Representing and Analyzing Action Plans as Networks of Concurrent Processes.

D.M. Lyons
dml@philabs.philips.com

Philips Laboratories
North American Philips Corporation
345 Scarborough Road
Briarcliff Manor
NY 10510

Constructing action plans for a robot operating in an environment containing uncertain and dynamic events is a difficult task. Indeed, the inadequacy of the standard approaches for representing and producing plans for such environments has led some researchers to abandon explicit plan representation and to directly program behavior instead. Ultimately, no matter which approach is taken, producing appropriate behavior in such environments requires writing robot programs or action plans that include conditionals, loops, requests for sensory data, concurrency, etc. Existing approachs are simply not adequate to the task of modeling and analyzing such plans. Nonetheless, it is necessary that such plans be open to formal analysis: the complexity of control necessary to operate in uncertain and dynamic environments demands that more than human intuition be used to verify, or preferably autogenerate, such plans.

This paper addresses the problem of constructing a plan representation that can deal with the complexity of representing and analyzing robot behavior in uncertain and dynamic environments. The key contributions are as follows. A concurrent-process based representation is developed which represents *both* the plan (or controller) and the uncertain and dynamic environment in which the plan operates. A methodology is outlined for analyzing the behavior of this interacting system of plan and world. This methodology is illustrated with a mixed-batch example from the domain of robotic kitting. To balance the theoretical work, a description of the implemented robot kitting cell is presented.

`Keywords: Plan Representation, Action Plan, Robotics, Planning, Robot Programming, Process-Based, Plan Analysis, Process Algebra.`

To be published, IEEE Transactions on Robotics and Automation 1993 .

Representing and Analyzing Action Plans as Networks of Concurrent Processes.

D.M. Lyons
dml@philabs.philips.com

Philips Laboratories
North American Philips Corporation
345 Scarborough Road
Briarcliff Manor
NY 10510

1 Introduction

Constructing action plans for a robot operating in an uncertain and dynamic environment is a difficult task. An uncertain and dynamic environment is one in which the robot has uncertain knowledge of what events can happen and in which agents other than the robot can produce dynamic effects. Existing plan representations, such as situation calculus (e.g. STRIPS[7]), procedural nets (NOAH[32]) and AND-OR Graphs [13], cannot be applied in these environments without the aid of a program such as a reactive plan exector, e.g., SROMA [37]. This approach can be so ineffective that it has encouraged some researchers to completely abandon the notion of explict plan representation and instead to directly program robot behavior, e.g., Brooks [2], Agre & Chapman [3].

No matter which approach is taken, actually producing appropriate behavior on a robot operating in an uncertain and dynamic environment requires writing flexible, detailed robot control programs. These programs need a rich vocabulary of conditionals, loops, sensory requests, etc., and are usually written as a set of concurrent, communicating modules. Existing plan representations were not designed to represent plans at this level of complexity. They present highly simplified actions and plan control structures. As McDermott [27] has pointed out, in operating in uncertain and dynamic environments, the line between robot plan representation and robot programming language blurs. Arguably, the only remaining difference is that a plan representation has an additional requirement that it be possible to reason formally about plan properties. This paper is motivated by the observation that since we ultimately have to write flexible, detailed robot programs, it is appropriate that we develop a plan representation that can capture and reason about such programs.

In previous work, we have developed a foundation for representing robot control programs as concurrent, communicating process networks. In [18, 19], we proposed the $\mathcal{RS}$ (Robot Schemas) model, a special model of computation designed for robot programming. That work set forth the set of special characteristics of robot computation and developed a model of concurrent computation to suit this set. In this paper, we develop an algebraic approach to specifying and analyzing $\mathcal{RS}$ process networks. This approach was first outlined in [17] and is being used in on-going work to formalize a reactive planning paradigm [23, 21]. In this paper, the algebraic system is employed to analyze the interactions between a flexible robot plan and an uncertain and dynamic environment. We follow the lead of discrete-event control [30, 15, 11] in considering it important to model *both* the robot plan *and* the environment in which the plan is to be carried out. It is not sufficient to verify a plan against an informal model of the environment: This may suffice in a static, well-understood environment; however, both dynamic events and uncertain events introduce sufficient 'branching' complexity into an environment model to render an informal approach untrustworthy. The approach developed in this paper models both plan and environment as a couple of concurrent, interacting process networks, and begins the development of a methodology to analyze this interaction.

Our view of uncertainty and dynamics will essentially be a *discrete-event* one; that is, we shall not directly focus on the geometric interpretation of these phenomena as many other authors have, e.g., Donald [4], Ellis [6], Hutchinson & Kak [14], etc. For the problem domain we shall introduce, robotic kitting (described in Section 2), geometric reasoning is not a major part of the control problem. Introducing specific techniques to handle geometric uncertainty is quite compatible with the discrete-event approach developed here. Indeed, one source of the uncertainty data describing discrete events might be a geometric model.

This paper is laid out as follows. We begin by presenting our problem domain: implementing a kitting robot. We will take a representative running example from this domain, that of kitting for mixed-batch production runs. The list of desiderata at the end of that section captures the objectives of the remainder of the paper. Section 3 looks at existing approaches to plan representation and why they are inadequate. Section 4 is an informal introduction to the $\mathcal{RS}$ model and the notation used throughout the paper. It concludes by developing an $\mathcal{RS}$ plan to carry out the kitting task. The point here will not be the novelty of the plan — indeed, we claim that writing this kind of program is effectively inescapable in real-world robotics — rather the point is that here we present the plan in a way directly amenable to formal reasoning. Section 5 introduces the concept of modeling uncertain and dynamic environments. The 'Traffic World' of Sanborn and Hendler[33] is used as an illustrative example, showing that the notation previously introduced for plans can also describe environments. A more formal view of $\mathcal{RS}$ is presented in Section 6, establishing the background for a sample liveness and efficiency analysis of the mixed-batch kitting example in Section 7. These two sections present a first attempt to construct a methodology for analyzing plans as interacting process networks. The more detailed formal material supporting these sections is presented in the appendices at the end of the paper.

2 Problem Domain: Reactive Kitting

A kitting robot is a robot system that puts together assembly kits. Simpler and cheaper automation can construct the assemblies once they have been placed in the kits and routed appropriately. Instead of building a factory full of expensive, intelligent robots, manufacturers can focus the intelligence and cost into a small number of kitting robots which feed the rest of the factory. Kitting provides the line stock availability, the line scheduling flexibility, and the reduced station cycle time productivity for competitive results [34]. However, to achieve this ideal, the kitting robot has to 'iron out' all the uncertainties associated with the assembly process so that it can be done by simpler automation.

The kitting robot typically has to deal with the following sources of uncertainty and dynamics:

1. Variable arrival rates and poses of incoming parts. The primary job of the kitting robot is to take parts (perhaps directly from stores) with high position and pose uncertainty and to put them in a kit.

2. Errors in incoming parts. Faulty parts need to be removed before they either cause errors in downstream automation, or become assembled into (faulty or low-quality) products.

3. Substitutability of incoming parts. Some available parts may be substitutable for other, unavailable parts.

4. Mixed batches of kits. The robot may be serving more than one line, or the lines may be running mixed batches.

5. Variable availability of resources. Examples of resources are tools or machines with which the the robot needs to coordinate directly.

6. Response to events in the downstream automation. For example, machines breaking down, or alternate machine configurations.

7. Response to factory management advice. For example, response to changing the batch mix or throughput, or implementing once-off fixes for special situations.

Thus, 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 the uncertain and dynamic events in its environment. This work emphasizes the discrete-event nature of the environment. However, nothing we develop will be incompatible with the use of more specific geometric work on uncertainty.

An assembly plant will typically produce some number (potentially in the hundreds) of *variants* of each of their assembly products. Each variant is essentially the same product but with a number of special parts to tailor it for a particular customer or market. The kitting robot is fed assembly components and produces kit trays with all the parts for each variant. These trays are then fed to hard automation machinery which produces the finished assembly with high quality and high speed. As a running example, we will look at an assembly kit for a product with four variants as follows:

$$TRAY \longleftarrow \left\{ \begin{array}{c} MOTOR1 \\ or \\ MOTOR2 \end{array} \right\} + \left\{ \begin{array}{c} SWITCH.SOCKET + \left\{ \begin{array}{c} SWITCH1 \\ and \\ SWITCH2 \end{array} \right\} \\ or \\ SEALED.DOUBLE.SWITCH \end{array} \right\} \tag{1}$$

Consider the situation where the factory is producing all four variants in mixed-batch and we need to program the kitting robot to correctly produce the four variants depending on what parts are fed to it. The uncertainty here concerns the arrival times of instances of the parts. Arrival events are dynamic in that multiple part instances can be piling up as kitting proceeds. Fox & Kempf [9] introduced a useful efficiency property that is relevant in this example, *opportunism.* We say that the kitting behavior is opportuistic if the robot chooses which variant to construct based on what parts have arrived to date.

The following are our list of desiderata for solving this example: Firstly, we want to be able to represent this plan in the same manner that we will need to execute it on a computer. That is, as a highly conditional program phrased as concurrent and communicating processes. Secondly, we want this representation to be amenable to formal analysis. Our objective is to represent the executable form of the plan — the form that all of us who use robots eventually end up writing as a set of C-programs or whatever — in such a manner that it can be reasoned about formally. The immediate goal of formal reasoning in this paper will be to verify properties of the plan; our long-term goal is to be able to automatically generate such plans based on a high-level description of the plan requirements.

Finally, we are not satisfied to verify a plan against an informal description of the robot's environment. There are cases where this may be sufficient, e.g., a static, certain environment. However, both dynamic events and uncertain events introduce sufficient 'branching' complexity into an environment model to render an informal approach untrustworthy. The perspective of discrete-event control is highly illuminating on this issue [30]: both controller (plan) and plant (environment) are modeled in the same formal representation and their mutual interaction is then analyzed[1].

It is worth repeating here that we don't claim novelty for this example or the program we ultimately build to solve it; rather the point here is that we will be able to capture models of both the executable, conditional plan and the uncertain and dynamic environment and reason formally about their interaction.

