Exploiting Patterns of Interaction to Achieve Reactive Behavior

D.M. Lyons & A.J. Hendriks

Philips Laboratories,

Philips Electronics North America Corporation,

Briarcliff Manor, NY 10510

ABSTRACT

This paper introduces an approach that allows an agent to exploit *inherent patterns of interaction in its environment, so-called dynamics, to achieve its objectives. The approach extends the standard treatment of planning and (re)action in which part of the input to the plan generation algorithm is a set of basic actions and perhaps some domain axioms. Real world actions are typically difficult to categorize consistently and are highly context dependent. The approach presented here takes as input a procedural model of the agent's environment and produces as output a set of action descriptions that capture how the agent can exploit the dynamics in the environment. An agent constructed with this approach can utilize context sensitive actions, "servo" style actions, and other intuitively efficient ways to manipulate its environment.*

A process-algebra based representation, $\mathcal{RS}$, is introduced to model the environment and the agent's reactions. The paper demonstrates how to analyze an $\mathcal{RS}$ environment model so as to automatically generate a set of potentially useful dynamics and convert these to action descriptions. The output action descriptions are designed to be input to an Interval Temporal Logic based planner. A series of examples of reaction construction drawn from the kitting robot *domain is worked through, and the prototype implementation of the approach described.*

1 Introduction

In this paper we address the problem of constructing an agent capable of producing intelligent action in an uncertain and changing environment. The challenge the agent faces is to carry out tasks given new opportunities or difficulties that arise in the environment. The approach we propose is to construct agents that *exploit* inherent patterns of interaction in the environment to achieve their objectives. Chapman [6] calls such a pattern a *dynamic,* and that usage is adopted here also. A dynamic may not be entirely controllable nor predictable by the agent; nonetheless, the agent can work with it to achieve its objectives in an intuitively efficient manner.

In previous work, we have developed the planner-reactor architecture as an approach to building such agents [19]. The reactor component of the planner-reactor handles uncertainty in the environment using reactions — hard-wired combinations of sensing and action — to "servo" out uncertainty. The planner component provides the deliberative capabilities to update the reactor component to best suit the current environment. It accomplishes this by adding new, or removing old reactions. The planner is implemented using an Interval Temporal Logic (ITL) language [24]; the reactor, using the $\mathcal{RS}$ language [17]. This paper presents an approach to the planner problem of constructing novel reactions that allow the agent to exploit the dynamics in its environment. The specific focus of the paper is on the problem of identifying dynamics that can be useful for building reactions. This extends the standard treatment of planning and (re)action where part of the input to the plan generation algorithm is a set of basic actions and (perhaps) some domain axioms. The set of actions, however, is normally invariant over the encountered instantiations of the problem domain.

The proposed approach, in contrast, derives additional, context-sensitive actions from an analysis of dynamics in the environment and how they interact with the basic sensory and motor actions available to the agent. A *procedural model of the environment*, representing the interactions between parts of the environment and the agent, is required for this analysis. The result of the analysis is a set of high-level action descriptions suitable for the ITL-based planner. Although the planner constructs reactions from these action schemas in a straightforward manner, the action schemas themselves ensure that the generated reactions can exploit context-sensitive actions, "servo" style actions, and other intuitively efficient ways to manipulate the environment.

Like Horswill [13], our objective is to allow the agent to take advantage of special features of the environment. His *habitat constraints* allow a designer to simplify the design of an agent so as to exploit its 'niche' in the environment. Our approach is an attempt to automate the identification and use of a class of such constraints, namely constraints that express how an agent can exploit processes at work in the environment.

The proposed method for generating action schemas has a number of advantages. The construction of a good reactive system depends on the designer understanding the kinds of activities in which the environment engages. This information is not captured well by the classical planning concept of a set of basic action descriptions. Real actions for dealing with dynamic events are typically highly context dependent and difficult to categorize into a discrete set in a consistent and useful way. The proposed approach can handle such unseparable and context sensitive actions, since the dynamic provides the separation and the context of the action.

The remainder of this paper is laid out as follows. In the next two sections our previous work on the planner-reactor framework is introduced, and the motivation for investigating the problem of identifying and exploiting dynamics discussed. Section 3 introduces the concept of dynamics and of exploiting dynamics to achieve objectives. It includes a motivating example from our application domain, the kitting robot. Section 4 introduces the two formal representations used in the paper: the $\mathcal{RS}$ language and Interval Temporal Logic. This section explains the mechanisms for identifying dynamics and selecting reactions. Section 5 uses this formal framework to investigate exploiting dynamics in three kitting domain problems:

visual tracking, guarded motion, and interaction with a conveyor belt. Section 6 describes the implementation of the approach.

2 The Planner Reactor Framework

Reactive systems, e.g., (Brooks 1986) [5], (Agre & Chapman 1987) [1], were introduced to address the fragility of classical approaches to plan generation and execution in an uncertain and changing environment. Previous work in constructing reactions has occurred in two fields: Learning better reactions based on feedback from the environment, e.g., DYNA[31], MD [7]; and the generation of reactive plans based on a model of the environment, e.g., the Universal Plan generator [25], ERE [4]. We are concerned with this latter area: constructing reactions or reactive plans. One solution to this problem has been to construct 'off-line' systems, such as Kaelbling's GAPPS [14] and Schoppers' Universal Plan generator [25], that automatically generate reactive machines to suit a goal specification. Another response was to build systems, such as Hendler's APE [29], Arkin's AURA [3] and Bresina & Drummond's ERE [4], that integrated reactive and deliberative capabilities.

Our application domain is robotic kitting [27]. A kitting robot is a robot system that puts together assembly kits — trays containing all the necessary parts from which to build a specific product. Simpler and cheaper automation can construct the assemblies once they have been placed in the kits and routed appropriately. The key characteristic of the kitting robot is not so much its ability to reason about assembly, but rather its ability to choose timely and effective actions to suit an uncertain and changing environment. To deal with events as they happen, the kitting robot needs to be capable of *real-time, reactive response.* On the other hand, to deal with changes in the environment (e.g., bursts of errors in parts, or failures of resources) or new goals issued by factory management, the kitting robot needs to be able *to plan ahead 'on the fly'.*

The definition of planning that we espouse is somewhat similar to that of Bresina & Drummond[4] and McDermott[23]: a planner is a module that continually modifies a concurrent and separate reactive system so that its behavior becomes more goal directed. The reactive component of the system will be referred to as the *reactor* after the terminology established by Bresina & Drummond. Figure 1 illustrates our *Planner-Reactor* architecture. A reactor, a network of sensory-motor reactions, interacts directly with an uncertain and changing environment. The planner is a concurrent and separate module that monitors the reactor and its environment by issuing perception requests to the reactor. (These are distinct from the reactor's sensory actions.) When, in the planner's view, the reactor structure will not produce behavior that conforms to the planner's goals, the planner issues adaptations, structural changes to the reactor, to bring reactor behavior back into line with the goals. The planner can generate novel reactions and "add" them into the reactor, as well as remove existing reactions from the reactor, in a safe manner. Planner and reactor communicate asynchronously.

The specific focus of this paper will be on the planner problem of generating novel re-

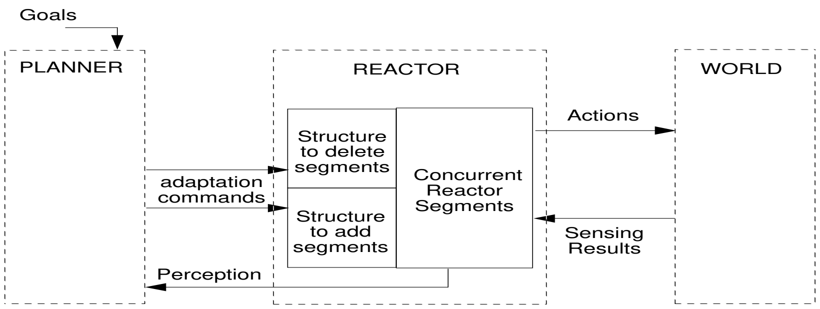

Figure 1: The Planner-Reactor Architecture.

actions for the reactor that exploit the dynamics in the environment. However, to explain the motivation for addressing this problem, a little additional detail on the Planner-Reactor work is necessary. The "full story" can be found in [19, 20, 21].

2.1 Reactor

The reactor contains a network of reactions. A reaction is a sensory process (or network) composed with an effector process (or network) in such a fashion that should the sensory process detect its triggering condition, then it will in turn trigger the effector process. The key property of the reactor is that it can produce action at *any* time. Unlike a plan executor, a reactor can act independently of the planner; it is always actively inspecting the world, and will act should one of its reactions be triggered. A reactor should produce timely, useful behavior even without a planner.

2.2 Planner

The planner is completely separate from, and concurrent with, the reactor. Rather than seeing the planner as a higher-level module that loads plans into a lower-level executor, we see the planner as an equal-level module that continually tunes the reactor to produce appropriate behavior (Figure 1). The interaction between the planner and reactor is entirely asynchronous.

The input to the planner includes: a model of the environment in which the planner is operating; a description of the reactor's structure; and information from the user about objectives the reactor should achieve (e.g., in the kitting problem domain, geometric goals such as kit layouts) and constraints the reactor should obey in its behavior (e.g., batch mix or resource usage constraints). The planner continually determines if the reactor's responses to the current environment would indeed conform to the objectives. If not, then the planner makes an incremental change to the reactor configuration — adding in novel reactions, or deleting old reactions —to bring the reactor's behavior more into line with the objectives. The planner module has been implemented in a conventional manner using an ITL interval reasoner [19]. However, we found that approach to be somewhat limiting, as section 2.4 will report, and this motivated the study of the approach proposed in this paper.

2.3 Planner-Reactor Interactions

As indicated in Figure 1, there are two routes of interaction between planner and reactor. An adaptation is essentially an instruction to delete part of the reactor structure or to add in some extra structure. Each individual adaptation should be small in effect and scope; large changes in reactor behavior only come about as the result of iterated adaptation. Large adaptations would be equivalent to downloading 'plans' to the reactor; something we need to avoid.

Perceptions are sensory data collected by the reactor to be sent to the planner. The knowledge needs of the planner and reactor are almost always different. The reactor uses sensory data to determine whether to fire its reactions. The planner needs sensory data to allow it to predict the future progress of the environment and the status of the reactor.

In [22] we demonstrate how it is possible to construct a reactor that can be adapted in a simple and concise manner. In [20] we describe how to make these adaptations in a safe manner despite the fact that the planner does know when or which reactions might be firing. In [19] we formalize the iterative improvement of a reactor by a planner to explore convergence. This latter paper gives the architecture of the ITL-based planner.

2.4 Generating Reactions

The planner employed in the Planner-Reactor work [19] has some special features: it needs to consider repeated improvement of a reactor, and it reasons about building reactions as process networks. Nonetheless, it is conventional in that it expects a set of action schema, describing the basic domain actions, as part of its input.

Our experience was that this input severely restricted the reactions that could be devised by the planner. Many of the potentially useful modes of interaction with the environment were hidden by the choice of domain actions. One approach to solve this problem is to develop a very general set of basic actions. This is far from trivial, since real actions for dealing with dynamic events are typically highly context dependent and difficult to categorize into a discrete set in a consistent and useful way.

An alternate approach is to give the system a model of the environment and ask it to identify useful modes of interaction with its environment to achieve a given goal. This environment model needs to capture the interaction topology between parts of the environment as well as between the environment and the agent. This approach has the additional advantage that the designer's focus then becomes the building an accurate environment model, rather than the devising of the correct view or interpretation of the environment for the agent, as seen through a set of action descriptions. It also has the advantage that the devised actions do not need to be consistent across different tasks, since they are extracted from the environment model on a per task basis, or even in the same task over time, since the processes at work in the environment may change over time.

