system is shown in figure 10 and Listing 29. I think the only slightly confusing bit is my abuse of λ-calculus notation to show that the tuple of functions being returned by the fuel health system are tagged before going into program_data. Given we don't yet know how program_data is structured or how we generate these transformers from the language in [8], there doesn't seem to be a lot of point in going into this further. It is just a note that tagging has to take place somehow.

There are two aspects of reconfiguration that we wish to consider here. First, there is the issue of the reconfigurations required by the malfunction in the fuel system, this involves the actions taken to manage the problem and the propagation of information about the effects of those actions around the system. This can be done without reconfiguring the system itself so I won't discuss it in detail here, but it may have interesting reasoning aspects we want to consider at a later date.

The second reconfiguration we wish to consider is the replacement, or update of the fuel health management system so that instead of reporting the effects of changes as a function over time, it reports instead the effects of the changes at particular flight phases in the current plan being enacted by the system.

Listing 30 shows the mission planner component which produces plans (annotated by plan data as in several of our other examples) which achieve some Goal (e.g., to reach a particular airfield). As part of determining the plans the planner can access predictions about performance from the knowledge base (potentially not just predictions supplied by the fuel system). These predictions are functions over time, tagged in some way by the property (e.g., trim, fuel remaining) that they purport to describe.

Listing 31 shows the fuel health management system itself as a component that can output predictions, warning and metrics (note: not expanded upon).

A change of fuel management system so that its predictions varied not with time but with flight phase is quite a difficult change to consider. At a trivial level it means altering the type of the predictions from time $->$ double to flightPhase $->$ double

```
uav {
    Components {
        h : human
        mc : mission_control
        ap : auto-pilot
        fh : fuel_health
    }

    Stream_Connections {
        ag.act[1] -- --> h.warning
        ag.act[2] -- --> ap.command
        ag.act[3] -- --> fh.command
        h.mission_plan -- --> program_data.plans
        mc.mission_plan -- --> program_data.plans
        mc.predictions <-- --> program_data.query
        ap.mission_plan <-- --> program_data.query
        fh.mission_plan <-- --> program_data.query
        fh.metrics -- --> program_data.metrics
        fh.predictions -- | λ(f1, f2).trim : f1, fuel_remaining : f2 |
                                                --> program_data.predictions
        fh.warning -- --> percepts.literals
    }

    Agent_Interest {
        percepts.warning(W)
    }
}
```

Listing 29: Fuel Health Management System

```
mission_planner {
    out-mission_plan: set((plan, plan_data))
    out-plan_data: set(plan_data)
    in-out-predictions: set((const,time->double))

    Capabilities {
        mission_plan(-PS:set((plan, plan_data), -PD:set(plan_data), +G:goal):
                                                mission_plans<PS>, plan_data<PD>
            pre{∃c predictions}
            post{∀P ∈ PS. achieves(G, P) ∧ RoAcompliant(P)}
    }
}
```

Listing 30: Mission Planner

```
fuel_health {
   out-metrics: metrics
   out-predictions: (time -> double, time -> double)
   out-warning: predicate

   Stream_Properties {
       output properties:
           predictions⟨(F1, F2)⟩ :
                   ∀T : trim(plane, T, F1(T)) ∧ fuel_remaining(plane, T, F2(T))
           warning⟨W⟩ : warning(W)
   }
}
```

Listing 31: Fuel Health Management System

and we could just assume that the mission planner can simply cope with such a change. However even at this fairly trivial level we would need to have a more generic type signature for **in**−**out**−predictions. A more realistic reconfiguration might be to assume that both fuel health system and mission planner are changed.