3 Background

STRIPS [7] demonstrated the use of *situation calculus,* in which a robot action is modeled as a state to state transition described by a list of preconditions and effects. This approach turned out to have severe disadvantages. Since the robot is the only effector of change, it becomes impossible to represent the actions of agents other than the robot, e.g., the asynchronous arrival of parts in our mixed batch example. If the representation is relaxed to allow other sources of change then the issue of how to model the resulting concurrent actions arises. Furthermore, the STRIPS concept of a plan is a single fixed sequence of actions. This is inefficient and possibly disasterous in a dynamic environment, since the environment may change and render useless the one action sequence generated.

Later work addressed some of these problems. The constraint-posting approach [36] addressed the issues of external and concurrent events. Sacerdoti's NOAH [32] introduced the *procedural net* plan representation, later used in many systems, e.g., NONLIN[35], SIPE[36]. This allows plans to be defined as partially ordered sequences of actions. However, the action representation still had little support for control structures such as loops and conditional actions, and robot essential issues such as sensory requests were omitted. As a recent example, despite the powerful uncertainty reasoning present in the Spar assembly planner [14], the plan representation it ultimately generates, although detailed enough to be executed, is not very flexible. Some researchers have focused on the problem of producing a rich and flexible plan language, e.g., Firby's RAPS [8] and McDermott's RPL [26]. To achieve this end, however, they have sacrificed the ability to precisely reason about plan execution, one of the nicer features of the simpler plan representations. It is not clear if this is a permanent disadvantage; indeed, the work presented in this paper could be used to construct, say, a process semantics for RAPS.

The AND/OR graph (Hypergraph) is an important plan representation introduced by Homem de Mello and Sanderson [13] for assembly planning. An AND/OR graph represents the valid states of the assembly; but not the necessary actions or sensing to identify or achieve those states. In this sense it is more akin to a subgoal decomposition than to an action plan. Could the graph simply be relabelled to specify actions rather than states? Unfortunately, AND/OR graphs then demand an explicit enumeration of all the possible courses of action. This can lead to state explosion with even simple uncertain and dynamic environments such as the mixed batch example where it becomes possible to have multiple instances of the same part.

The usual solution to bridging the gap between a plan representation such as those discussed above and the real-world behavior necessary from the robot is to introduce a reactive plan exector, e.g., SROMA [37]. In general, this does provide a working solution. However, it introduces some extra problems too: we may lose whatever formal guarantees the plan was constructed with, since the exector needs to be ably to modify the plan; or alternatively, we may miss opportunities or encounter errors

[1] Indeed, Heyman [11] has a DEC approach which is initially very similar to the approach we will demonstrate here.

if the exector is prevented from modifying the plan. This problem arises because the plan generation process has now been artificially divided over two modules.

As a reaction to the inflexible behavior that existing plan representation and planner control architectures produce, some reseachers have disavowed having *any* explicit plan representation, e.g., Brooks [2], Agre & Chapman [3] (see [22] for a review). Instead they control the robot via the interaction of a number of concurrent behavior-generating modules or programs. They have the full flexibility of a (robot) programming language in building these modules, and can thus specify behavior that would be impossible to specify with the plan representations discussed. These researchers have demonstrated that versatile and robust behavior can be produced in unstructured environments. However, in disavowing traditional approaches, they have lost the ability to formally explore the effects of executing a plan. We argue that formal methods are needed *especially* in unstructured environments, since the more complex the environment, the less reliable human intuition will be in determining plan correctness or efficiency.

In summary, to generate appropriate behavior in a uncertain and dynamic environment, it is necessary to use an action representation capable of capturing detailed programs including conditionals, loops, sensory commands, and concurrent, interacting processes. However, to make this representation useful from a plan verification and generation perspective, it should be open to formal analysis. There is some work along these lines. Rosenschein's *situated automata* [31] provide a way to reason about the correctness of reactive machines. However, in that formalism, while the reactive machine is cast precisely, the environment in which the machine operates is captured by an implicit 'background theory' assumed to contain all the 'right' information about what the world can do. We advocate following the lead of the discrete-event control literature to address this problem [30, 15, 11]: both controller (plan) and plant (environment) are modeled in the same formal representation and the interactions in their concurrent composition analyzed. This means that the environment can be modeled with the same formality as the plan.

In the next section we introduce a plan representation that can deal with the complexity of representing and analyzing the kind of programs that are unavoidable in programming a robot to produce appropriate behavior in an uncertain and dynamic environment. The emphasis will be on introducing the notation and representing plans. In Section 5 we will discuss using the same notation to represent uncertain and dynamic environments. Finally, in Sections 6 and 7 we present a methodology for analyzing the interactions between plan and environment.

4 Representing Action Plans using the $\mathcal{RS}$ Model

In this section, we present an approach in which robot action plans are represented as networks of concurrent processes. In [19] we argued that this is an appropriate model for robot programming. Here we build on that approach to develop a plan representation which has the power of a programming language. The first step in defining our plan representation will be to define what we mean by a process. We will then define how processes are combined together to produce plans, and how individual processes can communicate with each other. We then present, initially without formal proof, the $\mathcal{RS}$ plan for the mixed batch kitting problem of Section 2.

Processes. A process is a unique locus of computation [19]; it could refer to a piece of hardware, or a time-sliced software thread, or a physical agent of change. A description of a process, or network of processes, is called a *schema*. For example, $\texttt{Joint}_{j=v}$ denotes a process that is an instance of the schema $\texttt{Joint}$ with parameter j set to value v (when unambiguous, we write $\texttt{Joint}_v$). Processes can return results when they terminate, e.g., $\texttt{Joint}_v\langle p\rangle$ denotes that $\texttt{Joint}$ has one parameter v and

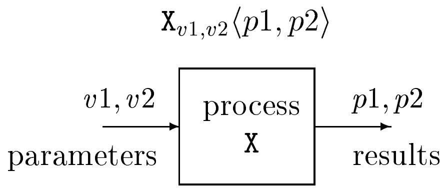

Figure 1: Notation for Process Parameters/Results

returns one result value p (see Figure 1). Formally, a process is defined as a special kind of automaton [19].

We distinguish two conditions under which a process terminates: a successful termination, a *stop*, or an unsuccessful termination, an *abort*. Processes can be deterministic or non-deterministic. A non-deterministic process when executed with identical parameters and under identical conditions may still produce differing results. (This is necessary to model uncertain environments).

Processes are built into networks using several kinds of *process composition operators*. These operators allow us to express the decision making in an action plan as a (conditional, usually based on sensory data) composition of more atomic processes. By using the techniques of process algebra[12, 10], such operators can also be used in the analysis of the behavior of a plan. However, we postpone that perspective for now.

Composition Operators. Processes can be defined in terms of compositions of other processes using the composition operators. Ultimately, all definitions must ground out to compositions of a set of atomic, pre-defined, processes. The set of *basic schemas* defines the 'building blocks' that can be used to construct networks, and is discussed later in this section. There are six composition operators in $\mathcal{RS}$ (summarized in table 1). The first two, sequential and concurrent, are common to almost any process model. The second two, conditional and disabling, are pretty much unique to this model. The final two are simply useful nonatomic combinations of the first four, used to capture iteration. Formally, process composition operators are functions that construct a composite automaton from a set of argument automata [24].

The most straightforward composition in $\mathcal{RS}$ is *sequential* composition. The process $\mathtt{T} = \mathtt{P;Q}$ is simply the network of the process $\mathtt{P}$ executed first, and when that terminates, process $\mathtt{Q}$ is executed until it terminates. This is used to enforce a strict ordering on operations, e.g., $\mathtt{Place}_{part1};\mathtt{Place}_{part2}$.

Concurrent composition indicates that two or more processes should be carried out concurrently, e.g., $\mathtt{T} = (\mathtt{P} \mid \mathtt{Q})$. This allows us to represent a lack of ordering between activities, e.g., $(\mathtt{Place}_{part1} \mid \mathtt{Place}_{part2})$, or parallel actions — actions which need to be done simultaneously, e.g., squeezing an object *obj* with two fingers $f1$ and $f2$: $(\mathtt{ApplyForce}_{f1,obj} \mid \mathtt{ApplyForce}_{f2,obj})$. Two unusual features of this operator are that it is *idempotent* and that concurrent processes can exchange messages using *communication ports*. The latter will be discussed below. The former states that all references to a process $\mathtt{P}$ are treated as referring to the same process, not multiple copies of the process, i.e., $(\mathtt{P} \mid \mathtt{P}) = \mathtt{P}$. If we need to distinguish multiple identical copies, then a superscript will be used[2].

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

[2]This is necessary to include nondeterministic processes, e.g., network (6) in Section 5.

1. Sequential Composition: $\mathtt{T} = \mathtt{P};\mathtt{Q}$. The process T behaves like the process P until that terminates, and then behaves like the process Q (regardless of P 's termination status).
2. Concurrent Composition: $\mathtt{T} = (\mathtt{P} \mid \mathtt{Q})^c$. The process T behaves like P and Q running in parallel and with the input ports of one connected to the output ports of the other as indicated by the port-to-port connection map c. This can also be written as $\mathtt{T} = (\bigm|_{i\in I} \mathtt{P}i)^c$ for a set of processes indexed by I.
3. Conditional Composition: $\mathtt{T} = \mathtt{P}\langle v \rangle : \mathtt{Q}_v$. The process T behaves like the process P until that terminates. If P aborts, then T aborts. If P terminates normally, then the value v calculated by P is used to intialize the process Q, and T then behaves like $\mathtt{Q}_v$.
4. Disabling Composition: $\mathtt{T} = \mathtt{P}\#\mathtt{Q}$. The process T behaves like the concurrent composition of P and Q until either terminates, then the other is aborted and T terminates. At most one process can stop; the remainder are aborted.
5. Synchronous Recurrent Composition: $\mathtt{T} = \mathtt{P}\langle v \rangle :; \mathtt{Q}_v$. This is a recursively defined as follows: $\mathtt{P} :; \mathtt{Q} = \mathtt{P} : (\mathtt{Q}; \mathtt{P} :; \mathtt{Q})$.
6. Asynchronous Recurrent Composition: $\mathtt{T} = \mathtt{P}\langle v \rangle :: \mathtt{Q}_v$. This is recursively defined as follows: $\mathtt{P} :: \mathtt{Q} = \mathtt{P} : (\mathtt{Q} \mid (\mathtt{P} :: \mathtt{Q}))$.