The remainder of this paper is based on this second approach. A method will be introduced for automatically identifying dynamics, and for generating action descriptions based on the identified dynamics, which can be given as input to our ITL-based planner.

3 Planning to Reactively Exploit the Environment

3.1 Modeling the Environment as a Set of Processes

We will represent the environment model as a set of concurrent, communicating *processes* (in the Computer Science sense of the word). The key information that this procedural model captures is the interaction topology: what parts of the environment 'are connected to' other parts and how 'messages' are passed along those communication links. This kind of world model does *not* contain information about the current state of the world[1] such as e.g., `(at-position object26 0.25 0.34 0.29)`; rather it contains information about how objects interact with each other, or how an object interacts with the agent's visual input sensors. In other words, the model describes the structural and temporal regularities, the patterns of interaction in the environment and how they evolve over time. The environment model may be partial and/or may contain uncertain information.

Modeling a dynamic environment as a set of processes is analogous to the approach taken by Hendrix[11] and in Qualitative Process Theory [9]. The sole mechanism assumption of QP-theory (but equally valid for Hendrix's work) states that all changes in (physical) systems are caused directly or indirectly by processes. Our modeling of the environment also subscribes to this assumption. This modeling only restricts the amount of interaction. It does not necessarily prescribe it, since the processes may contain uncertainty, e.g., random motion of objects, or the unpredictable occurrence of events such as a new object being introduced.

3.2 Exploiting Dynamics

Constructing agents that *reactively exploit* their environment to achieve their objectives differs in some important aspects from the standard approach of building agents that plan and act to achieve their goals. In the latter approach, actions are considered similar to building blocks, which must be composed together in some ordering so that the agent can force the environment to a desired state. The first consequence of this viewpoint is that, because it demands well-defined 'blocks', it guides a designer to choose actions that have a one to one mapping with effects on the environment.

Unfortunately, the actions typically available to an agent are not so well behaved — their effects are usually context dependent and they may indeed produce uncertain effects. Proper modeling of the effects and applicability conditions of such actions is very difficult for a designer; it may require elaborate descriptions encompassing extensive qualification axioms

[1] At least, not primarily.

(e.g. Ginsburg [10]) or Causal Theories (Shoham [28]). More importantly, in this approach to action modeling the range of environments in which the agent operates must be known a priori, and an encountered environment which is slightly different can render such a model incorrect.

The second consequence is that it leads a designer to naturally considering the space of all compositions of the building blocks as the space in which to look for plans, i.e., tasks which require a servoing behavior, achieving an objective through repetitive actions, are rarely considered. Schoppers [26] has begun to address this type of behavior. However, he chose to make the dynamics of the environment explicit in his action descriptions in terms of temporal *soon* and *sust* operators, modeling the servoing aspect of an action only as an uncertain completion time.

The approach proposed in this paper, exploiting environment dynamics, extends the traditional planning viewpoint principally in that it enriches an agent's action repertoire by inspecting the (procedural) model of the environment and looking for useful inherent patterns that an agent can capitalize on. Chapman [6] calls these patterns *dynamics,* and that usage here is also adopted here. Dynamics are structural or temporal regularities of the environment. They exist independent of the agent and are essentially acausal. A simple example from the kitting robot domain is of a conveyor belt carrying parts past the robot. A part on the belt will start by being drawn inexorably towards the robot; after a certain time, it will be within the robots workspace; and, if it remains on the belt, a certain time after that it will be irrecoverably gone.

Some dynamics admit participation by the agent. That is, the actions available to the agent can be affected by and/or can affect the course of the dynamic. This is a much weaker concept that the notion of an event being within or partially within an agent's control [24]. For example, the motion of the belt may indeed be beyond the control of an agent. Nonetheless, an agent can participate in this dynamic to acquire an object. To participate in a dynamic, an agent must be capable of knowing about, or sensing components of, the dynamic and of producing actions that affect, or are affected by, the dynamic. For example to exploit the belt dynamic to acquire an object, the kitting robot needs to be able to tell when the part is in its workspace and it needs to be able to take a part from the belt. We say that an agent *participates* in a dynamic to achieve an objective when it uses a reaction triggered off the dynamic to carry out some action that in conjunction with the dynamic achieves the objective. Note that this allows us to utilize actions whose effects are context dependent, since the dynamic resolves the context. As will be shown later on, the generated reactions capture 'closed loop' control law behavior, as opposed to plans generated in traditional planning approaches, which are essentially 'open loop' strategies.

3.3 Example: The Tracking Reaction

To explain the concept of exploiting dynamics, consider the simple tracking environment shown in figure 2. This environment contains a number of objects. The objects may be moving, and that motion is beyond the control of the agent. The environment also contains

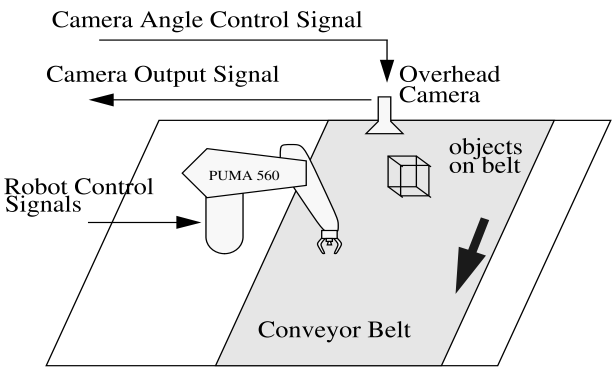

Figure 2: The Tracking Environment.

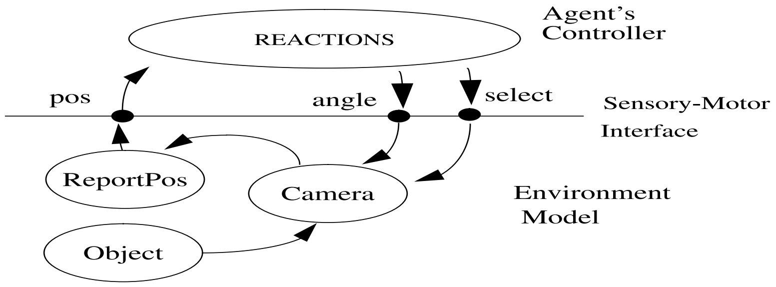

Figure 3: Modeling the Tracking Environment

a camera, the orientation of which can be controlled. If an object is in view, the camera will report the position of that object. We will assume that the agent has an object of type T in view initially, and that its objective is to know the position of this (possibly moving) object at all times.

The procedural model of this tracking environment `TEM` and the agent's reactor are shown in figure 3. (The formal representation for these will be described in a later section.) The central line denotes the sensory-motor interface of the agent. Above the line are the agents reactions. Below the line is the model of the environment. Reactions and environment model interact only through the sensory-motor interface. In the reactor, of course, these reactions will affect, and be affected by, the actual environment the agent is in. The entire contents of figure 3 would be contained "inside" the planner and constitute its model of the environment and of the reactor.

The agent has access to the environment through its sensory-motor interface. This interface is a set of communication channels, ports, through which the agent can either affect the environment by transmitting a control value, e.g., a setpoint for a robot, or obtain information by receiving a value, e.g., the position of an object, given that the camera process is prodded to look for objects. For the agent to receive sensory information, it has to explicitly perform a sensing action, which is modeled as 'hooking' into one or more of its sensory interface ports. This contrasts with systems such as ERE [4] and RAPS [8], where a hidden sensory subsystem continually updates a database with knowledge about the world, which the agent can access for free. This latter solution is only viable if the knowledge needs of an agent can be satisfied by sensing that is performed independent of the agent's course of action. At best this it is inefficient. In the realistic case of limited sensing resources, it can lead to an agent waiting for crucial information, and possibly missing deadlines, while the sensing system is updating other pieces of knowledge (knowledge perhaps irrelevant to the agent's current needs).

The tracking environment model consists of a process representing the agent's visual sensor `Camera`, and some number of object processes, `Object`. The `Camera` process accepts communication on a port, *angle*, which selects the pose of the camera and *select*, which tells the camera the type of object the agent is interested in. If an object of the selected type is in view at the angle the camera is currently set to, then the position of that object is returned on the port *pos*. `Camera` determines what objects are in view as follows. It has communication port connections to all existing `Object` processes (only one such process is shown in figure 3). These processes represent objects in the environment. Each `Object` processes continually reports its position and type to `Camera`, which applies a function `InView` based on the *select* and *angle* values to determine if a selected object is in view. The position of any object in view is echoed back on *pos*.

The agent can acquire position information via its sensory interface port *pos*. To find out what kind of information this port contains, it's necessary to trace back the connections from this port through the processes in the environment, and possibly back to the motor interface ports (in this case *angle* and *select*). As we trace back, we can annotate the connections with the conditions under which they actually transmit information. (The formalization of this analysis in our process model will be explained presently). For now, say that the connection from the `Camera` to the *pos* port is only transmitting whenever an object of type T is in the field of view of the camera, as determined by the condition $\texttt{InView}_{angle,pos}$.

Thus, in order to keep receiving information on port *pos* the agent has to keep this condition true. In other words, we have extracted a condition that should be the loop invariant for the tracking reaction. An example of a tracking reaction that maintains this invariant is as follows: The agent continually reads in the position of an object on port *pos*, transforms this position to a camera angle and transmits that angle to the motor interface port *angle*.

4 Representing and Analyzing Environments

To put the approach on a more formal footing, the following problems need to be addressed:

1. A concise language for specifying environment models needs to be developed (the "input").
2. A concise language for specifying action schemas needs to be developed (the "output").
3. A method for identifying dynamics, based on the environment model language, needs to be designed.
4. A method for translating identified dynamics into action schemas needs to be designed.

4.1 A Concise Language for Representing Environment Models

$\mathcal{RS}$ is a formal model of computation targeted at sensory-based robotics [18, 16]. Plans, actions and world models are all represented in $\mathcal{RS}$ as hierarchical networks of concurrent processes. Techniques for analyzing $\mathcal{RS}$ process networks to capture the interaction between plans and environments have been developed [17, 15, 19]. Process networks allow us to more easily capture the recursion or interaction that are central to dynamics. However, $\mathcal{RS}$ doesn't help us to mechanize the automatic generation of reactions. For that we make use of an Interval Temporal Logic (ITL) based on Pelavin's model [24]. ITL on its own has difficulty capturing recursion. We will show that the mix of these two formalisms allows us to identify useful dynamics and then select reactions that achieve the objectives.

4.1.1 Processes in $\mathcal{RS}$

A description of a process, or network of processes, is called a *schema*. For example, $\texttt{Joint}_v\langle x\rangle$ denotes a process that is an instance of the schema `Joint` with one ingoing parameter v and one outcoming result x. A process can terminate either successfully (stop) or unsuccessfully (aborts). Networks are built by composing processes together using several kinds of *process composition operators*. These allow processes to be ordered in various ways, including concurrent, conditional and iterative orderings. At the bottom of this hierarchy, every network must be composed from a set of atomic, pre-defined processes. The set of *basic schemas* defines what processes are atomic.

$\mathcal{RS}$ notation has a *process algebra* [12] style: processes are specified as algebraic compositions of other processes. The relationship between this and state-machine models of computation is well understood. A process is essentially a state-machine; for $\mathcal{RS}$ a process is a special kind of automaton called a *port automaton*. A process composition operation is an instruction on how to combine automata: more specifically, how to combine state-transition graphs. Thus, $\mathcal{RS}$ provides a way to build processes without ever explicitly mentioning states. This allows us to avoid some of the problems of state-based approaches to representing and reasoning about actions [24].