Listing 32 shows a mission control component that contains a reconfigurable mission planner. In this case the type of the predictions stream is derived from the type of the mission planner's predictions stream. At reconfiguration the mission planner component would be replaced and streams reconnected appropriately.

```
mission_control(MP: mission_planner){
   out-mission_plan: set((plan, plan_data))
   out-plan_data: set(plan_data)
   in-out-predictions: MC.predictions.type

   Components {
     MC: mission_planner
   }

   Capabilities {
      mission_plan(-PS:set((plan, plan_data), -PD:set(plan_data), +G:goal):
        pre{pre(MC.mission_plan(PS, PD, G)}
        post{post(MC.mission_plan(PS, PD, G))}
   }

  Stream_Connections {
     predictions <-- --> MC.predictions
     MC.mission_plan -- --> mission_plan
     MC.plan_data -- --> plan_data
  }

}
```

Listing 32: Mission Control

Another alternative is to assume that the mission planner can not reason about predictions based on flight phase in which case the agent will need to either possess or create an intermediate component that can transform flightPhase based predictions into time based predictions. I've not considered this here, but I think it would be an interesting synthesis challenge.

A Language Definition

A.1 Syntax

⟨system⟩ ⊨ ⟨rais_const⟩ {⟨component_desc⟩? ⟨stream_ontology⟩? ⟨streamconnections⟩? ⟨interests⟩?}
⟨component⟩ ∗

⟨component_desc⟩ ⊨ **Components** {(⟨rais_const⟩ : ⟨rais_ground_term⟩) + }

⟨stream_ontology⟩ ⊨ **Stream Ontology** {(⟨rais_iff_exp⟩) + }

⟨streamconnections⟩ ⊨ **Stream_Connections** {⟨connection⟩ + }

⟨interests⟩ ⊨ **Agent_Interest** {((percepts | configuration).⟨rais_term⟩) + }

⟨alexei⟩ ⊨ *Something translatable into Alexei's temporal logic for stream processing*

⟨connection⟩ ⊨ (⟨one_way_connection⟩ | ⟨query_connection⟩)

⟨one_way_connection⟩ ⊨ ⟨out_stream_id⟩ -- (⟨stream_transformation⟩)? --> ⟨in_stream_id⟩

⟨query_connection⟩ ⊨ (⟨component_stream⟩ | ⟨agent_in_stream⟩) <-- --> ⟨query_stream⟩

⟨out_stream_id⟩ ⊨ (⟨component_stream⟩ | ⟨agent_action⟩ | ⟨connection_stream⟩)

⟨stream_transformation⟩ ⊨ | ⟨alexei⟩ |

⟨in_stream_id⟩ ⊨ (⟨component_stream⟩ | ⟨agent_in_stream⟩ | ⟨kb_in_stream⟩ | ⟨command_stream⟩ | ⟨connection_stream⟩)

⟨component_stream⟩ ⊨ (⟨rais_const⟩ | ⟨rais_var⟩).⟨rais_const⟩

⟨agent_in_stream⟩ ⊨ agent.(beliefs | goals | plans | messages)

⟨query_stream⟩ ⊨ ⟨kb_module_name⟩.query

⟨agent_action⟩ ⊨ agent.act[⟨int⟩]

⟨kb_in_stream⟩ ⊨ (percepts.literals | configuration.literals | program_data.⟨rais_const⟩)

⟨command_stream⟩ ⊨ ⟨rais_const⟩.command

⟨connection_stream⟩ ⊨ ⟨rais_const⟩

⟨kb_module_name⟩ ⊨ percepts | configuration | program_data

⟨component⟩ ⊨ ⟨rais_const⟩((⟨component_var_desc⟩(,⟨component_var_desc⟩) *))? ({(⟨stream⟩) * (⟨component_var_desc⟩)? (⟨capabilities⟩)? (⟨stream_properties⟩)? (⟨streamconnections⟩)?}

⟨component_var_desc⟩ ⊨ **Components** {(⟨rais_var⟩ : ⟨rais_type⟩) + }

⟨stream⟩ ⊨ (**in** | **out** | **in−out**)−⟨rais_const⟩ : (⟨rais_type⟩ | ⟨rais_var⟩.⟨rais_const⟩.type)

⟨capabilities⟩ ⊨ **Capabilities**{⟨capability⟩ + }

⟨stream_properties⟩ ⊨ **Stream_Properties**{(**rates**:⟨rate_exp⟩+) * (**output properties**:⟨property_exp⟩+) * }

⟨capability⟩ ⊨ ⟨cap_name⟩ : (⟨out_stream⟩) * ⟨cap_conditions⟩

⟨rate_exp⟩ ⊨ (⟨rais_const⟩ : ⟨rate⟩ | ⟨rais_logic⟩→⟨rate_exp⟩)

⟨property_exp⟩ ⊨ (⟨base_property_exp⟩ | ⟨inherited_property_exp⟩)((prob : ⟨int⟩%))?