Operator Precedence: The operator precedence from loosest to tightest is as follows: Concurrent; Disabling; Sequential; Conditional; Synchronous Recurrent; Asynchronous Recurrent.

Table 1: Summary of $\mathcal{RS}$ Composition Operators.

not depends on whether `LidOpen` terminates successfully or not. This composition can also be used to parameterise the consequent process. For example, in $\mathtt{Locate}_{part1}\langle p \rangle : \mathtt{Place}_p$, `Locate` might search for an instance of *part*1 and return an instance pointer p which is used to direct `Place` to acquire and place that object.

Disabling composition allows one process to terminate another. The network `T = A#B` behaves like (A | B) except that it terminates whenever *either* process terminates. This captures the logic of a guarded move, for example, $\mathtt{MoveToward}_x\#\mathtt{Contact}$, where if a contact causes `Contact` to terminate this terminates the motion command `MoveToward`.

The final two composition operators are defined recursively in terms of these four. *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.

It is also convenient to introduce a 'not' operator. Strictly speaking this is *not* a composition operator, it simply changes the termination status of a process: `~STOP = ABORT` and `~ABORT = STOP`. However, this simplifies the writing of conditional statements.

Message Passing. The notation so far allows us to describe networks of processes with various orderings between them. However, it does not yet support allowing concurrent processes to communicate with each other; that is the next step. A *port* is a communication object associated with a process. Messages are written to, and read from ports by a special subset of basic schemas called communication schemas, described below. Ports which receive messages are called *input ports*, and those which transmit messages are called *output ports.*

To indicate how ports are connected to one another, concurrent composition has a third, op-

1. `STOP` terminates successfully immediately on creation. It has no ports and takes no parameters. Formally, `STOP` is mapped to an automaton that has a null transition map and whose start state is one of its termination states [24].
2. `ABORT` terminates unsuccessfully immediately on creation. It has no ports and takes no parameters.
3. `INF` never terminates once created. It has no ports and takes no parameters.
4. $\mathtt{IN}_p\langle v\rangle$ reads its input port p and can pass on that value in v if used in a conditional composition. It terminates as soon as the value is read. This schema implements full synchronous reception of messages on port p.
5. $\mathtt{OUT}_{p,v}$ writes the value v to its output port p. It terminates once the write has completed. This schema implements full synchronous transmission of messages on port p.
6. $\mathtt{DELAY}_t$ terminates after a period of duration t measured according to some universal clock.
7. `TERM` terminates successfully after waiting for a period of unknown duration.
8. $\mathtt{SELECT}_{N,\phi}\langle v\rangle$ selects a value $v \in N$ with probability given by the distribution ϕ over N and then terminates successfully, allowing the value generated to be passed on using composition.
9. $\mathtt{EXISTS}_S\langle p1, p2, \ldots\rangle$ will terminate iff an instance of schema S exists. The results $p1, p2, \ldots$ are the values of the parameters of the instance of S.

Table 2: List of $\mathcal{RS}$ Basic Schemas.

tional argument: a port to port *connection relation.* This relation is a set of couples $op \mapsto ip$ indicating that port ip and op are connected; whatever is written to op can be read from ip. For example, if the ports of `P` and `Q` were connected by a relation c, the concurrent composition would be written $(\mathtt{P} \mid \mathtt{Q})^c$. In $N = (\mathtt{P0} \mid \ldots \mid \mathtt{Pn})^c$, the domain of c is the set of input ports in N and the range is the set of output ports in N.

Basic Schemas. Basic schemas describe atomic processes and are the fundamental vocabulary from which process networks can be built. We will not be able to describe or analyze behavior below this level, so it is wise to have basic schemas implement small, well-defined units of behavior. (Section 8 describes some basic schemas that we have built to interface to our robot workcell.)

The set of basic schemas is partitioned into basic sensory schemas, basic motor schemas, and basic task schemas. Task schemas are those that perform internal computation. The communication schemas alluded to earlier are examples: $\mathtt{OUT}_{p,v}$ synchronously writes value v to port p and then terminates, and $\mathtt{IN}_p\langle v\rangle$ syncronously reads a value from p and can pass it on via conditional composition in v. Sensing and motor schemas delineate the interface between a controller and its environment. An example of a motor schema is $\mathtt{Joint}_{i,v}$ which results in robot joint i being set to position v. An example of a sensory schema is $\mathtt{BLOB}_b\langle x, y\rangle$ which returns the coordinates x, y of some blob b in a camera image. Plans are constructed by linking sensory schemas to motor schemas via computations performed by task schemas. The set of general-purpose basic schemas shown in table 2 will be assumed in the remainder of the paper and will be augmented with problem specific schemas as necessary.

4.1 The Mixed-Batch Kitting Problem

We now construct a robot program to address the mixed batch kitting problem introduced in section 2. We assume the following basic schemas: $\mathtt{Locate}_m\langle p\rangle$ does a sensory search of the environment for an object of class m and terminates returning a pointer to it in p when it is found[3]. $\mathtt{Place}_p$ acquires and places part p in the assembly kit. We will specify the sensory search for, and placement of, a part a as follows, $\mathtt{Locate}_a\langle p\rangle : \mathtt{Place}_p$. The conditional composition operation ':' links the two basic processes so that part a is searched for and then placed. We will abbreviate the object model name as follows: t for the $TRAY$, $m1$ for $MOTOR1$, $m2$ for $MOTOR2$, sc for $SWITCH.SOCKET$, $s1$ for $SWITCH1$, $s2$ for $SWITCH2$, and sds for $SEALED.DOUBLE.SWITCH$.

The placement of the tray and one of the motors is specified as follows:

$$\mathtt{Part1} = \mathtt{Locate}_t\langle p\rangle : \mathtt{Place}_p \ ; \ (\ \mathtt{Locate}_{m1}\langle p\rangle \ \# \ \mathtt{Locate}_{m2}\langle p\rangle\ ) : \mathtt{Place}_p \tag{2}$$

The sequential composition forces the tray t to be placed first, when and if it arrives. The disabling composition between the processes to locate $m1$ and $m2$ ensures that as soon as *either* part is found the place is triggered.

Choosing between one or more actions depending on conditions is handled by using the idempotance of concurrency. A choice of actions $\mathtt{A}i$, $i \in I$ each depending on conditions $\mathtt{C}i$ has the following structure:

$$(\ \Big|_{i\in I}\ \mathtt{C}i : \mathtt{A}i)\ \mid\ (\mathop{\#}_{i\in I}\mathtt{C}i) \tag{3}$$

The disabling composition ensures that only *one* condition is taken; the first one to terminate. That condition is then the only one to trigger an action from the set of conditional compositions; all the other conditional compositions are aborted when their conditions are aborted. (If the conditions are mutually exclusive, e.g. an IF-THEN-ELSE statement, then the disabling composition is not necessary.)

The conditional choice of sc followed by the two switches or the sealed version sds is implemented as follows:

$$\begin{array}{rl} \mathtt{Part2} = & (\ \ \mathtt{Locate}_{sc}\langle p\rangle \# \mathtt{Locate}_{sds}\langle p\rangle \\ & \ \mid \mathtt{Locate}_{sds}\langle p\rangle : \mathtt{Place}_p \\ & \ \mid \mathtt{Locate}_{sc}\langle p\rangle : (\mathtt{Place}_p\ ; \\ & \qquad\qquad\qquad (\ \mathtt{Locate}_{s1}\langle p\rangle : \mathtt{Place}_p \\ & \qquad\qquad\qquad\ \mid \mathtt{Locate}_{s2}\langle p\rangle : \mathtt{Place}_p\ \ )\ ) \\ &) \end{array} \tag{4}$$

Again the disabling composition forces one of sds or sc to be found, and aborts the locate of the other. But note that in aborting the other locate process, by idempotence, the conditional compositions that refer to that process are forced to terminate. For example, if an instance of part sds is found the following events occur: $\mathtt{Locate}_{sds}\langle p\rangle$ terminates, forcing $\mathtt{Locate}_{sc}\langle p\rangle$ to abort, this forces the `Place` in $\mathtt{Locate}_{sds}\langle p\rangle : \mathtt{Place}_p$ to trigger, and the entire composition $\mathtt{Locate}_{sc}\langle p\rangle\ :\ (\ldots)$ to abort. In the case that an instance of sc is found, then it is placed. The concurrency in subsequently searching for, and placing, $s1$ and $s2$ means that these operations will be done in whatever order the environment enables. The full program is $\mathtt{Plan} = \mathtt{Part1}; \mathtt{Part2}$.

It is not possible to describe the behavior of this plan in any concise way without referring to the environment in which the plan is being carried out. The appropriate unit to analyze therefore, is

[3]See Section 8 for a note on our implementation of this; Section 7 will present the formalization of the effects of this process.

a plan embedded or situated in an environment. However, to do that, we need to have a formal way to represent environments. Our approach is to use $\mathcal{RS}$ to build models of the environment. We call these *world models.* In the next section, we describe how the same notation we have used to build plans can also capture models of dynamic and uncertain environments.

5 Uncertain and Dynamic World Models

By an uncertain environment, we mean one in which there is uncertainty about whether or not an event will occur, how many times and when it occurs, and what numerical parameters are associated with it when it does occur. By a dynamic environment, we mean one in which change occurs as time passes. Sanburn & Hendler's [33] 'Traffic-World' is an example of a dynamic and uncertain environment. In that domain, an agent tries to cross a road against a stream of traffic travelling at unknown, and dynamically changing, velocities. The traffic contains three sorts of cars: cars that try to avoid the agent, cars that ignore the agent, and cars that try to hit the agent. The agent is unaware in advance of the arrival times, quantities and types of the cars. We now show how $\mathcal{RS}$ can be used to capture such an environment.