4.1.2 Composing Processes

We will make use of four atomic composition operations, namely, sequential, concurrent, conditional and disabling, and two non-atomic composition operations, namely, synchronous recurrent and asynchronous recurrent.

Sequential composition is used to enforce a strict ordering on operations. For example, ensuring that $part2$ is always placed after $part1$, $\mathtt{T} = \mathtt{Place}_{part1};\mathtt{Place}_{part2}$.

Concurrent composition indicates that two or more processes should be carried out concurrently. This allows us to represent a lack of ordering between activities, e.g., $\mathtt{T} = \mathtt{Place}_{part1} \mid \mathtt{Place}_{part2}$, or actions which need to be done simultaneously, e.g., squeezing an object obj with two fingers $f1$ and $f2$, $\mathtt{T} = \mathtt{ApplyForce}_{f1,obj} \mid \mathtt{ApplyForce}_{f2,obj}$. The process $\mathtt{T}$ terminates once all its arguments have terminated.

Concurrent processes can communicate with each other via *communication ports*. Two special basic processes are defined for this purpose: $\mathtt{IN}_p\langle v\rangle$ reads a value from port p and can pass it on through its result v; $\mathtt{OUT}_{p,v}$ writes the value v out on port p. All port communication is synchronous[2] and there can be fan-in and fan-out on ports. For example, if $\mathtt{X}_p = \mathtt{IN}_p$ and $\mathtt{Y}_q = \mathtt{OUT}_{q,n}$ then $\mathtt{N} = \mathtt{X}_a \mid \mathtt{Y}_a$ is a network of an input process in parallel with, and connected to, an output process[3].

Conditional composition allows the construction of networks whose behavior is conditional. The network of $\mathtt{T} = \mathtt{P:Q}$ behaves like $\mathtt{P;Q}$ iff $\mathtt{P}$ terminates successfully. If $\mathtt{P}$ aborts, then $\mathtt{Q}$ is not carried out, and $\mathtt{T}$ aborts. For example, in $\mathtt{LidOpen}_{box} : \mathtt{Place}_{x,box}$ whether $\mathtt{Place}$ is carried out or not depends on whether $\mathtt{LidOpen}$ terminates successfully or not. Conditional composition can be used to pass results between processes in a 'pipelined' fashion. If $\mathtt{X}_p = \mathtt{IN}_p\langle v\rangle : \mathtt{A}_v$ the value read in on the port p is used as a parameter to the process $\mathtt{A}$.

---

[2] Asynchronous communication can be built from dynamic process creation plus synchronous communication.

[3] The concept of a *port variable*, a variable referring to a port, is being used here in place of the port connection relation used in previous $\mathcal{RS}$ work, e.g., [18].

Disabling composition allows one process to terminate another. The network $\texttt{T} = \texttt{A\#B}$ behaves like $\texttt{A|B}$ except that it terminates whenever *any* of its arguments terminates.

Finally, two useful non-atomic composition operations are defined in terms of these four: *synchronous recurrent* $\texttt{A} :; \texttt{B} = \texttt{A} : (\texttt{B}; (\texttt{A} :; \texttt{B}))$, and *asynchronous recurrent* $\texttt{A} :: \texttt{B} = \texttt{A} : (\texttt{B} \mid (\texttt{A} :: \texttt{B}))$. Synchronous recurrent composition is similar to *while-loop* iteration. Asynchronous recurrent composition does not iterate, but rather 'spawns' off a set of concurrent processes every time its 'condition' is satisfied. These special forms of guarded recursion are useful (and can be implemented efficiently).

One technique for analyzing process networks is based on the algebraic properties of these compositions operators. These properties can be used to develop a set of 'process identities' that can be used to rewrite networks into alternate forms (e.g., simplify networks). One such technique that we will make reference to in the next section is to rewrite a network so that all 'local' processing is hidden and treated simply as a time delay. For example, in the network $\texttt{T} = \texttt{IN}_p\langle u \rangle : \texttt{A}_u\langle v \rangle : \texttt{OUT}_{q,v}$ if the process $\texttt{A}$ does not use the ports p and q, then from the perspective of communication on these ports, $\texttt{A}$ is local. The network can be written as $\texttt{T} = \texttt{IN}_p\langle u \rangle : \texttt{Delay}_t : \texttt{OUT}_{p,f(u)}$, for some time t and mapping f.

The network $\texttt{STREAM}_{y,v} = \texttt{OUT}_{y,v} :; \texttt{STOP}$ will reappear often in our examples. This network offers to transmit the value v on the port y infinitely often.

4.2 A Concise Language for Representing Action Schemas

We have already developed a planner based on an interval temporal logic (ITL) similar to that described by Pelavin [24]. Thus, the output of our proposed dynamics identification system should be compatible with the action schema input language for this planner.

Formulae in our ITL language are first order logic predicates augmented with a modal operator @, e.g., $InView(P)@i1$ specifies that the predicate $InView(P)$ should be evaluated over interval $i1$. In addition the special predicate $ITL($ $interval1$, $itlrelation$, $interval2)$ is used to specify constraints between intervals, e.g., $ITL(i1, [e], i2)$ specifies that the interval $i1$ should denote the same interval of time as $i2$, with e denoting equal. Other relevant time relation are s for starts, d for during and f for finishes. The i postfix denotes the inverse and including more than one relation denotes disjunction of the relations (see [2] for more details).

Processes are described by action schemas which specify executability conditions and effects (executability conditions are generalized 'pre'-conditions which may specify constraints that must hold *while* the the process is executing). For instance, below the specifications are given for the processes $LOOK(outport)$ and $MOVECAMERA(inport)$. The process $LOOK$ has one output port *outport*, which will hold the position information of the object, if it is in view.

```
atp_actionschema LOOK
output:  outport
intervals i1,i2;
         executability:  InView(obj)          @ i2 and
```

```
                           ITL(i2, [si e di fi],LOOK);
          effects:        ObjectPosition(outport) @ LOOK
endactionschema;
```

(We will assume that every action schema has an associated interval of the same name that refers to the interval of execution of the process). The process $MOVECAMERA$ is similarly defined but has an input port *inport*, which needs position information about the object, and will keep that object in view as long as that port has the up-to-date position of the object.

```
atp_actionschema MOVECAMERA
input:  inport
intervals i1,i2;
          executability:  ObjectPosition(inport) @ i1 and
                           ITL(i1, [si e di fi], MOVECAMERA);
          effects:        InView(obj) @ MOVECAMERA
endactionschema;
```

Note that the ports *inport* and *output* "carry" the (possibly changing) position information of the object. In addition to predicates denoting static information and process models, the planner can accept rules specifying consistency conditions, e.g., that only one process is allowed to transmit information to a port.

Reactions are straightforward to construct when the component actions are represented as producing or consuming port values over an interval in this continuous way. Consider constructing a $TRACKING$ reaction that moves the camera to maintain an object in view. The output port of $LOOK$ always has the current position of the object if that object is in view of the camera. If we place the position of an object on the input port of $MOVECAMERA$ then the camera will be moved so as to place that object in view. Carrying out these actions concurrently, with the ports connected, allows us to ensure that the current position of the camera is always set to the current position of the object and thus ensure that the object is always kept in view.

4.3 A Method to Identify Dynamics

4.3.1 Process Evolution

To analyze how process networks (world models and/or plans) execute over time, it is important to be able to derive how networks evolve as component processes dynamically terminate. To this end, we introduce the *evolves* operator. We say that process `P` evolves into process `Q` under condition Ω if `P` possibly becomes equal to `Q` when condition Ω occurs; we write this as $\mathtt{P} \xrightarrow{\Omega} \mathtt{Q}$. If there is a set of processes that `P` can possibly become equal to on condition c, then `P` necessarily becomes equal to exactly one of these when c occurs.

To define the evolves operator, we need two pieces of information: First, we need to know for each basic process under what conditions it terminates. Second, we need to know for each composition operator, how the composition is affected by the termination conditions of each argument process.

The conditions under which the basic process P terminates successfully will be written ΩP. In some cases, we will also need to use the conditions under which P terminates unsuccessfully, i.e., *aborts* itself; these necessary (but not sufficient[4]) conditions we write as $\mho P$.

We start by defining *single-step* evolution; this captures the concept of the very next single process creation/termination change that a network can undergo. Consider a sequential composition of processes T = P1;P2 , where P1 and P2 are basic. This network can only change as follows

$$\mathtt{P1;P2} \xrightarrow{\Omega\mathtt{P1}\vee\mho\mathtt{P1}} \mathtt{P2}$$

I.e., P1;P2 always evolves into P2 whether P1 terminates successfully or aborts. In this fashion, we can define the way in which evolves interacts with all the composition operators. Appendix B contains the full description of this interaction.

A specific network P might be capable of more than one possible next evolution. We define the set of next possible evolutions as

$$poss(\mathtt{P}) = \{(c, \mathtt{Q}) \text{ such that } \mathtt{P} \xrightarrow{c} \mathtt{Q}\} \tag{1}$$

In general we will be interested in the ultimate effect of a single chain of successive one-step evolutions. Thus, we define the evolves operator as the transitive closure of a chain of one-step evolutions:

$$\mathtt{P} \xrightarrow{c1} \mathtt{P1} \xrightarrow{c2} \mathtt{P2} \xrightarrow{c3} \ldots \xrightarrow{cn} \mathtt{Q} \iff \mathtt{P} \stackrel{c}{\longrightarrow} \mathtt{Q} \qquad \text{where } c = c1.c2.c3\ldots.cn. \tag{2}$$

We write temporal ordering of conditions using a period, e.g., $A.B$ is read A then B.

The set of all possible evolutions is the transitive closure of $poss(\mathtt{P})$ over the one-step evolves operator (analogous to the concept of 'reachability' and the 'reachable set' in linear systems theory). We can define this recursively in terms of one-step evolution as 'every network that can be reached in one evolution plus everything that can be reached from there', or, equivalently, in terms of the transitive evolves operator as:

$$poss^*(\mathtt{P}) = \{(c, \mathtt{Q}) \text{ such that } \mathtt{P} \stackrel{c}{\longrightarrow} \mathtt{Q}\} \tag{3}$$

4.3.2 Process Limits

To pull out dynamics from an environment model, we need to understand how that environment behaves in the following sense. We need to understand how sensory input reflects change in the model, and ultimately how that change is related to motor output (if it is). That is, we need to construct something equivalent to the usual transfer function of linear

[4]A process can also be forced to abort in disabling composition '#'.

systems theory: the result of transitively closing all the internal evolution steps between the motor and sensory interfaces. For a set of reactions, this gives us a sensory input to motor output mapping; for an environment model, it gives us a motor output to sensory input mapping. The concept of process limits and bound processes is introduced below as an approach to pulling out dynamics from an environment model expressed in $\mathcal{RS}$.

A *process limit* is a process that has evolved from the initial process, but cannot evolve any further. The significance of the limit is that it prevents any further change of behavior by the process. Any process may have zero or more limits; we call the set of limits of a process `P`, $limitset(\mathtt{P})$, and it is a subset of $poss^*(\mathtt{P})$:

$$
\begin{array}{ll}
(c, \mathtt{Q}) \in limitset(\mathtt{P}) & \\
 & i) \quad \mathtt{P} \xrightarrow{c} \mathtt{Q},\ and \\
 & ii) \quad there\ exists\ no\ (cr, \mathtt{R}) \quad s.t.\ \mathtt{Q} \xrightarrow{cr} \mathtt{R}
\end{array}
\tag{4}
$$

The evolves operator provides us a way to determine process limits. In the case that a process does have limits, then we can use evolves to 'simulate' the execution of the network until we reach a limit. To get *all* the limits of a network we need to produce $poss^*$ for the network. Therein lies a problem: If the process is in an endless loop, however, then the 'simulated' execution will also be in a loop! This can be addressed by stopping the production of $poss^*$ whenever a process recursion is detected. In the remainder of the paper, $limitset(\mathtt{P})$ will refer to this augmented concept: all the process limits *plus* any recursive process invocation by `P`.