⟨base_property_exp⟩ ⊨ (⟨out_stream⟩(,⟨out_stream⟩) * : ⟨rais_logic⟩ | ⟨rais_const⟩ . ⟨rais_const⟩)

⟨inherited_property_exp⟩ ⊨ ⟨rais_var⟩ . ⟨rais_const⟩

⟨cap_name⟩ ⊨ ⟨rais_const⟩(((+ | −)?⟨rais_var⟩ : ⟨rais_type⟩(,(+ | −)? ⟨rais_var⟩ : ⟨rais_type⟩) *))?

⟨out_stream⟩ ⊨ ⟨rais_const⟩<⟨rais_var⟩>

⟨cap_condition⟩ ⊨ *Something that can be mapped to PDDL 2.1 action descriptions*

⟨rate⟩ ⊨ rate(⟨rais_var⟩ . ⟨rais_const⟩) | ⟨time_interval⟩

⟨time_interval⟩ ⊨ [⟨double⟩(s | m),(⟨double⟩(s | m) | ∞)]

⟨rais_logic⟩ ⊨ *FOL plus some distinguished predicates*

⟨rais_iff_exp⟩ ⊨ ⟨rais_term⟩↔⟨rais_logic⟩

⟨rais_term⟩ ⊨ ⟨const⟩(((⟨const⟩ | ⟨var⟩)(, (⟨const⟩ | ⟨var⟩)) *))?

⟨rais_ground_term⟩ ⊨ ⟨const⟩((⟨const⟩(, ⟨const⟩) *))?

⟨rais_const⟩ ⊨ (a...z1...0) +

⟨rais_var⟩ ⊨ A...Z(a...z1...0) *

⟨rais_type⟩ ⊨ ⟨rais_term⟩ | (⟨rais_type⟩(, ⟨rais_type⟩) +)

$\langle \text{double} \rangle \models$ *A double*

$\langle \text{int} \rangle \models$ *An integer*

References

[1] Jonathan M. Aitken, Affan Shaukat, Elisa Cucco, Louise A. Dennis, Sandor M. Veres, Yang Gao, Michael Fisher, Jeffrey A. Kuo, Thomas Robinson, and Paul E. Mort. Autonomous nuclear waste management. *IEEE Intelligent Systems*, 2017. In Press.

[2] Louise Dennis, Michael Fisher, Marija Slavkovik, and Matt Webster. Formal Verification of Ethical Choices in Autonomous Systems. *Robotics and Autonomous Systems*, 77:1–14, mar 2016.

[3] Louise A. Dennis, Michael Fisher, Nicholas K. Lincoln, Alexei Lisitsa, and Sandor M. Veres. Practical verification of decision-making in agent-based autonomous systems. *Automated Software Engineering*, 23(3):305–359, 2016.

[4] George Edwards, Joshua Garcia, Hossein Tajalli, Daniel Popescu, Nenad Medvidovic, Gaurav Sukhatme, and Brad Petrus. Architecture-driven self-adaptation and self-management in robotics systems. In *SEAMS'09*, pages 142–151, 2009.

[5] Maria Fox and Derek Long. PDDL+: Modeling continuous time dependent effects. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002.

[6] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[7] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[8] Alexei Lisitsa. Temporal logic for data stream queries, sensing abstractions and hierarchical control. working note. Unfinished Paper.

[9] N.R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 178 –187, 2000.

[10] Charles Patchett. Autonomous control of UASs: Architecture considerations for an intelligent air systems. Presentation.

[11] Dumitru Roma, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Web service modeling ontology. *Applied Ontology*, 1:77–106, 2005.

[12] Affan Shaukat and Yang Gao. Model learning for reconfigurable autonomous systems: A case study. Internal Technical Report for Reconfigurable Autonomy, 2013.

[13] S.M. Veres. *Natural Language Programming of Agents and Robotic Devices*. SysBrain Ltd., 2008.