An instance of `TERM` is a process that terminates at some unknown, finite time t after it has been created. We can represent the fact that event X will happen at some future stage as the composition `TERM;X`. The `SELECT` basic schema is used to model uncertain parameter values. We can use `SELECT` to provide initial parameter values v to X.

$$\mathtt{TERM}\ ;\ \mathtt{SELECT}_{N,\phi}\langle v\rangle : \mathtt{X}_v \tag{5}$$

At some arbitrary time t after this network has been started, it will produce the process $\mathtt{X}_n$ with n chosen from N according to ϕ. However, if we want the values of the parameters of `X` to change dynamically (e.g., the velocity in Traffic-World cars) then we need a recurrent process (see below).

The fact that event X *may* take place at some future time is represented as[4]

$$(\mathtt{TERM}^1 : \mathtt{X})\#\mathtt{TERM}^2 \tag{6}$$

where $\mathtt{TERM}^2$ could terminate at any stage and thus abort the potential to have `X` occur.

An aperiodic, but repeated event is represented as `TERM:;X.` The `DELAY` process would be used instead of `TERM` to represent a periodic event. A car in Traffic-World, for example, would be represented as:

$$\mathtt{Car}_{\phi v,\phi t} = (\mathtt{SELECT}_{V,\phi v}\langle v\rangle \mid \mathtt{SELECT}_{T,\phi t}\langle t\rangle)\ \ :;\ (\ \mathtt{Drive}_v\ \#\ \mathtt{DELAY}_t\ ) \tag{7}$$

This process maintains velocity v for time t using the distributions ϕv and ϕt to choose values. The process `TERM ::` $\mathtt{Car}_{\phi v,\phi t}$ would at random times create a `Car` process whose driving behavior was governed by ϕv and ϕt. Sanburn's different types of cars can be built by choosing different values for ϕv and ϕt. However, for cars of type three (cars that try to hit the agent) the values of ϕv and ϕt need to be a function of the agents current position.

A periodic event of unknown period would be represented as

$$\mathtt{SELECT}_{N,\phi}\langle t\rangle : (\mathtt{DELAY}_t\ :;\mathtt{X}) \tag{8}$$

where N and ϕ can be used to capture the uncertainty quantitatively if this data is available. We can represent the fact that an event will occur between time bounds $tmin$ and $tmax$ as follows:

$$\mathtt{DELAY}_{tmin}\ ;\ (\mathtt{DELAY}_{tmax-tmin}\#\mathtt{TERM});\mathtt{X} \tag{9}$$

[4]Since `TERM` is nondeterminstic, we need to use superscripts to distinguish the multiple identical copies in this example.

The event is forced to happen at a time later than $tmin$. TERM can force the event to occur at any point up to $tmax$, at which point the second delay process forces the event to occur.

6 Analysis of Behavior

Formal analysis provides us with an objective tool for evaluating the behavior of a plan in an environment. We argue that this is particularly needed in the case of action plans which operate in an uncertain and dynamic environment, since the complexity of interaction between world and plan may lead to non-intuitive plan behavior. The two main tools we develop for this analysis are the evolves operator, to express how networks change over time; and the algebraic properties of the composition operators (i.e., what operators distribute over what others, etc), to allow us to 'rewrite' process networks in alternative equivalent forms (e.g., to simplify network descriptions). The composition operators were designed specifically to have straightforward algebraic properties, and we will not spend much time on them here. Appendix A contains a full description. The emphasis in this paper is on the use of the process evolution operator to explore the interaction of plan and world when both are modeled in $\mathcal{RS}$.

6.1 Process Evolution

To analyze how world models or plans execute over time, it is important to be able to derive how process networks evolve as component processes dynamically terminate or are created. 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 to which P can possibly become equal on condition c, then P necessarily becomes equal to exactly one of these when c occurs. To apply this operator, we need to associate with each basic schema P the conditions under which it stops, $\Omega\mathtt{P}$. In some cases, we will also need to discuss the conditions under which a process *aborts* itself; these necessary (but not sufficient[5]) conditions we write as $\mho P$.

A network of processes can also *unconditionally* evolve. For example, a network of an input process and an output process on connected ports p and $\hat{p}$ unconditionally evolve to effect a transfer of the message:

$$(\mathtt{OUT}_{p,v} \ \mid \ \mathtt{IN}_{\hat{p}}\langle x\rangle : \mathtt{A}_x)^c \longrightarrow \mathtt{A}_v \qquad \textit{where } c:\ p \mapsto \hat{p} \tag{10}$$

This formula axiomatically describes synchronous communication in our model.

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. For example, a sequential chain of processes $\mathtt{T} = \mathtt{P1;P2}$, where P1 and P2 are basic, can only change as follows

$$\mathtt{P1;P2} \overset{\Omega\mathtt{P1} \vee \mho\mathtt{P1}}{\longrightarrow} \mathtt{P2}$$

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

$$\textit{poss}(\mathtt{P}) = \{(c, \mathtt{Q}) \textit{ such that } \mathtt{P} \xrightarrow{c} \mathtt{Q}\} \tag{11}$$

[5] A process can also be forced to abort, but we will see how to include those conditions later.

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:

$$\mathtt{P} \xrightarrow{c1} \mathtt{P1} \xrightarrow{c2} \mathtt{P2} \xrightarrow{c3} \ldots \xrightarrow{cn} \mathtt{Q} \iff \mathtt{P} \xrightarrow{c} \mathtt{Q} \qquad \textit{where } c = c1.c2.c3\ldots.cn. \tag{12}$$

We write temporal ordering of conditions using a period, e.g., $A.B$ is read A then B. Notice that if a network composition terminates, then either all the processes terminated together, or they terminated in some arbitrary order. For convenience, we will write the termination condition of a network of processes as the disjunction of all orderings of the termination conditions. For example, for a network of two processes $(\mathtt{A} \mid \mathtt{B})$ the (successful) termination conditions would be $(\Omega\mathtt{A}.\Omega\mathtt{B}) \vee (\Omega\mathtt{B}.\Omega\mathtt{A}) \vee (\Omega\mathtt{A} \wedge \Omega\mathtt{B})$. For convenience, we will define a function $\pi(\Omega\mathtt{A}, \Omega\mathtt{B}, \ldots)$ that maps a set of conditions onto the disjunction of all their orderings.

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':

$$poss^*(\mathtt{P}) = poss(\mathtt{P}) \cup \Big(\bigcup_{(c,\mathtt{Q}) \in poss(\mathtt{P})} \{(c.\hat{c}, \hat{\mathtt{Q}}) \textit{ such that } (\hat{c}, \hat{\mathtt{Q}}) \in poss^*(\mathtt{Q})\}\Big) \tag{13}$$

Or, equivalently, in terms of the transitive evolves operator as:

$$poss^*(\mathtt{P}) = \{(c, \mathtt{Q}) \textit{ such that } \mathtt{P} \xrightarrow{c} \mathtt{Q}\}$$

For any interesting robot plan, this set is at least huge, and often infinite. Much of our work will be involved in trying *not* to have to calculate this.

6.2 Process Limits

A *bound* process is a process such as `STOP` or `ABORT` or `INF` that does not evolve into anything else. These processes are interesting because if they occur in a chain of evolutions then they immediately 'cap' that chain. We can use this idea to define the notion of a *limit* for a process: A 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`, $limitset(\mathtt{P})$.

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

A subset of the limitset of a process is the set of `STOP` or `ABORT` limits. These limits indicate that not only will the process no longer change its behavior, it is now additionally stopped or aborted. The conditions under which this occurs is captured as follows:

$$endconditions(\mathtt{P}) = \{c \textit{ such that } (c, \mathtt{STOP}) \textit{ or } (c, \mathtt{ABORT}) \in limitset(\mathtt{P})\}$$

We have already made the point that $poss^*$ is difficult to calculate. Yet, the limit definitions seem to demand this calculation. In practice, we are relieved of this problem because it is possible to determine for some networks what their limitset or periodic behavior is from their composition structure. We call theorems that establish this relationship, *limit theorems*. Appendix D contains a list of seven useful limit theorems and their proofs. We will refer to these theorems as we need them in the remainder of the paper.

Note: We will make frequent use of recursive process definitions. This style naturally leads to considering the periodic behavior of processes. We say a process `P` is periodic if it 'reinvokes' itself in all possible evolves chains. Appendix C contains a more formal definition of the recursive, periodic and semi-periodic behavior of processes.

6.3 Plan and World as a System

To analyze the interactions of a plan, `Plan`, and a model of the environment, `World`, it is necessary to somehow combine them so that they can interact. This composition needs to allow both processes to proceed asynchronously and concurrently except for those occasions where they influence each other. We follow the discrete-event control literature [11] in considering the controlled system of plan and world as the concurrent composition of the plan and world processes $(\mathtt{Plan} \mid \mathtt{World})$.

In a concurrent network $N = \Big|_{i \in I} \mathtt{P}^i$, any one of the concurrent processes, e.g. $\mathtt{P}^k, k \in I$ is referred to as a *component* of the network, and with respect to any one process, the others $\Big|_{i \in I - \{k\}} \mathtt{P}^i$ are called the *co-network* or simply co-net. A key point of this work is that it involves the analysis of plan and world as coupled concurrent processes. In the case of plans that reach a limit process, it will be useful to determine the nature of the world process when the plan process has reached a limit. We define a limit co-net of a concurrent network with respect to one of its components as the co-net remaining when that component has reached a limit process. In particular, we define the stop co-net, $scn()$ as

$$
\begin{aligned}
scn(\mathtt{P},\mathtt{N}) = (c,\mathtt{R}) & \\
& i)\;\; \mathtt{P}\ \textit{is a component of}\ \mathtt{N},\ \textit{and} \\
& ii)\;\; \mathtt{N} = (\mathtt{P} \mid \mathtt{Q}) \xrightarrow{c} \mathtt{R}\ \textit{for}\ c \in \textit{endconditions}(\mathtt{P})
\end{aligned}
$$

And the set of $scn(\mathtt{P},\mathtt{N})$ of a network with respect to a process is called the $scnset(\mathtt{P},\mathtt{N})$. The *scnset* forms the basis of the methodology we introduce to analyze plan and world interactions. In the next section we will re-introduce the mixed-batch kitting example, and develop some straightforward techniques for computing its *scnset*.

Note: In the case of processes that are periodic rather than once-off, the appropriate analysis would concern the $scnset()$ of the periodic component of the plan process with the world model.