A process, such as `Q` in the above definition, that does not evolve to anything, we call a *bound*, since it may bound the evolution of any process it occurs in. Any basic process that never terminates is a bound. The basic processes `STOP` and `ABORT` are also bounds. Strictly speaking, whether or not a process is a bound is just a question of its definition. However, we will also find it convenient to also designate specific processes as bounds for the purpose and duration of some analysis. Such processes will be referred to as pseudo-bounds.

Let `EM` be an $\mathcal{RS}$ model of the environment, i.e., a process network. First, we treat any sensory processes in `EM` as pseudo-bounds. That is, whenever one of those process is produced in the evolution of `EM`, then the evolution will go no further, and this bound and the conditions under which it was evolved, will become part of the limitset. This will tell us the ultimate sensory process(es) produced due to any motor input or any environment changes. This will also tell us about internal cycles in the environment, e.g., the motion of an autonomous object, since recursive process invocations are now also contained in the limitset.

We will look at a number of examples of limitset generation in Section 5 and our implementation of limitset generation is discussed in section 6.

4.4 Translating Dynamics into Action Schemas

In section 4.2 we discussed the actions $LOOK$ and $MOVECAMERA$ and noted how easily they could be reasoned given the assumption that their associated ports continously carried

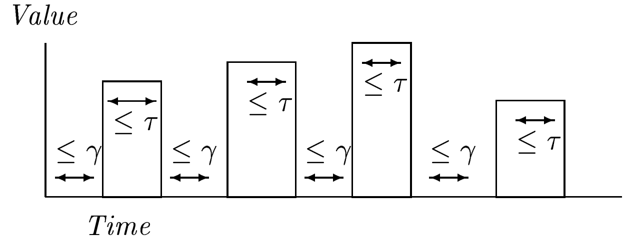

Figure 4: Timing Characteristics

data values. Recall that our objective is to build action schemas such as these two by combining the basic sensory and motor capabilities of the agent with dynamics in the environment. The tool introduced to extract dynamics is the $\mathcal{RS}$ limitset. The limitset is a discrete concept and will tell us about the sequence processes and/or of values on a port. Before we can reason in the aforementioned fashion with ITL formalizations like *LOOK MOVECAMERA* we need to be able to say when discrete port communication can approximate a continuous signal.

The intuition here is that if a value is produced sufficiently often, then it is a reasonable approximation to continuous. So, the first criterion is that the network producing the value must be in a 'loop', continually producing new values. The second criterion is that new values must be produced 'often enough.' We will begin by introducing some canonical network forms that produce outputs in a continual sequence with some associated time delays. These definitions will formalize the first criterion. We will then place a constraint on the timing delays to formalize the second criterion.

4.4.1 Characterizing denseness

Definition 1:
For any network that can be reduced to the form

$$\mathtt{T}_v = \mathtt{Term}_\gamma \ ; \ (\mathtt{STREAM}_{x,v} \# \mathtt{Term}_\tau) \ ; \ \mathtt{T}_v$$

for small τ and γ then we say x is (γ, τ)-dense.

This definition establishes when a signal source (there is no input in this network) is considered (γ, τ)-dense. By 'dense' we mean that the network continually produces signal values. The γ and τ are the timing characteristics for the network. It is realistic to expect that the transit from input to output will incur some small, bounded delay. This delay is modeled by the `Term` process, which we define as follows: `Term` waits for an arbitrary time duration between 0 and its argument, and then terminates. Once a new value is calculated, it will be 'locked in' for a time so that other networks can read it. `STREAM` never terminates on its own; it continually offers output on its port (see its definition in section 4.1.2). The

$\texttt{Term}_\tau$ process will terminate, however, after at most a time τ. Thus, through the disabling composition, '#', it terminates `STREAM`. This produces a discrete signal as shown in figure 4. (This is *not* a periodic signal in the standard sense [30]. It is, however, bounded by a periodic square pulse train of period $\gamma + \tau$ and duty ratio $\frac{\tau}{\tau+\gamma}$.)

We also need to consider the case when a value is transformed by a network. So given a $(\gamma i, \tau i)$-dense value on an input port x, for what kinds of network that transform that value onto a value on the output port y do we consider the value on y to be dense? This leads to our next two definitions:

Definition 2:
For any network that can be reduced to the form

$$\texttt{T} = \texttt{IN}_x\langle v\rangle : (\texttt{Term}_{\gamma o}; (\texttt{STREAM}_{y,f(v)}\#\texttt{Term}_{\tau o}))\ ;\ \texttt{T}$$

for small values of τo and γo and a continuous function f then y is locally $(\gamma o, \tau o)$-dense with x.

A process being locally dense on a port means that the internal transfer of information from input to output conforms to our intuition of denseness.

.

Definition 3:
If an input port x is $(\gamma i, \tau i)$-dense and an output port y is locally $(\gamma o, \tau o)$-dense with x then y is $(\gamma i + \gamma o, MAX(\tau i, \tau o))$-dense.

This says that in composing dense signals, the signal delays add in the resultant signal, and the resolution factor of the result is the same as larger of the two resolutions.

These (γ, τ)-denseness definitions can now be applied to a model of the environment to find the characteristics associated with environment dynamics. They are applied as indicated at the end of section 4.1.2: rewriting networks so that all 'local' processing, processing that doesn't refer to the output and/or input ports in question, is hidden and treated simply as a time delay (modeled for example by `Term`). We can use the denseness definitions in two ways. Firstly, we can use them to filter out processes that cannot be utilized in our ITL reaction generation approach. This is essentially a qualitative approach, since we don't need to know the values of τ and γ, just that the process is of the right form. But we can also use these definitions in a quantitative way, to ensure a stable mating between the timing characteristics of reactions and the environment in which they operate.

4.4.2 Denseness constraints for reactions

We say a reaction is *sufficiently dense* with respect to a port in the model of the environment if that reaction generates or consumes data more quickly than the environment does.

Let us assume that that the environment model provides data, for example the position of an object, on a $(\gamma e, \tau e)$-dense port, p. Let us assume that values on p are consumed by a reaction that is $(\gamma r, \tau r)$-dense. For the reaction to be sufficiently dense with respect to this environment, we demand that the bounding period of the reaction be less than that of the environment, i.e., that $\tau r + \gamma r < \tau e + \gamma e$.

In the case that the reaction is writing to p, rather than reading from it, then we additionally constrain the reaction to enforce $\gamma r < \gamma e$. This ensures that from the point of view of the environment, the reaction is always ready to produce new data.

Under what time constraints can this combination of environment and reaction be considered stable? We will consider the system stable if the reaction is sufficiently dense, i.e., able to generate responses at a faster rate than the environment can change. Thus, if it can follow the amplitude of the changes in the environment, the reaction will indeed provide a stable behavior, by undoing the (assumed small) deviation in a single correction step each cycle.

If, on the other hand, the controls become saturated or the amplitude of the changes are large enough to lose sensor coverage in a single environment change, the interaction pattern gets broken. Saturation is a difficult subject in control theory in general, and, in our work, the implicit assumption is made that this doesn't occur.

If these denseness constraints are met, we can abstract from the interactive generation or consumption of data. Thus, the ITL description just needs to mention that the port 'carries' the information. For the examples in this paper, we will employ denseness to filter out dynamics that cannot be used in our ITL reasoner. We will assume, however, that the timing characteristics of the reactions do ensure that they are sufficiently dense.

5 Detailed Examples of Exploiting Dynamics

Sufficient structure has now been established to apply the proposed approach to some detailed examples. The following examples are from the kitting robot domain. The first is the tracking example introduced back in section 3.3. Subsequent examples build on this one. The guarded motion example introduces motions of the robot to acquire an object. The belt example introduces environmental motions that can be exploited to acquire an object. In each case, the environment will be specified as a set of $\mathcal{RS}$ network definitions, the limitset concept used to find useful dynamics in this environment, and action schemas generated by combining these dynamics with the sensory-motor capabilities of the agent.

5.1 The Tracking Example

The $\mathcal{RS}$ network to implement the environment model for the tracking example discussed previously is presented below. The model is comprised of a set of mutually recursive process equations. The topmost process is the `TEM` (for tracking environment model) network. Its parameters are the initial position of the object to be tracked p, and the camera angle a. For simplicity, the *select* port has been omitted from the camera model; this allows us to focus more directly on the reactive portion of the problem. We will assume for now that the camera only recognizes objects of the desired type.

$$\begin{array}{lcl}\texttt{TEM}_{p,a} & = & (\texttt{Object}_p \mid \texttt{Camera}_a) \\ \texttt{Object}_p & = & (\texttt{STREAM}_{op,p}\#\texttt{Delay}_{t0}) \ ; \ \texttt{Ran}_{-\epsilon,\epsilon}\langle\gamma\rangle \ : \ \texttt{Object}_{p+\gamma} \\ \texttt{Camera}_a & = & (\texttt{Monitor}_a\#\texttt{IN}_{angle}\langle na\rangle) \ : \ \texttt{Camera}_{na} \\ \texttt{Monitor}_a & = & (\texttt{Delay}_{ts0};\texttt{IN}_{op}\langle p\rangle;\texttt{Delay}_{ts1}) \ :: \ (\texttt{InView}_{p,a} : \texttt{ReportPos}_p) \\ \texttt{ReportPos}_p & = & \texttt{STREAM}_{posn,p}\#\texttt{Delay}_{ts0}\end{array} \tag{5}$$

The basic process $\texttt{Ran}_{l,h}$ is defined as producing an arbitrary result value between l and h. `Ran` is being used here to model the uncertainty in object motion. Thus, $\texttt{Object}_p$ continually repeats the following behavior: it offers its current position p for a time $t0$ on the port op, and then 'moves' (updates its position) some arbitrary amount between $+\epsilon$ and $-\epsilon$ relative to p. $\texttt{Camera}_a$ is continually checking (via `Monitor`) for any object in view at the current position, while concurrently being ready to accept a new position and repeat the behavior for that position. `Monitor` polls the position data from `Object` processes and spawns off a subnetwork to determine if the object is in view and passes on the position data if it is. The process `InView` carries out the calculations to determine if the object is in view. If it is, then the `ReportPos` process continually offers the position of the object on port pos. If the object is not in view, the subnetwork just aborts. The values of $t0, ts0$ and $ts1$ are constant and are timing characteristics of the model. (The use of $ts0$ and $ts1$ in `Monitor` and `ReportPos` are necessary for denseness reasons explained later.)

The `TEM` model describes the environment itself. However, the agent interacts with this environment through its sensors and actuators. These sensors and actuators may impose limitations on this interaction, so we need to model them also. We will assume the sensory-motor interface is defined by the two processes `sense1` and `motor1` defined as follows:

$$\begin{array}{lcl}\texttt{sense1} & = & (\texttt{IN}_{pos}\langle v\rangle;\texttt{Delay}_{gl}) ::; (\texttt{STREAM}_{s1,v}\#\texttt{Delay}_{tl}) \\ \texttt{motor1} & = & (\texttt{IN}_{m1}\langle v\rangle;\texttt{Delay}_{gm}) ::; (\texttt{STREAM}_{angle,v}\#\texttt{Delay}_{tm})\end{array} \tag{6}$$

These primitive 'actions' affect or are affected by parts of the environment model, i.e., `sense1` is affected by pos and `motor1` affects $angle$. There is an important issue of style here that needs to be emphasized: the agent does not invoke whatever effect `motor1` has on the environment by creating an instance of `motor1` . The action is invoked by writing a message on `motor1` 's input port $m1$. A single instance of each of `sense1` and `motor1` is always present and these constitute the sensory-motor interface. Each is in a loop, continually reading a value and passing that value to/from the environment.

The timing delays in these actions, gl, tl, gm, and tm, model the timing delays in real sensor and effector hardware. These actions have been given neutral names, rather than names such as 'movecamera' and so on, to reinforce the fact that they do not constitute actions usable by the ITL planner. Rather the *combination* of one of these actions plus some dynamic extracted from the environment is what is presented to the ITL planner as an action.

The environment model and sensory-motor interface have now been described. Any *specific* environment and agent will be characterized by an instance of `TEM` with its initial parameters and instances of `sense1` and `motor1`, e.g.,

$$\texttt{ThisWorld} \quad = \quad \texttt{TEM}_{pinit,ainit} \mid \texttt{sense1} \mid \texttt{motor1} \tag{7}$$

where *pinit* and *ainit* are constants.

Notice that all these networks produce values on ports in a *discrete* fashion. For example, the port *pos* does not present a continuously changing value, rather it holds a value that changes in discrete time increments of $ts0$. The value $ts0$ models the rate at which object position information changes in the environment. In contrast, the timing delays in `sense1` and `motor1` model the rate at which *the agent's* sensors and actuators interact with the environment.

The next step is to proceed with extracting dynamics from the environment model. As discussed in section 4.3.1, this is done by looking at the limitset of the environment model network assuming that the sensory input processes (and any recursive calls) are pseudo-bounds. This produces the following set (the mechanization of this operation is described in section 6):

$$\begin{array}{lll} limitset(\texttt{TEM}_{p,a}) & = \{ & (\gamma \in [-\epsilon, \epsilon] \quad , \quad \texttt{TEM}_{p+\gamma,a}), \\ & & (\Omega\texttt{IN}_{angle}\langle na\rangle \quad , \quad \texttt{TEM}_{p,na}), \\ & & (inview(p) \quad , \quad \texttt{TEM}_{p,a} \mid \texttt{ReportPos}_p) \\ & \} & \end{array} \tag{8}$$

The first element of the limitset tells us that left on its own (no conditions) the object process may "move" (i.e., the location value that it transmits may change). The second element tells us that changing the value on the *angle* port changes the camera position. Let the termination condition of the process `InView` be written as $inview(p)$. The final element tells us that if the predicate $inview(p)$ holds then the port *pos* holds the value of the object position. We are now in a position to combine the information on the dynamics in the environment with the basic sensory and motor definitions, based on what will provide dense information, and generate a set of high-level abstractions for ITL.

Based on definitions 2 and 3 and the assumption that *tm* and *gm* are small enough, we can say that if the value on *angle* in `motor1` is dense, then so is the position of the camera. Therefore we can add an ITL actionschema $MOVECAMERA(a)$ with the formalization as given earlier in section 4.2. When $MOVECAMERA(a)$ is applied on an interval I then the position of the camera in that interval is always the same as the value set in a.

Now considering *pos*, we can say that if $inview(p)$ holds, then *pos* reflects the position of the object. But is it (γ, τ)-dense? `TEM` employs an asynchronous recurrent composition (the '::') to spawn off a `ReportPos` process to report on the position of the object. Our denseness definitions were based on a simpler recursive form. (We have to use the asynchronous composition for `ReportPos` to appear as part of the limitset.) As it happens, the denseness of the asynchronous form can be related to Definition 2 *as long as* the signal is periodic and we 'sandwich' the input process between two delay processes as shown in `Monitor`. This

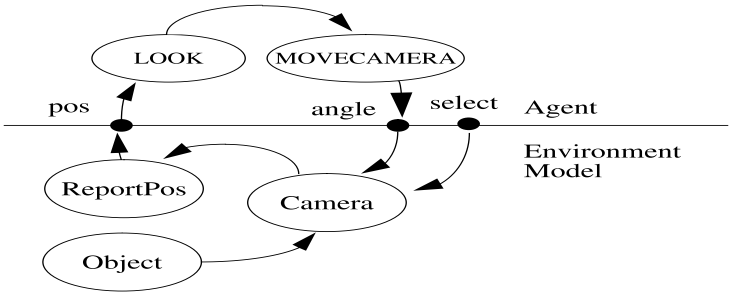

Figure 5: The Tracking Reaction

result is demonstrated in Appendix A. A more general result can be obtained at the expense of a slightly more complex environment model.

Thus, by Appendix A, *pos* is dense if the position reported by the object is dense and if the object is in view. Definition 1 establishes that the position reported by the object is dense. Thus, the abstract ITL action $LOOK(p)$, defined as in section 4.2, can be constructed: If $LOOK(p)$ holds on an interval I then if the object is in view, then the value of p is the value of the position of the object.

From these candidate actions, it is straightforward for the ITL planner to construct the TRACK reflex to maintain an object in view. This consists of running the LOOK and MOVECAMERA actions concurrently with their ports connected. Translating this ITL specification back into $\mathcal{RS}$ produces the network:

$$\texttt{TRACK} \quad = \quad (\texttt{LOOK}_p \mid \texttt{MOVECAMERA}_p) \tag{9}$$

The mechanization of this example is reported in section 6.

5.2 The Guarded Motion Environment

The guarded move environment model `GMEM` is shown in figure 6. This is similar to the tracking environment except that we now additionally represent the robot itself and a tactile sensor mounted on the robot's gripper. There is an additional control port that allows us to *incrementally* move the robot, and an additional sensor port that reports back when an object has been contacted.

The objective in this example is for the robot to acquire the object. To do this, it needs to reach out by incrementally updating its position in the direction of the object, and when the tactile sensor indicates that contact has occurred, to grasp the object.

The $\mathcal{RS}$ network to implement the environment for the guarded move environment is the

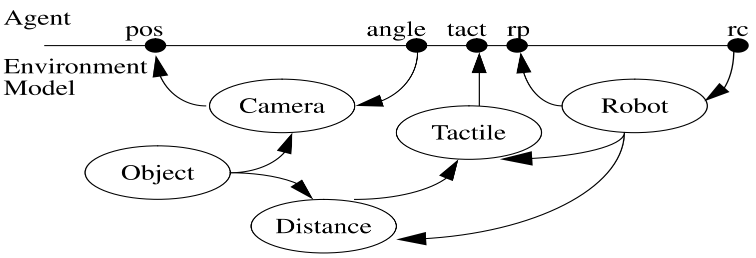

Figure 6: The Guarded Motion Environment GMEM

set of process equations for the tracking environment *plus* the equations given below:

$$\begin{array}{lcl} \mathtt{GEM}_{p,a,rp} & = & (\mathtt{TEM}_{p,a} \mid \mathtt{Robot}_{rp} \mid \mathtt{Tactile} \mid \mathtt{Dist}) \\ \mathtt{Robot}_{p} & = & (\mathtt{IN}_{rc}\langle\delta\rangle \ \# \ \mathtt{STREAM}_{rp,p}) \ ; \ \mathtt{Robot}_{p+\delta} \\ \mathtt{Dist} & = & (\mathtt{IN}_{op}\langle p1\rangle \mid \mathtt{IN}_{rp}\langle p2\rangle) \ :; \mathtt{OUT}_{d,p1-p2} \\ \mathtt{Tactile} & = & (\mathtt{Delay}_{tt0}; \mathtt{IN}_{d}\langle v\rangle; \mathtt{Delay}_{tt1}) \ :: \ (\mathtt{Equal}_{v,0} : \mathtt{ReportTact}_{1}) \\ \mathtt{ReportTact}_{v} & = & \mathtt{STREAM}_{tact,v}\#\mathtt{Delay}_{tt0} \end{array} \quad (10)$$

The highest level process is GEM and its arguments are the initial object and camera positions and the initial position of the robot arm. $tt0$ and $tt1$ are constant values, timing characteristics of the model.

The additional basic sensory-motor interface processes we need are now:

$$\begin{array}{lcl} \mathtt{touch} & = & (\mathtt{IN}_{tact}\langle v\rangle; \mathtt{Delay}_{gto}) \ :; \ (\mathtt{STREAM}_{t,v}\#\mathtt{Delay}_{tto}) \\ \mathtt{move} & = & (\mathtt{IN}_{r}\langle d\rangle; \mathtt{Delay}_{gmv}) \ :; \ (\mathtt{STREAM}_{rp,d}\#\mathtt{Delay}_{tmv}) \end{array} \quad (11)$$

We can build on the tracking reaction established in the previous section. The tracking reaction ensures that the object always remains in view. This in turns establish that $LOOK(p)$ always returns the object's position. To determine what other high-level actions we can construct, we need to again identify dynamics in the environment model.

This gives us the following set (duplicating partially the results from `TEM`):

$$
\begin{array}{llll}
limitset(\mathtt{GEM}_{p,a,pr}) & = \{ & (\gamma \in [-\epsilon, \epsilon] & , \ \mathtt{GEM}_{p+\gamma,a,pr}), \\
& & (\Omega\mathtt{IN}_{angle}\langle na\rangle & , \ \mathtt{GEM}_{p,na,pr}), \\
& & (inview(p) & , \ \mathtt{GEM}_{p,a,pr} \mid \mathtt{ReportPos}_p), \\
& & (\Omega\mathtt{IN}_{rc}\langle\delta\rangle & , \ \mathtt{GEM}_{p,a,pr+\delta}), \\
& & (dist(p,pr) = 0 & , \ \mathtt{GEM}_{p,a,pr} \mid \mathtt{ReportTact}_1) \\
& \} & &
\end{array}
\tag{12}
$$

The first three entries parallel those of `TEM.` The fourth entry reports that sending a value of δ to the rc port will cause the robot to move δ from its current position. The final entry claims that when the distance between the robot and the object is zero, then the tactile sensor will signal a 1. This is sufficient to allow us to construct a high-level process $CONTACT$ whose behavior is to terminate whenever the tactile sensor signals a contact. However, our definitions of denseness are not sufficient to allow us to build a $MOVETOWARDS(r)$ action that we can connect with a concurrent $LOOK(p)$ to push us towards the object. We are missing the notion of approximating continuous increasing or decreasing values. This motivates the following definition:

Definition 4:
For any network that can be reduced to the form

$$\mathtt{T}_v = \mathtt{Term}_\gamma \ ; \ (\mathtt{STREAM}_{x,v}\#\mathtt{Term}_\tau) \ ; \ \mathtt{T}_{v+\delta}$$

for small values of τ and γ then x is

1. (γ, τ)-densely increasing if $\delta > 0$.
2. (γ, τ)-dense if $\delta = 0$.
3. (γ, τ)-densely decreasing if $\delta < 0$.

Definitions 2 and 3 now need to be extended so that they 'transfer' the densely increasing/decreasing properties also. If Definition 4 is used along with the others in our generator of high-level actions then we can suggest some interesting new actions. The third entry in the limitset combined with `move` tells us that for a positive value, r will be densely increasing on robot position, suggesting a high-level action $MOVEUP(r)$. For a negative value, r will be densely decreasing, suggesting a high-level action $MOVEDOWN(r)$. More interestingly, if r is $p - pr$, then it is densely decreasing with respect to $dist(p, pr)$, suggesting an action $MOVETOWARDS(r)$ and, if r is $pr - p$, then it is densely increasing with respect to $dist(p, pr)$, suggesting an action $MOVEAWAY(r)$.