7 Analysis of the Mixed-Batch Kitting Example

Let us return to the mixed batch kitting plan of section 4.1: There are two questions we want to ask of the plan behavior:

1. Will it construct all and only all four variants?
2. Will it behave opportunistically?

The first is called a *liveness* property, it asks whether the plan will achieve a stated objective, and the second is called an *efficiency* property, it asks how well the plan will achieve a stated objective.

World Model. We start by building a model of an environment where any of the parts can be delivered to the robot at any time. We model the fact that a part is delivered to the robot at arbitrary time as $\texttt{TERM} :: \texttt{Part}_m$ where m is one of $Parts = \{t, m1, m2, sdc, sc, s1, s2\}$. The complete world model is:

$$\texttt{Winit} = \Big|_{m \in Parts} \texttt{TERM} :: \texttt{Part}_m \tag{14}$$

meaning that any instance of the parts can arrive at any time (and thus they can arrive in any order). Note that there could be multiple instances [6] of each $\texttt{Part}_m$. Over time, as parts 'arrive', this world model will evolve to include Wfin , a network consisting of instances of the $\texttt{Part}_m$ schema:

$$\begin{aligned} &\texttt{Winit} \stackrel{time(t)}{\longrightarrow} (\texttt{Wfin}_V \mid \texttt{Winit}) \\ &\texttt{Wfin}_V = \Big|_{i \in V \subset Parts} \texttt{Part}_i \end{aligned} \tag{15}$$

The effect of carrying out a $\texttt{Place}_m$ operation on a part will be modeled by the $\texttt{Placed}_m$ process, indicating that that part has been inserted into the assembly kit. The result of assembling some variant of the assembly kit requiring the subset of parts $V \subset Parts$ is a process $\texttt{Assem}_V$, where

$$\texttt{Assem}_V = \Big|_{i \in V} \texttt{Placed}_i. \tag{16}$$

Sensory-Motor Interface We now have our plan (from (2) and (4) in Section 4.1) and our world model (from (14)(15) and (16)), but we haven't indicated how they interact. The relationship between `Locate` and `Part,` and between `Place` and `Placed` needs to be formalized.

The connection between `Locate` and `Part` can be captured using the basic schema `EXISTS` as follows:

$$\texttt{Locate}_m\langle p\rangle = \texttt{EXISTS}_{\texttt{Part}_m}\langle m\rangle \tag{17}$$

That is, the `Locate` process in the plan will be 'triggered' whenever `Part` processes occur in the world model. This defines the sensory interface for the plan.

The desired relationship between `Place` and `Placed` is that carrying out a `Place` should result in the object being placed; represented here by an instance of the the `Placed` schema. We define an addition to the world model that is 'triggered' by instances of the `Place` process:

$$\texttt{Placelaw} = \texttt{EXISTS}_{\texttt{Place}_m}\langle p\rangle :: (\texttt{Delay}_{tp}; \texttt{Placed}_p) \tag{18}$$

where tp is a (constant) time value, modeling a fixed time taken to carry out the place motion. The principle purpose of `Place` is to have the effect of triggering the `Placelaw` process. Therefore, `Place` does not have to do any internal processing. It could simply be defined as `STOP.` It seems more appropriate however, to define it as a delay of time tp.

$$\texttt{Place}_p = \texttt{Delay}_{tp} \tag{19}$$

The two networks, (19) and (18), interact together to achieve the following:

$$(\texttt{Place}_x \mid \texttt{Placelaw}) \stackrel{time(tp)}{\longrightarrow} (\texttt{Placed}_x \mid \texttt{Placelaw}) \tag{20}$$

[6] We could assign a unique name to each instance of each of the part models which arrive; however, this is not necessary and complicates the analysis.

This can be easily verified by algebraic substitution and application of the evolves definition. The details are presented in Appendix E. (20) defines the motor interface for the plan.

Note that we could have used message-passing, instead of `EXISTS`, to implement the sensory-motor interaction. In general message-passing is a more realistic tool. `EXISTS` was chosen here because it yields a shorter analysis for this problem.

There is now sufficient detail to use the evolution operators to 'simulate' an execution of the plan and determine what effects it produces on the world model. While this is a useful exercise (the details are presented in Appendix E), it will only tell us about one possible way the plan can be executed. To satisfy our liveness question, it is necessary to be able to say something about the result of *all* possible execution sequences. For this we use the *scnset* concept developed in the previous section.

The Liveness Question. The plan is designed to terminate once one assembly variant has been kitted. Thus, we are interested in the stop co-net of $(\mathtt{Plan} \mid \mathtt{World}_{\emptyset})$ with respect to `Plan`, where

$$\mathtt{World}_V = (\mathtt{Winit} \mid \mathtt{PlaceLaw} \mid \mathtt{Wfin}_V). \tag{21}$$

The stop co-net — the world model process once the plan has terminated — should contain all and only all valid assembly variants. The only conditions necessary for achieving a variant should be that all the parts for that variant have arrived and that a sufficient amount of time has elapsed. Let $VV \subset 2^{Parts}$ be the set of the four valid variants. Our liveness question can be phrased as

$$(\ time(t)\ ,\ \ (\mathtt{World_U} \mid \mathtt{Assem_V})\ )\ \in\ scnset(\mathtt{Plan}, (\mathtt{Plan} \mid \mathtt{World}_{\emptyset}))\quad \Longleftrightarrow\quad V \in VV\ and\ V \subseteq U \subseteq Parts. \tag{22}$$

If we attempt to calculate the *scnset* by looking directly at $poss^*(\mathtt{Plan} \mid \mathtt{World}_{\emptyset})$ then we will have to explore all the internal evolutions in both the plan and world models. It would be simpler if we only had to explore those evolutions in which the plan influenced the world or vice-versa. Plan and world processes only interact through the sensory-motor interface. (We have already identified this boundary for the kitting example).

The 'Transfer Process' Approach. A *transfer process* `TP` for a process `P` is a process in which all the internal evolution steps of `P` have been transitively closed between the sensory and motor interfaces. The transfer process is then analogous to the usual transfer function of linear systems theory in that it provides an input-output view. Investigating the concurrent composition of the transfer functions $(\mathtt{TPlan} \mid \mathtt{TWorld})$ is easier than investigating the composition $(\mathtt{Plan} \mid \mathtt{World})$ because only $poss()$, not $poss^*()$ needs to be examined, since the only elements of $poss()$ are the direct interactions of plan and world.

The transfer process is used in the remainder of this example as follows. Rather than producing `TP` and `TW`, the limit theorems of Appendix D are used to derive $poss(\mathtt{TP})$ and $poss(\mathtt{TW})$, which are the more useful quantities. A short algorithm is then presented to derive the *scnset* from these two quantities.

The Plan Transfer Process, `TP`. The sensory-motor interface for the mixed-batch kitting example are the processes `Place` and `Placed`, and `Locate` and `Part`. To determine $poss(\mathtt{TP})$ it is necessary to determine how the plan effects a linkage between `Locate` and `Place` processes. We accomplish this by 'cutting' the plan into sensory-motor segments. That is, we will divide up the plan into segments by determining which `Locate` process affect which `Place` process. We can do this by tracing the parameter dependancy of a `Place`, terminating when ever we get to a `Locate`. We call the resultant network the *Snet* (sensory network) of that `Place` process. Any processes or composition operators

$$\begin{array}{lcll} Snet1 & = & (\Lambda & , \ \texttt{Locate}_t\langle p\rangle : \texttt{Place}_p) \\ Snet2 & = & (\Omega\texttt{Snet1} & , \ (\texttt{Locate}_{m1}\langle p\rangle \# \texttt{Locate}_{m2}\langle p\rangle) : \texttt{Place}_p) \\ Snet3 & = & (\Omega\texttt{Snet2} & , \ (\texttt{Locate}_{sc}\langle p\rangle \# \texttt{Locate}_{sds}\langle p\rangle \ \mid \\ & & & \quad \texttt{Locate}_{sds}\langle p\rangle : \texttt{Place}_p \ \mid \ \texttt{LP}_{sc})) \\ Snet4 & = & (\Omega\texttt{LP}_{\texttt{sc}} & , \ (\texttt{Locate}_{s1}\langle p\rangle : \texttt{Place}_p \mid \texttt{Locate}_{s2}\langle p\rangle : \texttt{Place}_p)) \end{array}$$

In the above, $m \in \{m1, m2\}$, Λ indicates the empty (always true) condition, and $\texttt{LP}_{sc} = \texttt{Locate}_{sc}\langle p\rangle : \texttt{Place}_p$.

Table 3: The Snet segments for the mixed batch assembly plan.

$$\begin{array}{lcl} limitset(Snet1) & = & \{ \ (\exists\texttt{Part}_t, \texttt{Place}_t) \ \} \\ limitset(Snet2) & = & \{ \ (\exists\texttt{Part}_{m1}, \texttt{Place}_{m1}) \ , \ (\exists\texttt{Part}_{m2}, \texttt{Place}_{m2}) \ \} \\ limitset(Snet3) & = & \{ \ (\exists\texttt{Part}_{sc}, \texttt{Place}_{sc}) \ , \ (\exists\texttt{Part}_{sds}, \texttt{Place}_{sds}) \ \} \\ limitset(Snet4) & = & \{ \ (\pi(\exists\texttt{Part}_{s1}, \exists\texttt{Part}_{s2}), (\texttt{Place}_{s1} \mid \texttt{Place}_{s2})) \ \} \end{array}$$

Table 4: The limitsets of the mixed batch assembly Snets.

encountered between the motor and sensory processes are included into the Snet. This segmentation is straighforward to implement (see the algorithm in Appendix F.)

With each Snet we associate a condition, which is the least condition necessary for this segment to begin execution. If we carry this segmentation out on `Plan` we get the four Snets shown in table 3. To calculate the $poss(\texttt{TP})$, we need to look at the *limitsets* for these Snets under the assumption that `Place` is a bound (since we're interested in the set of next placement actions the plan may produce). Using the limit theorems these can be derived as shown in table 4.

World Transfer Process, `TW.` The Snet segmentation algorithm need not be applied to the world model, since it is already phrased as separate sensory-motor segments. As in `TP,` $poss(\texttt{TW})$ is determined by looking at the *limitset* of the segments.

`Place` affects the world model through `Placelaw.` The limitset of `Placelaw` is constrained by theorem 5 as follows:

$$\{(\exists\texttt{Place}_m, \texttt{Placed}_m) \textit{ such that } m \in W \subset Parts\} \subset limitset(\texttt{Placelaw}) \tag{23}$$

where we adopt the notational convention $\Omega\texttt{EXISTS}_{\texttt{Place}_\texttt{m}} = \exists\texttt{Place}_m$. Since we can't say how often or with what part `Placelaw` will be triggered, we can only say that $W \subset Parts$ parts could end up placed.

The sensory process `Locate` is essentially just a test for an instance of `Part,` $\Omega\texttt{Locate}_\texttt{m} = \exists\texttt{Part}_m$. `Winit` is the only part of the world model that can generate `Part` instances. The limitset of `Winit` is also constrained by theorem 5 as follows:

$$\{(time(t), \texttt{Part}_m) \textit{ such that } m \in Parts, t \geq 0\} = limitset(\texttt{Winit}) \tag{24}$$

where $\Omega\texttt{TERM} = time(t), t \geq 0$ and since for any $t1, t2$ there exists a $t3$ such that $time(t1).time(t2) = time(t3)$.

We now have everything we need to verify property (22): $poss(\texttt{TP})$ is specified by tables 3 and 4, and $poss(\texttt{TW})$ by equations (23) and (24).

$$
\begin{aligned}
scnset(\texttt{Plan}, (\texttt{Plan} \mid \texttt{World}_\emptyset)) &= scnset(\texttt{TP}, (\texttt{TP} \mid \texttt{TW})) \\
&= \{\ (\exists\texttt{Part}_t.\exists\texttt{Part}_{m1}.\exists\texttt{Part}_{sds}, (\texttt{World}_{U1} \mid \texttt{Assem}_{V1})), \\
&\quad (\exists\texttt{Part}_t.\exists\texttt{Part}_{m2}.\exists\texttt{Part}_{sds}, (\texttt{World}_{U2} \mid \texttt{Assem}_{V2})), \\
&\quad (\exists\texttt{Part}_t.\exists\texttt{Part}_{m1}.\exists\texttt{Part}_{sc}.\pi(\exists\texttt{Part}_{s1}, \exists\texttt{Part}_{s2}), \\
&\qquad\qquad (\texttt{World}_{U3} \mid \texttt{Assem}_{V3})), \\
&\quad (\exists\texttt{Part}_t.\exists\texttt{Part}_{m2}.\exists\texttt{Part}_{sc}.\pi(\exists\texttt{Part}_{s1}, \exists\texttt{Part}_{s2}), \\
&\qquad\qquad (\texttt{World}_{U4} \mid \texttt{Assem}_{V4}))\ \}
\end{aligned}
$$

$$\textit{where } Vi \subset Ui \textit{ and } Vi \in VV.$$

Table 5: A listing of $scnset(\texttt{Plan}, (\texttt{Plan} \mid \texttt{World}_\emptyset))$.

C	t	m1	m2	sc	s1	s2	sds	done
$\emptyset$	1	0	0	0	0	0	0	0
t	0	1	1	0	0	0	0	0
t,m*	0	0	0	1	0	0	1	0
t,m*,sds	0	0	0	0	0	0	0	1
t,m*,sc	0	0	0	0	1	1	0	0
t,m*,sc,s1	0	0	0	0	0	1	0	0
t,m*,sc,s2	0	0	0	0	1	0	0	0
t,m*,sc,s*	0	0	0	0	0	0	0	1

This table defines the assembly kitting constraints for the variant assembly kit example as a function $C(row, column)$ returning 1 iff that column entry can be placed once the assembly kit is the state indicated by the row. For example $C(t, m1) = 1$ indicates that m1 can be added iff the tray is in place. By m^* and s^* we denote either $m1$ or $m2$ and either $s1$ or $s2$. The constraint $C(state, done) = 1$ iff the assembly kit is finished.

Table 6: Variant Assembly Constraints.

Calculating $scnset(\texttt{TP}, (\texttt{TP} \mid \texttt{TW}))$. The *scnset* can be calculated by repeated application of the following three steps which compose the effects of the two transfer processes.

1. Determine which Snet segment, *Sneti*, of the plan is enabled (table 3; initially only Λ holds). Terminate if no segment holds.
2. Determine from $poss(\texttt{TW})$ (24) and $limitset(Sneti)$ what $poss(\texttt{TP} \mid \texttt{TW})$ will result.
3. Use $poss(\texttt{TW})$ (23) to determine the effect of this on the world.

The *scnset* for this example is shown in table 5. By (24), $\exists\texttt{Part}_m$ can be simply replaced by $time(t_m)$. Note also that a succession of times $t1.t2.t3\ldots$ or a permutation of times $\pi(t1, t2, t3, \ldots)$ can always be simplified to a single time value. Give these two facts, table 5 collapses to property (22). Note that we haven't addressed the correctness of the component sequence in this example. This can be addressed by considering the sequence of conditions in the *scnset*.

The Opportunism Question. Any assembly will have associated with it a set of constraints that minimally order the assembly actions based on geometrical and stability reasons. These constraints are captured for the kitting example as a function $C(S, c) = 1$ iff the kit is now in state S and component c can be added to it (see Table 6). The state of the kit can be simply represented by the set of parts assembled so far.

Opportunism can now be defined as the property possessed by the plan `P` if all next possible actions of `TP` when the world `W` is in some state $S(\mathtt{W})$ match exactly those specified by the assembly constraints C for this state *and* if the the remainder of the plan is also opportunistic. We write this formally as

$$OPP(\mathtt{TP}, \mathtt{TW}) \iff [\forall c. [C(S(\mathtt{TW}), c) = 1\,] \Leftrightarrow (time(t), (\mathtt{A}_c; \hat{\mathtt{TP}} \mid \hat{\mathtt{TW}})) \in poss(\mathtt{TP} \mid \mathtt{TW}) \wedge OPP(\hat{\mathtt{TP}}, \hat{\mathtt{TW}})]$$

where $\mathtt{A}_c$ is the motor action to which $C(S, c)$ refers; in our example this is $\mathtt{Place}_p$, where $p \in Parts$.

We can verify that the mixed batch assembly plan obeys this definition of opportunism by using the same three-step iteration used for liveness, but with step two is augmented as follows. For each non-zero entry in C, $C(S(\mathtt{W}), c) = 1$, there must be an appropriate entry in $poss(\mathtt{TW} \mid \mathtt{TP})$ that indicates that the action $\mathtt{Place}_c$, $c \in Parts$ is a possible next action of the plan transfer process: $(time(t), (\mathtt{Place}_c; \hat{\mathtt{TP}} \mid \hat{\mathtt{TW}})) \in poss(\mathtt{TW} \mid \mathtt{TP})$. Furthermore, such actions should be the only possible next actions. Tables 3 and 4 can again be used to calculate $poss(\mathtt{TW} \mid \mathtt{TP})$ in this step. Straightforward repetition of this augmented three-step loop shows the mixed batch kitting plan is indeed opportunistic.

8 Implementation Environment

We have constructed a robot workcell to explore the use of $\mathcal{RS}$ in robot action planning in the kitting domain.

Hardware. A Puma-560 robot equipped with a four-fingered force-sensing gripper is the basis of the kitting workcell. The workcell has two cameras: A global camera situated above the workcell, whose view covers the entire workspace, and a local camera, embedded in the 'palm' of the gripper, whose view is determined by the position and pose of the gripper. The remainder of the cell consists of the worksurface and a conveyer belt on which parts can be introduced or removed from the worksurface. The kitting task consists of taking parts as and whenever they come in, putting them into trays, and shipping the trays out on the conveyer belt.

The Puma's VAL controller is connected by an RS232 line to one of a set of M68020 processors on a common VMEBUS. This processor functions as a robot server processor. It can issue commands to the robot in two modes: a coarse position mode (VAL `MOVE`) and a fine position/velocity mode (VAL `ALTER`).

The two cameras are connected to a Philips PAPS industrial vision system. This system is controlled by another of the M68020 processors on the common VMEBUS. The processor functions as a vision server processor. A `recognize` command causes this server to scan the image from one of the cameras and determine if there are any instances of a given set of 2D object models in the image.

The `RS-L3` System. A subset of the $\mathcal{RS}$ model has been implemented as a robot programming language. The subset is basically $\mathcal{RS}$ as introduced in this paper except that concurrent and disabling compositions are not idempotent. The `RS-L3` programming environment consists of a YACC/LEX-based parser and a thread-based exector. The parser accepts systems of process definition equations in a computer-keyboard version of the $\mathcal{RS}$ syntax. It passes on these parsed definitions, as they are made, to the exector, which executes them as dictated by the model semantics. Thus, all execution is interpretive. The exector implements composition, port-i/o, buffer management, and scheduling. It uses the SunOS® light-weight process library to implement concurrency.

A set of basic sensory and motor schemas has been built to interface to the robot and vision servers. These basic schemas run as atomic processes that send command messages to the servers and return results from them. For example, $\mathtt{Locate}_m$ is a basic sensory process whose implementation (invisible in $\mathcal{RS}$) continually queries the vision server with `recognize(m)` commands, and only terminates when such a model instance is seen. $\mathtt{Domove}_p$ is a basic motor process that transmits a `move(p)` command to the robot server, and terminates when that command has been carried out.

Implementation Experience. Kitting examples have been constructed and run in $\mathcal{RS}$ on the kitting workcell as part of an investigation into combining planning and reaction [23, 21]. (The system can be seen in action on the 1992 IEEE Rob. & Aut. VideoProceedings.) That work does not make use of any of the process-based reasoning techniques described here; it uses $\mathcal{RS}$ as a robot programming language. Nonetheless, it has generated some interesting feedback:

1. Idempotance. This plays an important role in both analysis and representation, yet it is next to impossible to implement as described. To be correct, the $\mathcal{RS}$ scheduler must not only ensure that networks such as $(\mathtt{A} \mid \mathtt{A})$ are unified to `A`, but also that networks such as $(\mathtt{TERM}^1 : \mathtt{A} \mid \mathtt{TERM}^2 : \mathtt{A})$ unify to `A` in the case that both `TERM` processes happen to terminate simultaneously. Not only is this difficult to implement, it can lead to unexpected unifications when it does occur. This is a serious flaw, since without idempotance, the representation of choices between actions (e.g. eq. (3)) becomes very difficult. It would be much better from an implementation perspective to be able to name process instances and rely on that to define idempotance.