We can now construct a useful reaction from these actions as follows. We make a concurrent combination of $LOOK(p)$ and $MOVETOWARDS(r)$ with p connected to r. We can

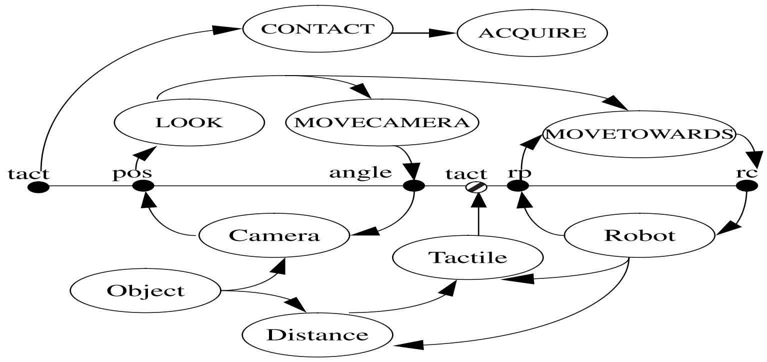

Figure 7: The Guarded Motion Reaction

tell from the descriptions of these actions that this will eventually produce $dist(p, pr) = 0$. We can recognize this fact with $CONTACT$ and begin the acquisition at that point. Let us assume an action $ACQUIRE$ to complete the acquisition. Translating this ITL specification back into $\mathcal{RS}$ we produce the reaction network:

$$\texttt{GuardedMove} \quad = \quad (\texttt{CONTACT}\#(\texttt{LOOK}_p \mid \texttt{MOVETOWARDS}_p)); \texttt{ACQUIRE} \tag{13}$$

5.3 The Belt Environment

The belt environment model `BEM` is shown in figure 8. This is very similar to the guarded move environment except that we replace the `Object` process with a `BeltObject` process. This process represents an object on the belt.

The objective in this example is the same as in the previous example: to acquire the object. The interesting variation here is that the belt motion will carry the object to the robot if it's "smart enough" to take advantage of it.

The $\mathcal{RS}$ network to implement the environment for the belt environment is the guarded move environment with `Object` replaced by `BeltObject` defined below:

$$\texttt{BeltObject}_p \quad = \quad (\texttt{STREAM}_{op,p}\#\texttt{Delay}_{tm0}) \; ; \; \texttt{LST}_{p,\nu} \; : \; \texttt{Object}_{p+\mu} \tag{14}$$

`BeltObject` is a version of `Object` in which the position of the object only increments in a fixed direction and amount μ in time $tm0$. Once the the position of the object hits the end of the belt, ν, the process recursion terminates, indicating that the object is now off the end of the belt and no more position signals are generated. The process $\texttt{BEM}_{p,a,pr}$ is the same as `GEM` but with `Object` replaced by `BeltObject.`

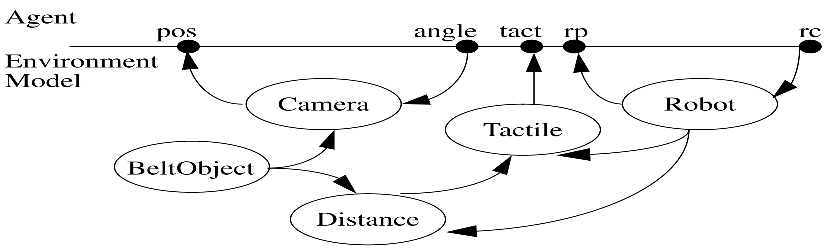

Figure 8: The Belt Environment `BEM`

Applying the limitset approach to identifying dynamics, we get the following set (duplicating partially the results from `GEM`):

$$
\begin{array}{rlll}
limitset(\mathtt{BEM}_{p,a,pr}) & = \{ & (p < \nu & , \mathtt{BEM}_{p+\mu,a,pr}), \\
 & & (\Omega\mathtt{IN}_{angle}\langle na \rangle & , \mathtt{BEM}_{p,na,pr}), \\
 & & (inview(p) & , \mathtt{BEM}_{p,a,pr} \mid \mathtt{ReportPos}_p), \\
 & & (\Omega\mathtt{IN}_{rc}\langle \delta \rangle & , \mathtt{BEM}_{p,a,pr+\delta}), \\
 & & (dist(p,pr) = 0 & , \mathtt{BEM}_{p,a,pr} \mid \mathtt{ReportTact}_1) \\
 & \} & &
\end{array} \tag{15}
$$

The first entry tells us that as long as the position of the object is less than ν, then the object will move incrementally in the direction μ. If we look at this entry with respect to Definition 3, then in addition to the actions we pulled out of `GEM`, we can suggest a high-level action that is densely decreasing with respect to $dist(p, pr)$ as long as $p < \nu$. This high-level action is $WAITFOROBJECT$. The reaction generated for the guarded move is equally valid in this environment. However, we can now also generate a reaction in which the robot waits for the part to arrive on the belt. This translates to

$$\mathtt{GuardedWait} \quad = \quad (\mathtt{CONTACT}\#\mathtt{WAITFOROBJECT});\mathtt{ACQUIRE}$$

6 Mechanization

This section contains a description of our prototype implementation. The purpose of this implementation is to demonstrate that the mechanization of our approach to identifying dynamics and constructing action descriptions based on them is indeed possible. The implementation is an action generation module designed to fit into our current planner-reactor framework. Its input is a world model described in $\mathcal{RS}$, and its output is a set of ITL action descriptions that the ITL planner can use to construct reactions or reactors. This prototype implementation is in PROLOG.

The generation of actions from world models is completed in three steps: First, the limitset of the world model is generated, to identify candidate dynamics in the environment. Secondly, those dynamics that can be sensed or affected by the agents sensors or actuators respectively, are identified and selected. This set is further reduced by considering only those dynamics that convey port information in a manner compatible with our definitions of denseness. Finally, ITL action schemas are generated from this reduced set of dynamics.

6.1 Limitset Generation

Recall from section 4.3.1 that a limit of a process `P` is a process that `P` has evolved into and which will not evolve into anything further; it 'caps' the evolution of `P`. The limitset of `P` is a set of couples, one for each limit of `P`, containing the limit and the conditions under which `P` will evolve to that limit.

The concept of process evolution is central to the definition of the limitset of a process. The easiest approach to implementing the limitset function is to implement evolution as *simulated execution*. Evolution is defined in [17] in terms of the simulated execution of each composition operator and the conditions under which basic processes terminate. This definition is easily implemented in PROLOG. To do this, of course, we need to construct a PROLOG notation for process names, parameters, and results and for composition operations. To simplify this exposition, however, we will continue to use $\mathcal{RS}$ notation for the PROLOG examples below.

The termination conditions of basic processes need to be supplied as an input to the implementation. We do this with the predicate **stopcondition(T,I,O,C)** defined for each basic process `T`, where **I** and **O** are the parameter and result lists of the process, and **C** is the termination condition (which may contain references to the parameters or results). These condition statements will eventually be used as part of the executability conditions and effects of candidate actions.

Here are the PROLOG definitions of the **stopcondition** predicate for some of the basic processes we have used in this paper:

stopcondition(	`Delay` ,	,[T]	, _	, **duration(T)**	).
stopcondition(	`Ran`	,[L,H]	,[N]	, (N=random(L,H))	).
stopcondition(	`In`	,[PN]	,[N]	, (N=commin(PN))	).
stopcondition(	`Out`	,[PN,V]	, _	, (commout(PN)=V)	).

The first statement establishes that the termination condition of $\mathtt{Delay}_T$ is the condition $duration(T)$. In this case, we interpret $duration(T)$ as that a time T has passed. The termination condition of $\mathtt{Ran}_{L,H}\langle N\rangle$ is that the result N is set to an arbitrary value between the values L and H. The termination condition for $\mathtt{In}_{PN}\langle N\rangle$ is that the result N is set to whatever communication arrives on the port PN. The termination condition for `Out` is interpreted similarly. These termination conditions need to be predicates that can be understood or translated by the ITL planner.

The evolves predicate **ev()** can now be written as follows, based on the **stopcondition** predicate and the definition of evolution in the Appendix.

ev(	$\mathtt{N}_P\langle R\rangle$	, `STOP`	**,C)**	**:-**	**stopcondition(** `N` **,P,R,C).**		
ev(	`X;Y`	**,Z**	**,C)**	**:-**	**ev(X,** `STOP` **,A),**	**ev(Y,Z,B),**	**C=A.B.**
ev(	`X:Y`	**,Z**	**,C)**	**:-**	**ev(X,** `STOP` **,A),**	**ev(Y,Z,B),**	**C=A.B.**
ev(	`X:Y`	**,Z**	**,C)**	**:-**	**ev(X,** `ABORT` **,C),**	**Z=** `ABORT` **.**	

and so on for `X | Y`, `X # Y` , `X:;Y` **and** `X::Y`

The notation $A.B$ indicates the temporal ordering between conditions A and B.

A process may evolve to more than one other process, so sometimes we will need to return a list of processes and conditions. For example, in the network `P#Q` either process could terminate first, so we need to return both possibilities. Applying the **ev()** function to the `TEM` world model (suitably transcribed into PROLOG, of course) we get:

?- ev($\mathtt{TEM}_{p,a}$**,P,C).**

P = [$\mathtt{Object}_{np}$ $\mathtt{Camera}_{na}$ $\mathtt{ReportPos}_{pos,p,ts}$]

C = [**(streamout(op,t0)=p) . (np=random(p+e,p-e))**
na=commin(ang)
duration(ts) . (p=commin(op)) . objectinview(p)
]

P and **C** above contain the same data as the *limitset*(`TEM`) shown in the previous section, but in a different format. Each list element of **P** is a limit process, and the corresponding list element of **C** is the condition under which that limit occurs. Notice also that **ev()** only lists the element of the network `TEM` that has become a limit, this is more convenient for generating action schemas than the limitset definition.

6.2 Selection of Candidate Dynamics

Having obtained the candidate dynamics, the next step is to determine which of these can be influenced by, or can influence the agent. We accomplish this by looking at the port connections between the sensory-motor interface of the agent and each of the dynamics. We define a PROLOG predicate **sensible()** (the dynamic can be sensed by the agent)

sensible(D,P,N) :- basicsense(P), usesport(N,P), usesport(N,D).

That is, dynamic **D** is sensible by the agent using sensory process **P** and port **N** if **P** is a basic sensory process and if **P** and **D** communicate via some port **N.** The predicate **effectable()** can be defined similarly for basic motor processes. Note that applying these predicates to the three candidate actions from the limitset analysis results in the first dynamic, that of `Object`, being rejected, and the sensory and motor connections for the second and third dynamics being identified.

?- effectable(`Camera`, **P1,N1), sensible(** `ReportPos`, **P2,N2).**

P1 = `motor1` **P2** = `sense1`
N1 = ang **N2** = pos

To validly translate port communications into ITL, it is necessary to ensure that the communications are dense in the manner we have already discussed. Definition 1 defines dense 'output-only' processes. Definition 2 defines local, i.e., input-to-output, denseness. Definition 3 gives the rule for determining under what conditions a locally dense process is dense. Definitions 1 and 2 require implementing a syntactic scan of the world model to determine for each process if it obeys the dense and/or local dense definitions. This only need be done once and off-line. For now, we assume that this scan has been done and we concentrate on implementing definition 3: A process is dense if it obeys definition 1, or if it is locally dense on a port and that port receives data from (if it's an input) or transmits data to (if it's an output) a dense process:

dense(IP,P) :- locallydense(IP,OP,P),usesport(OP,DP),not(DP=P), dense(OP,DP).

dense(OP,P) :- locallydense(IP,OP,P),usesport(IP,DP),not(DP=P), dense(IP,DP).