2. Kitting examples of the type described in this paper generally give rise to from 50 to 100 $\mathcal{RS}$ processes when they are implemented in detail. This does not include any world description. Such a description is required for reasoning about the effects of actions but not for executing the actions. This gives rise to an immediate problem: 50 to 100 threads ($\mathcal{RS}$ is implemented on top of a threads package) is a heavy computational load. The problem is relatively easily fixed. Every $\mathcal{RS}$ process does not need its own thread; only parallel or disabling compositions actually require the creation of new threads. This change results in very few threads being created, typically 10 to 15 in kitting examples.

3. Fair scheduling among 10 to 15 threads almost always ensures that some $\mathcal{RS}$ processes do not execute as often as they need. We have not discussed process time-constraints in this paper, though we have looked at the problem elsewhere [20]. For now, we program around the problem, but it remains a difficulty inherent in multi-process systems.

Automated Reasoning. Our use of the implemented $\mathcal{RS}$ system did not involve the implementation of the reasoning algorithms presented in this paper. Instead, we chose to automate planner reasoning about $\mathcal{RS}$ networks using an Interval Temporal Logic (ITL) reasoner[28]. There were several disadvantages to this approach: Firstly, there was the problem of having to map between the procedural $\mathcal{RS}$ program and the coresponding ITL constraints[25]. Secondly, there was the fact that ITL reasoners have to settle for having either reduced representation power or incompleteness. Automating the $\mathcal{RS}$ analysis techniques of this paper could potentially bypass both of these issues. The evolves operator has been implemented in PROLOG, based on the definitions in Appendix B, but it has *not* been used in conjunction with the execution system yet.

A process network is rephrased in prefix notation to simplify PROLOG analysis: `A:B` is written $colon(\mathtt{A},\mathtt{B})$, etc. Most of the implementation is a straightforward translation of the evolves definition into PROLOG clauses, e.g.,

$$ev(colon(\mathtt{X},\mathtt{Y}),\mathtt{Z},c)\ :-\ ev(\mathtt{X},\mathtt{STOP},a),\ ev(\mathtt{Y},\mathtt{Z},b),\ c=then(a,b).$$

where the prefix form of $a.b$ is $then(a,b)$. The only difficulty is in dealing with the effect of idempotance in networks such as ($\mathtt{Err}\#\mathtt{Oper} \mid \mathtt{Err} : \mathtt{Rec}$). Note that if `Err` terminates, then `Oper` is aborted and `Rec` is triggered; but if `Oper` terminates, then `Err` is aborted and `Rec` will never be triggered. (This network captures the logic of error detection `Err` and recovery `Rec` associated with an action or plan `Oper.`) To model these effects we use interaction clauses of the form:

$$ev(\mathtt{Z},\mathtt{STOP},c)\ :-\ \mathtt{Z}=net(\mathtt{P},\mathtt{Q}),\mathtt{P}=colon(\mathtt{X},\mathtt{Y}),\mathtt{Q}=hash(\mathtt{X},\mathtt{U}),ev(\mathtt{U},\mathtt{STOP},c).$$

However, without making use of the limit theorems, $poss^*$ boils down to a forward search procedure. The limit theorems allow the leaves of the reachability tree to be estimated based not on expanding out the tree (as $poss^*$ does), but rather on the initial structure of the network. As yet, the limit theorem approach has not been integerated into the implementation of $poss^*()$, though one straightforward approach would be to use them to build a set of limit clauses by analogy with the interaction clauses above. The Snet and *scnset* algorithms given in this text have been implemented for simple cases.

9 Conclusion

In summary, producing appropriate behavior in a robot operating in an uncertain and dynamic environment requires a plan representation that has both the rich vocabulary of loops, conditionals, sensory requests and concurrency necessary to represent flexible robot control programs and also a methodology for formally reasoning about the effects of such programs in a given environment. This paper has introduced a process-based representation, $\mathcal{RS}$, to address this problem. This representation uses a vocabulary of composition operators and basic proceses to support the description of detailed concurrent control programs. A version of the model, `RS-L3` has been implemented to control an experimental robot kitting workcell.

This paper presents groundwork for reasoning about process networks. An algebraic framework has been introduced for analyzing the interactions between a plan and the environment in which the plan is operating. A formalization of the ultimate possible effects of a plan on its environment, the *scnset*, was introduced. To simplify the calculation of the *scnset*, two supporting concepts were introduced: the *limitset* of a process and the *transfer* process. The *limitset* of a process describes the process or processes into which that process may finally evolve. A set of theorems was also presented for relating process structure to the contents of the *limitset*. Finally, the concept of the *transfer* process was introduced to simplify analysis of coupled plan-world systems. These concepts were then used to explore *liveness* and *opportunism* for a mixed-batch kitting problem.

The main contribution of this paper is that it establishes the formal groundwork for reasoning about plans described as concurrent, communciating process networks — a level of detail simply not possible before. The nearest formal framework is that of Petri-nets which has been used extensively in manufacturing and modeling applications [29]. There is an elegant body of formal theory in place for Petri-nets, some of which has already been applied to robotic and manufacturing problems [5, 16]. There are two reasons why we followed a process-based rather than a Petri-net based approach in this paper: Firstly, the process-based representation has a freer and more programming-like style than static sized graphs of nodes passing control tokens to each other. Secondly, strictly speaking, Petri-nets are a less powerful computational model than process models (i.e. the inability to test for a zero marking in a place)[29]. They gain their formal power from this simplification. Extensions can be made to Petri-nets to make them equivalent in power to process models, e.g. timing, inhibitor arcs, and priorities. However, these result in them losing their formal power.

Future work in this model involves addressing the following problems: The error detection and recovery formula discussed briefly in Section 8 turns out to be very useful but is not supported by the `RS-L3` implementation. We are, therefore, now investigating the 'process instance naming' approach to idempotance. The limit theorems introduced in this paper offer a way to calculate even infinite limit sets. Rather than simply enhancing the $poss^*$ algorithm to make use of limit theorems, we are currently determining if it is possible to build a version using only limit theorems. This involves determining first if there is a fundamental set of limit theorems and then how individual limit theorem results can be combined. The Snet and *scnset* algorithms introduced in this paper have been tested for simple problems such as the mixed-batch kitting problem; however, their completeness and computational complexity is still being investigated. In conclusion, this paper has laid a groundwork, both in theory and implementation, for the representation and analysis of detailed, flexible robot plans; however, further work is still necessary to refine and evaluate the ability of the representation to support automated reasoning.

Acknowledgements.

I would like to thank the following people for the insightful feedback they gave me on this paper: Teun Hendriks, Paul Kallis, Drew McDermott, Danny Stimler, Tom Murphy, Hsiang-Lung Wu, and the anonymous Transactions reviewers.

References

[1] M.A. Arbib, A.J. Kfoury, and R.N. Moll. *A Basis for Theoretical Computer Science*. Springer-Verlag, New York, Heidelberg, Berlin, 1981.

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

[3] D. Chapman and P. Agre. Abstract reasoning as emergent from concrete activity. In *Workshop on Planning& Reasoning about Action*, pages 251–266, Timberline OR, 1986.

[4] B. Donald. A geometric approach to error detection and recovery for robot motion planning with uncertainty. *Artificial Intelligence*, 37(1–3):223–271, Dec. 1988.

[5] M. Drummond. A representation of action and belief for automatic planning systems. In *Workshop on Planning& Reasoning about Action*, pages 267–289, Timberline OR, 1986.

[6] R.E. Ellis. Geometric uncertanties in polyhedral object recognition. *IEEE Trans. on Rob. & Aut.*, 7(3):361–371, 1991.

[7] R.E. Fikes and N.J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[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] B.R. Fox and K.G. Kempf. Opportunistic scheduling for robotic assembly. In *IEEE Int. Conf. Robotics & Automation*, pages 880–889, St.Louis MO, 1985.

[10] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[11] M. Heymann. Concurrency and discrete event control. *IEEE Cont. Sys.*, pages 103–112, Jun. 1990.

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

[13] L.S. Homem de Mello and A.C. Sanderson. And/or graph representation of assembly plans. *IEEE Trans. Rob. & Aut.*, 6(2), Apr. 1990.

[14] S.A. Hutchinson and A.C. Kak. Spar: A planner that satisfies operational and geometric goals in uncertain environments. *AI Magazine*, pages 31–61, Spring 1990.

[15] K. Inan and P. Varaiya. Algebras of discrete event models. *Proceedings of the IEEE*, 77(1):24–38, Jan. 1989.

[16] B.H. Krogh and R.S. Sreenivas. Essentially decision free petri nets for real-time resource allocation. In *IEEE Int. Conf. Robotics & Automation*, pages 1005–1011, 1987.

[17] D.M. Lyons. A Process-Based Approach to Task-Plan Representation. In *IEEE Int. Conf. Robotics & Automation*, Cincinatti, Ohio, May 1990.

[18] D.M. Lyons and M.A. Arbib. A task-level model of distributed computation for sensory-based control of complex robot systems. In *IFAC Symposium on Robot Control*, Barcelona, Spain, Nov. 6-8th 1985.

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

[20] D.M. Lyons and A.J. Hendriks. Planning and acting in real-time. In *IMACS World Congress on Computation and Applied Math.*, page 1387, University of Dublin, Dublin Ireland, 1991.

[21] D.M. Lyons and A.J. Hendriks. Planning for reactive robot behavior. In *IEEE Int. Conf. Rob. & Aut.*, Apr. 7–12th 1992.

[22] D.M. Lyons and A.J. Hendriks. Reactive planning. In S.Shapiro Editor in chief, editor, *Encyclopedia of Artificial Intelligence (2nd Edition).* John Wiley and Sons, 1992. Also: Philips TR-91-016 (public).