To finish the selection of candidate dynamics, we use **dense()** to ensure that the candidate dynamic is dense on the port returned by **sensible()** or **effectable().** In this case, both are. In our current system, this is the only use that is made of the *duration* information. After this step, the duration information is dropped from all conditions.

6.3 Generation of Action Schemas

The final step is to generate the ITL action schemas from the candidate dynamics. An ITL action schema has the following fields:

Name

Input Port and Basic Sensory Process

Output Port and Basic Motor Process

Temporal intervals used

Executability

Effects

Most of these are straightforward to fill in from the description of the dynamic and its associated sensory-motor connection. The executability condition is derived from the condition element of the dynamic. For example, the condition on the `ReportPos` dynamic is $duration(ts).(p = commin(op)).objectinview(p)$. After some cleaning up (described in the following paragraph), this roughly translates into saying that there should be an object in view. The effect comes from the limit process element of the dynamic, `ReportPos` in this example. However, this latter is a process. It is necessary to have a predicate **effects()** that translates a process into a language that the ITL planner can use, namely, a form similar to the condition generated by **ev()**. The mapping from process to effects, like the **stopcondition** predicate, needs to be supplied as input for our mechanization. For example, the effect of the process $\texttt{Camera}_{na}$ is that the camera position is set to na, and any object that is at position na is now in view. The effect of $\texttt{ReportPos}_{p,v,t}$ is that the object position is reported on port p.

There is also some minor housekeeping to be done. ITL actions refer to ports, not to values on the ports. Thus whenever we have a $v = commin(p)$ or $commout(p) = v$ statement in a condition or effect, we need to replace all occurrences of the variable v with p throughout the rest of that condition. We can then remove the *commin* or *commout* from the condition. The condition for the `ReportPos` dynamic then becomes $objectinview(op)$ (duration has been dropped after the denseness step, *op* has been substituted for p, and the *commin* dropped). Finally in all conditions and effects, we replace references to all ports which are connected to sensory or motor ports, with the name of the sensory or motor ports to which they are connected.

The following are the action schema generated by the two dynamics:

atp_actionschema camera-55

Input: [m1]

Output: [ ]

Intervals: [int-camera-55, int-54]

Executability:

objectposition(m1) @ int-54 and

ITL(int-54, [si,e,di,fi], int-camera-55)

Effects:

cameraposition(m1) @ int-camera-55 and

objectinview(m1) @ int-camera-55

endactionschema camera-55

atp_actionschema reportpos-57

Input: [ ]

Output: [s1]

Intervals: [int-reportpos-57, int-56]

Executability:

objectinview(op) @ int-56 and

ITL(int-56, [si,e,di,fi], int-reportpos-57)

Effects:

objectposition(s1) @ int-reportpos-57

endactionschema reportpos-57

6.4 Extensions

The purpose of this implementation has been to demonstrate that mechanization of the approach is feasible. The primary work was in mechanizing the limitset product, and this was based on simulated execution using the evolves operator. The limitset implementation is surprisingly robust, given its simplicity and the brute force approach taken. It doesn't explicitly know about variable or result scope, or about recursion, yet in general it handles both of these in a satisfactory manner. This is not altogether surprising for two reasons: Firstly, simulated execution is technique already used in many branches of computer science and is not that challenging to implement correctly. Secondly, much of the work of interpreting the limitset results is actually done by the ITL planner — the limitset implementation doesn't worry about variable scope, for example, because the ITL planner handles this in processing the executability condition and effects derived from the limitset.

Nonetheless, the limitset implementation used here is simplified in that it doesn't reason about port communication between concurrent processes, and certain kinds of recursion will trigger an endless loop. In [17], a set of limit theorems are developed that directly relate the structure of the limitset and the structure of the process network. Employing that approach to synthesizing the limitset promises a more efficient implementation, and one which handles recursion in a more elegant way.

The connectivity analysis, to determine which dynamic is connected to which sensory or motor port, is straightforward. Indeed, the entire processing could be done off-line and accessed when necessary. The denseness definitions 1 and 2 can also be performed off-line. However, definition 3 needs to be carried out for each dynamic. It can only be done off-line at the expense of analyzing all possible connections. The implementations of densely increasing/decreasing were not shown here, but can be addressed in a manner similar to the implementations of definitions 1 and 2; that is, carried out off-line and accessed on-line.

The implementations of the denseness definitions is in progress. The approach is that all processes in a process network that do not concern input or output on a port are reduced to time delays. In the case that a process uses only one port, or a single input and output port, the implementation is not difficult. In the case that more ports are involved, the problem may complicate, and we have not developed any general solutions yet.

7 Conclusions

We have presented an approach to identifying and exploiting dynamics — patterns of interaction — in the environment. The work has been done in the context of, but not yet integrated with, the planner-reactor architecture. The approach divides the problem into two related sub-problems: Firstly, the environment model is analyzed to identify dynamics. This analysis is carried out using the the limitset concept from the $\mathcal{RS}$ model. Secondly, the dynamics are used to construct high-level ITL actions, each representing a process with inputs and outputs, that can be used to construct reactions. The concept of the 'denseness' of the information on a communication port is important for addressing the timing and continuity issues. ITL reasoning can then be used to select high-level actions and connect them together appropriately. Although we did not pursue the derivation of the timing characteristics in reactions, we did lay the groundwork for it. That information becomes important in real-time scheduling of the processes that implement reactions.

There are a number of architectures in the literature similar to our planner-reactor architecture: Bresina & Drummond's ERE [4], McDermott's Transformational Planner[23] and Sutton's DYNA [31] among others. The principle novelty of the planner-reactor architecture is in the loose and well-defined interaction between planner and reactor, and in the characterization of the reactor as a concurrent network of processes using the $\mathcal{RS}$ language. These two features enable the formalization of the on-going, iterative process of adaptation to explore the issue of the convergence of the reactive system [19].

Our definition of the planner as incrementally generating reactions to exploit the environment has few comparisons in the planning literature. Chapman [6] introduces the definition of dynamics as acausal patterns of interaction. However, he does not investigate the problem of analyzing an environment to derive dynamics. Schoppers [26] addresses a very similar problem to ours. However, he chooses to make the dynamics of the environment explicit in his action descriptions in terms of temporal *soon* and *sust* operators. In our case, these dynamics are implicit in the process model description of the environment, and are only identified by the limitset calculation.

Our approach means that it's not necessary to be aware of the agent's objectives when constructing the environment model; no action repertoire needs to be modeled, justified and qualified a priori. Rather, the on-line construction of reactions based on the *current* environment enables an agent to achieve its objectives using environment dynamics which may only partly be in its control. The generated reactions capture 'closed loop' control law behavior, derived from 'open loop' actions. This context-sensitive abstraction of repetitive behavior is a major contribution of this approach.

We earlier made the comparison with Horswill's *habitat constraints* and his language of equivalence transformations to simplify agents so as to exploit their environment. Our limitset analysis allows us to derive dynamics that do work the agent may want to do and offer these as candidate portions of reactions. It does not address equivalence in the sense Horswill does, and it may be possible to enrich our system by developing analogs of his transforms in our limitset analysis.

There are two drawbacks to the approach. It presumes that a (possibly partial) model of the environment can be put together beforehand, and its usefulness hinges on the environment having dynamics to exploit that are relevant to the agents objectives. The latter, we argue, is a reasonable 'richness' assumption to make about 'real-life' environments. The former can only be addressed by learning.

Future work needs to address the following points:

1. Focusing in on a set of environment processes given the planner's objectives and constructing a subnet of the environment from these. (The *Snet* algorithm discussed in [17] may be of use in this problem.)

2. The planner-reactor policy of assumption relaxation needs to be incorporated. Assumptions are a second way to limit the size of the subnet of the environment model under consideration. However, the reactions chosen will need to obey the convergence constraints for the planner-reactor developed in [19].

References

[1] P. Agre and D. Chapman. Pengi: an implementation of a theory of action. In *Proc. AAAI-87*, pages 268–272, Seattle, WA, Oct. 1987.

[2] J.F. Allen. Towards a general theory of action and time. *Artificial Intell.*, 23(2):123–154, 1984.

[3] R. C. Arkin. Motor schema-based mobile robot navigation. *Int. J. Rob. Res.*, 8(4):92–112, Aug 1989.

[4] J. Bresina and M. Drummond. Integrating planning and reaction. In J. Hendler, editor, *AAAI Spring Workshop on Planning in Uncertain, Unpredictable or Changing Environments*, Stanford CA, Mar. 27–29 1990. Systems Research Center, Univ. of Maryland.

[5] R. Brooks. A robust layered control system for a mobile robot. *IEEE J. Rob. & Aut.*, RA-2(1):14–22, Mar. 1986.

[6] David Chapman. *Vision, Instruction, and Action.* MIT Press, Cambridge, 1992.

[7] S. Chien, M. Gervasion, and G. DeJong. On becoming decreasingly reactive: Learning to deliberate minimally. In *9th Machine Learning Workshop*, pages 288–292, 1991.

[8] R.J. Firby. Adaptive execution in complex dynamic worlds. Ph.D. Dissertation & Research Report YALEU/CSD/RR#672, Yale University, New Haven CT, Jan. 1989.

[9] K.D. Forbus. Qualitative process theory. *Artificial Intelligence*, (24):85–168, 1984.

[10] M. Ginsberg. Reasoning about action II: The qualification problem. *Artificial Intelligence*, (25):311–342, 1988.

[11] G. Hendrix. Modeling simultaneous actions and continuous processes. *Artificial Intelligence*, (4):145–180, 1973.

[12] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science, 1985.

[13] I. Horswill. Analysis of adaptation and environment. *Artificial Intelligence, Special Issue on Comp. Theories of Interaction*, TBA(TBA), 1993.

[14] L.P. Kaelbling. Goals as parallel program specifications. In *AAAI-88*, pages 60–65, 1988.

[15] D. Lyons and A. Hendriks. A practical approach to integrating reaction and deliberation. In *1st Int. AI Conference on Planning Systems*, Univ. of Maryland, College Park, MD, 1992.

[16] D.M. Lyons. A formal model for reactive robot plans. In *2nd International Conference on Computer Integrated Manufacturing*, RPI, Troy, NY, May 21–23 1990.

[17] D.M. Lyons. Representing and analysing action plans as networks of concurrent processes. *IEEE Trans. Rob. & Aut.*, 9(3), June 1993.

[18] D.M. Lyons and M.A. Arbib. A Formal Model of Computation for Sensory-Based Robotics. *IEEE Trans. Rob. & Aut.*, 5(3):280–293, June 1989.

[19] D.M. Lyons and A.J. Hendriks. Planning for reactive robot behavior. In *IEEE Int. Conf. Rob. & Aut.*, Apr. 7–12th 1992. (See also video proceedings).

[20] D.M. Lyons and A.J. Hendriks. Safely adapting a hierarchical reactive system. In *SPIE Intelligent Robots and Computer Vision XII*, September 1993.

[21] D.M. Lyons and A.J. Hendriks. Planning by adaptation: Experimental results. In *IEEE Int. Conf. Rob. & Aut.*, San Diego CA, May 1994.

[22] D.M. Lyons, A.J. Hendriks, and S. Mehta. Achieving robustness by casting planning as adaptation of a reactive system. In *IEEE Int. Conf. Rob. & Aut.*, Apr. 7–12th 1991.

[23] D. McDermott. Robot planning. Technical Report YALEU/CSD/RR#861, Yale University, Aug. 1991.

[24] R. Pelavin. *A Formal Approach to Planning With Concurrent Actions and External Events.* PhD thesis, Univ. of Rochester, Dept. of Computer SC., Rochester, NY, 1988.

[25] M. Schoppers. *Representation and Automatic Synthesis of Reaction Plans.* Technical Report UIUCDCS-R-89-1546 (PhD Thesis), Dept of Computer Science, University of Illinois at Urbana-Champaign, 1989.

[26] M. Schoppers. Building plans to monitor and exploit open loop and closed loop dynamics. In J. Hendler, editor, *AI Planning Systems*, pages 204–213, 1992.

[27] C.J. Sellers and S.Y. Nof. Performance analysis of robotic kitting systems. *Rob. & Comp. Integ. Manuf.*, 6(1):15–24, 1989.

[28] Y. Shoham. *Reasoning about change.* MIT Press Series in Artificial Intelligence. MIT Press, 1988.

[29] L. Spector and J. Hendler. Knowledge strata: Reactive planning with a multi-level architecture. Technical report UMIACS-TR-90-140, UM Institute of Advanced Computer Studies, Univ. of Maryland, Nov. 1990.

[30] H. Stark and F. Tuteur. *Modern Electrical Communications: Theory and Systems.* Prentice Hall New Jersey, 1979.

[31] R. Sutton. First results with Dyna. In J. Hendler, editor, *AAAI Spring Symposium on Planning in uncertain and changing environments*, Stanford CA, Mar. 27–29 1990. Systems Research Center, Univ. of Maryland.

References

[1] P. Agre and D. Chapman. Pengi: an implementation of a theory of action. In *Proc. AAAI-87*, pages 268–272, Seattle, WA, Oct. 1987.

[2] J.F. Allen. Towards a general theory of action and time. *Artificial Intell.*, 23(2):123–154, 1984.

[3] R. C. Arkin. Motor schema-based mobile robot navigation. *Int. J. Rob. Res.*, 8(4):92–112, Aug 1989.

[4] J. Bresina and M. Drummond. Integrating planning and reaction. In J. Hendler, editor, *AAAI Spring Workshop on Planning in Uncertain, Unpredictable or Changing Environments*, Stanford CA, Mar. 27–29 1990. Systems Research Center, Univ. of Maryland.

[5] R. Brooks. A robust layered control system for a mobile robot. *IEEE J. Rob. & Aut.*, RA-2(1):14–22, Mar. 1986.

[6] David Chapman. *Vision, Instruction, and Action.* MIT Press, Cambridge, 1992.

[7] S. Chien, M. Gervasion, and G. DeJong. On becoming decreasingly reactive: Learning to deliberate minimally. In *9th Machine Learning Workshop*, pages 288–292, 1991.

[8] R.J. Firby. Adaptive execution in complex dynamic worlds. Ph.D. Dissertation & Research Report YALEU/CSD/RR#672, Yale University, New Haven CT, Jan. 1989.

[9] K.D. Forbus. Qualitative process theory. *Artificial Intelligence*, (24):85–168, 1984.

[10] M. Ginsberg. Reasoning about action II: The qualification problem. *Artificial Intelligence*, (25):311–342, 1988.

[11] G. Hendrix. Modeling simultaneous actions and continuous processes. *Artificial Intelligence*, (4):145–180, 1973.

[12] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science, 1985.

[13] I. Horswill. Analysis of adaptation and environment. *Artificial Intelligence, Special Issue on Comp. Theories of Interaction*, TBA(TBA), 1993.

[14] L.P. Kaelbling. Goals as parallel program specifications. In *AAAI-88*, pages 60–65, 1988.

[15] D. Lyons and A. Hendriks. A practical approach to integrating reaction and deliberation. In *1st Int. AI Conference on Planning Systems*, Univ. of Maryland, College Park, MD, 1992.

[16] D.M. Lyons. A formal model for reactive robot plans. In *2nd International Conference on Computer Integrated Manufacturing*, RPI, Troy, NY, May 21–23 1990.

[17] D.M. Lyons. Representing and analysing action plans as networks of concurrent processes. *IEEE Trans. Rob. & Aut.*, 9(3), June 1993.

[18] D.M. Lyons and M.A. Arbib. A Formal Model of Computation for Sensory-Based Robotics. *IEEE Trans. Rob. & Aut.*, 5(3):280–293, June 1989.

[19] D.M. Lyons and A.J. Hendriks. Planning for reactive robot behavior. In *IEEE Int. Conf. Rob. & Aut.*, Apr. 7–12th 1992. (See also video proceedings).

[20] D.M. Lyons and A.J. Hendriks. Safely adapting a hierarchical reactive system. In *SPIE Intelligent Robots and Computer Vision XII*, September 1993.

[21] D.M. Lyons and A.J. Hendriks. Planning by adaptation: Experimental results. In *IEEE Int. Conf. Rob. & Aut.*, San Diego CA, May 1994.

[22] D.M. Lyons, A.J. Hendriks, and S. Mehta. Achieving robustness by casting planning as adaptation of a reactive system. In *IEEE Int. Conf. Rob. & Aut.*, Apr. 7–12th 1991.

[23] D. McDermott. Robot planning. Technical Report YALEU/CSD/RR#861, Yale University, Aug. 1991.

[24] R. Pelavin. *A Formal Approach to Planning With Concurrent Actions and External Events.* PhD thesis, Univ. of Rochester, Dept. of Computer SC., Rochester, NY, 1988.

[25] M. Schoppers. *Representation and Automatic Synthesis of Reaction Plans.* Technical Report UIUCDCS-R-89-1546 (PhD Thesis), Dept of Computer Science, University of Illinois at Urbana-Champaign, 1989.

[26] M. Schoppers. Building plans to monitor and exploit open loop and closed loop dynamics. In J. Hendler, editor, *AI Planning Systems*, pages 204–213, 1992.

[27] C.J. Sellers and S.Y. Nof. Performance analysis of robotic kitting systems. *Rob. & Comp. Integ. Manuf.*, 6(1):15–24, 1989.

[28] Y. Shoham. *Reasoning about change.* MIT Press Series in Artificial Intelligence. MIT Press, 1988.

[29] L. Spector and J. Hendler. Knowledge strata: Reactive planning with a multi-level architecture. Technical report UMIACS-TR-90-140, UM Institute of Advanced Computer Studies, Univ. of Maryland, Nov. 1990.

[30] H. Stark and F. Tuteur. *Modern Electrical Communications: Theory and Systems.* Prentice Hall New Jersey, 1979.

[31] R. Sutton. First results with Dyna. In J. Hendler, editor, *AAAI Spring Symposium on Planning in uncertain and changing environments*, Stanford CA, Mar. 27–29 1990. Systems Research Center, Univ. of Maryland.

A Denseness for Asynchronous composition

The following lemma and theorem establish a denseness result for asynchronous recurrent composition. We will show this result only for the periodic case (where delays of exactly γ and τ are incurred). The same results can be proven for the non-periodic case (where γ and τ are upper bounds) at the expense of a slightly more complex network.

Lemma 1:
For the network

$$\mathtt{ST} = (\mathtt{IN}_x\langle v\rangle;\mathtt{Delay}_{\gamma o})\ \ :;\ (\mathtt{STREAM}_{y,f(v)}\#\mathtt{Delay}_{\tau o})$$

for small values of γo and τo and a continuous function f then if y is $(\gamma i, \tau i)$-dense then x is $(\gamma i + \gamma o, MAX(\tau i, \tau o))$-dense.

This is straightforward to show, since the :; operator is defined recursively. Applying its definition to the network of Lemma 1 yields the exact form of the network in Definition 2. More difficult to show, and more useful in our case, is the next theorem:

Theorem 1:
For the network

$$\mathtt{AT} = (\mathtt{Delay}_{\tau o};\mathtt{IN}_x\langle v\rangle;\mathtt{Delay}_{\gamma o})\ \ ::(\mathtt{STREAM}_{y,f(v)}\#\mathtt{Delay}_{\tau o})$$

for small values of γo and τo and a continuous function f then if y is $(\gamma i, \tau i)$-dense then x is $(\gamma i + \gamma o, MAX(\tau i, \tau o))$-dense.

The two τo delay processes end up being activated in parallel. Output then ceases, input happens, and γo passes before the next output. Thus, the delays sequence the `IN` process to begin only once the previous `STREAM` network has been terminated. This allows the reduction of `AT` to the form `ST` which Lemma 1 establishes is dense.

B Definition of One-Step Process Evolution

The following definitions capture one-step evolution. All process terminations below are of basic processes.

- $\mathtt{P} \xrightarrow{\Omega\mathtt{P}} \mathtt{STOP}$ (definition of termination condition).
- $\mathtt{P} \xrightarrow{\mho\mathtt{P}} \mathtt{ABORT}$ (definition of abort condition).
- Sequential: $\mathtt{P};\mathtt{Q} \xrightarrow{\Omega\mathtt{P}\vee\mho\mathtt{P}} \mathtt{Q}$.
- Concurrent:
  - $(\mathtt{P} \mid \mathtt{Q}) \xrightarrow{\pi(cp,cq)} (\mathtt{P}' \mid \mathtt{Q}')\ iff\ \mathtt{P} \xrightarrow{cp} \mathtt{P}'\ and\ \mathtt{Q} \xrightarrow{cq} \mathtt{Q}'$ (where $\pi(c1, c2, \ldots)$ maps a set of conditions onto the disjunction of all their orderings).
  - $\Big|_{i\in I} \mathtt{P}i \xrightarrow{c} \mathtt{ABORT}$ iff all the processes abort, $c = \bigwedge_{i\in I} \mho\mathtt{P}i$.
  - $\Big|_{i\in I} \mathtt{P}i \xrightarrow{c} \mathtt{STOP}$ iff at least one process terminates successfully,

$$c = (\bigwedge_{i\in A} \mho\mathtt{P}i) \wedge (\bigwedge_{j\in S} \Omega\mathtt{P}j), \text{ where } A \cup S = I,\ A \cap S = \emptyset \text{ and } S \neq \emptyset.$$

- Conditional:
  - $\mathtt{P}\langle i\rangle : \mathtt{Q}_i \xrightarrow{\Omega\mathtt{P}} \mathtt{Q}_i$, and
  - $\mathtt{P}\langle i\rangle : \mathtt{Q}_i \xrightarrow{\mho\mathtt{P}} \mathtt{ABORT}$

  Different behavior on termination versus abortion is what makes this operator 'conditional.'
- Disabling:
  - $(\mathtt{P}\#\mathtt{Q}) \xrightarrow{\pi(cp,cq)} (\mathtt{P}'\#\mathtt{Q}')\ iff\ \mathtt{P} \xrightarrow{cp} \mathtt{P}'\ and\ \mathtt{Q} \xrightarrow{cq} \mathtt{Q}'$
  - $\#_{i\in I}\mathtt{P}i \xrightarrow{c} \mathtt{ABORT}$ iff all processes abort.
  - $\#_{i\in I}\mathtt{P}i \xrightarrow{c} \mathtt{STOP}$ iff at least one process terminates.

Note that disabling composition can force a component process to abort. Thus, components of disabling composition need to have their abort conditions extended to include the cases under which the network forces them to abort. It is important to distinguish these (necessary and sufficient) extended conditions $\bar{\mho}\mathtt{A}$ from the (necessary) spontaneous conditions $\mho A$. The relationship between these conditions is, for $\mathtt{A}$ in a disabling network with $\mathtt{P1}, \mathtt{P2}, \ldots$,

$$\bar{\mho}\mathtt{A} = \mho\mathtt{A} \vee (\bigvee_{\mathtt{P}i\neq A} (\mho\mathtt{P} \vee \Omega\mathtt{P})) \tag{16}$$