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

[24] D.M. Lyons and I. Mandhyan. Fundamentals of $\mathcal{RS}$ - part II: Process composition. Technical Report TR-89-033, Philips Laboratories, 345 Scarborough Rd., Briarcliff Manor, NY 10510, Jun. 1989.

[25] D.M. Lyons and R.N. Pelavin. An analysis of robot task plans using a logic with temporal intervals. Technical Note TN-88-160, Philips Laboratories, 345 Scarborough Rd., Briarcliff Manor, NY 10510, Nov. 1988.

[26] D. McDermott. Planning reactive behavior: A progress report. In *DARPA Workshop on innovative approaches to Planning, Scheduling and Control*, San Diego, 1990. Morgan Kaufman.

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

[28] 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.

[29] J.L. Peterson. *Petri-Net Theory and the Modelling of Systems.* Prentice-Hall, 1981.

[30] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM J. Cont. & Opt.*, 25(1):206–230, Jan. 1987.

[31] S.J. Rosenschein and L.P. Kaelbling. The synthesis of digital machines with provable epistemic properties. Technical Note 412, SRI International, Menlo Park, CA, April 1987.

[32] E.D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier, new York, 1977.

[33] J. Sanborn and J. Hendler. A model of reaction for planning in dynamic environments. *AI in Engineering*, 3(2):95–102, 1988.

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

[35] Austin Tate. Generating project networks. *IJCAI-5*, page 888 to 893, 1977.

[36] D.E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22(3):269–301, 1984.

[37] X. Xiaodong and G.A. Bekey. Sroma: An adaptive scheduler for robotic assembly systems. In *IEEE Int. Conf. Robotics & Automation*, pages 1282–1287, Philadelphia, PA, 1988.

A Algebraic Properties

Strictly speaking, processes are only equal when they both map to the same formal automaton [24]. Equality can be made a little easier than that however. First of all, process are equal if they are defined so, e.g. $\texttt{Plan} = \texttt{Locate}_x\langle p\rangle : \texttt{Place}_p$. Another way to find out if processes are equal is to discover process *identities* that can be applied to composition expressions. Such identities can be constructed by investigating the algebraic properties of the composition operations.

The following table summarizes the most important properties of the composition operators. Communication is omitted from concurrent composition for technical convenience. Notice that since no composition operators have inverses, none will form groups. The strongest structure formed is a commutative monoid[7] (network and disabling operations). The rest are monoids. The disabling operation has an interesting 'subsumption' property when used in conjunction with concurrent composition.

Property	Concurrent	Cond./Seq.	Disabl.
Closed	By definition.	By definition.	By definition.
Associative	(A | (B | C)) = ((A | B) | C)	A:(B:C)=(A:B):C	A#(B#C)=(A#B)#C
Commutative	(A | B)=(B | A)	A:B≠B:A	A#B=B#A
Idempotent	(A | A)= A	A:A≠A	A#A=A
Identity	(A | ABORT)= A =(ABORT | A)	A:STOP=A=STOP:A	INF#A=A=A#INF
Zero	None	ABORT:A=ABORT≠A:ABORT	ABORT#A=ABORT=A#ABORT
Subsumes	—	—	(A#B | B#C) = A#B#C

It is also important to understand how the operations interact with each other. One important such interaction is distribution. The following table captures what operations distribute over other operations (the row distributes over the column).

Left-Distributes	Concurrent	Cond./Seq.	Disabl.
Concurrent	—	—	—
Cond./ Seq.	A : (B | C) = (A : B | A : C)	—	A : (B#C) = (A : B)#(A : C)
Disabl.	—	—	—
Sync/Async. Recurr.	—	—	—

Note that the non-commutative monoids left-distribute over the commutative monoids. This means that Cond./Seq. compositions form semi-rings with network and disabling composition[8].

[7] A monoid is a set operation which is closed, associative and has an identity element.

[8] A semi-ring is a pair in which the first element is a commutative monoid and the second is a (usually non-commutative) monoid that distributes over the first. Semi-rings occur often in computational mathematics [1].

B Definition of One-Step Process Evolution

The following defintions capture one-step evolution. All the processes below are basic.

- $\mathtt{P} \xrightarrow{\Omega\mathtt{P}} \mathtt{STOP}$ (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:
  - $(\mathtt{P} \mid \mathtt{Q}) \xrightarrow{cp\wedge cq} (\mathtt{P}' \mid \mathtt{Q}')\ iff\ \mathtt{P} \xrightarrow{cp} \mathtt{P}'\ and\ \mathtt{Q} \xrightarrow{cq} \mathtt{Q}'$
  - $\Big|_{i\in I} \mathtt{P}i \xrightarrow{c} \mathtt{ABORT}$ 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:
  - $(\mathtt{P}\#\mathtt{Q}) \xrightarrow{cp\wedge cq} (\mathtt{P}'\#\mathtt{Q}')\ iff\ \mathtt{P} \xrightarrow{cp} \mathtt{P}'\ and\ \mathtt{Q} \xrightarrow{cq} \mathtt{Q}'$
  - $\#_{i\in I}\mathtt{P}i \xrightarrow{c} \mathtt{ABORT}$ 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 (we alluded to this fact in Section 3) 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 $\mathtt{P1}, \mathtt{P2}, ...,$

$$\bar{\mho}\mathtt{A} = \mho\mathtt{A} \vee (\bigvee_{\mathtt{P}i\neq A} (\mho\mathtt{P} \vee \Omega\mathtt{P})) \tag{25}$$

C Periodic Process Definitions

Informally, a periodic process is one that 'repeats itself'. The simplest definition of such a process is the standard guarded recursion $\mathtt{T} = \mathtt{X} : \mathtt{T}$. This process executes one instance of $\mathtt{X}$ after another, until $\mathtt{X}$ terminates unsuccessfully (aborts), if it ever does. In this sense, $\mathtt{X}$ is the 'period' of the process. However, since each instance might take a different time and could exhibit different behavior, it is better to avoid the word 'period'; instead, this will be referred to as the *cycle* of the process. In practice, there are a number of useful varieties of periodicity. We will define four below, depending on whether a process is entirely repetitive or whether just some parts of it are repetitive and on whether the process ever terminates.

Periodic. We define a periodic process as one that can be written $\mathtt{T} = \mathtt{X} : \mathtt{T}$. The following test uses evolution to determine if a process is periodic or not:

$$
\begin{array}{lrl}
recurs(\mathtt{P},\mathtt{Q}) \iff & \forall & (c,\mathtt{R}) \in \ poss(\mathtt{P}) \ either \\
& i) & \mathtt{R} \in \{\mathtt{Q},\mathtt{STOP},\mathtt{ABORT}\} \ or \\
& ii) & recurs(\mathtt{R},\mathtt{Q})
\end{array}
$$

$$
periodic(\mathtt{P}) \iff recurs(\mathtt{P},\mathtt{P})
$$

The intuition here is that a process should be periodic if it 'repeats itself.' Part one of the definition catches the repetition (or a termination) when it occurs. Part two recursively scans all evolutions of the process. This branch is taken as long as we are within the cycle of the process. The definition of periodic demands that the original process $\mathtt{Q}$ should appear again somewhere in *all* of its evolutions. It can be easily verified that $\mathtt{T} = \mathtt{X} : \mathtt{T}$ obeys this definition. In the other direction, if a process obeys this definition, then its cycle can be reconstructed by tracing it as it proceeds through the second branch of the test.

We can split periodic processes up into two kinds: ones that never terminate, *infinite periodic*; and ones that may terminate after some finite number of periods, *finite periodic*. An infinite periodic process is one that obeys a stronger version of constraint $i)$ in the definition of $recurs$ above: $\mathtt{R} = \mathtt{Q}$. A simple example is the unguarded recursion process $\mathtt{T} = \mathtt{X};\mathtt{T}$. A finite periodic process is a periodic process which does not obey this stronger definition. An infinite periodic process has an empty limitset (since it never terminates). A finite periodic process has a non-empty limitset.

Semi-Periodic. A second useful definition to capture is when the original process appears in *some* but not all of its own evolutions. That is, guarded recursions of the form $\mathtt{T} = \mathtt{X} : (\mathtt{B} \mid \mathtt{T})$ where $\mathtt{X}$ is again the cycle of the process and the process $\mathtt{B}$ isn't defined in terms of $\mathtt{T}$. To build a test for semi-periodic we need to define the concept of a *subnet* of a network. In a concurrent network $N = \Big|_{i\in I} \mathtt{P}^i$, any network $S = \Big|_{i\in IS} \mathtt{P}^i$ for IS a proper subset of I, is considered a subnet of N. (This excludes $IS = \emptyset$ and $IS = I$.)

$$
\begin{array}{lrl}
semirecurs(\mathtt{P},\mathtt{Q}) \iff & \forall & (c,\mathtt{R}) \in \ poss(\mathtt{P}), \ either \\
& i) & \mathtt{R} \in \{\mathtt{Q},\mathtt{STOP},\mathtt{ABORT}\}, \ or \\
& ii) & For\ \mathtt{N}\ a\ subnet\ of\ \mathtt{R},\ either \\
& & \quad a)\ \ \mathtt{N} = \mathtt{Q} \\
& & \quad b)\ \ semirecurs(\mathtt{N},\mathtt{Q})
\end{array}
$$

$$
semiperiodic(\mathtt{P}) \iff semirecurs(\mathtt{P},\mathtt{P})
$$

Part $ii)$ of this definition only asks that *some* subnet of an evolution repeat; other parts of the subnet can be non-repetitive. We can again split this category into two: infinite and finite semi-periodic, based on whether a process obeys the stronger version of constraint *i*). A semi-periodic process will typically have a nonempty limitset because of contributions from its non repetitive components.

It is possible for a process to be non-periodic and still have no limits, we call such a process *chaotic*. If the periodic and semi-periodic tests given above are applied to a chaotic process, then they will not terminate.

D Process Limit Theorems

Theorem 1 *If* B *is a bound then limitset*$(\mathtt{A} : \mathtt{B})$ *is of the form* $\{(\Omega\mathtt{A},\mathtt{B}),(\mho\mathtt{A},\mathtt{ABORT})\}$.

Proof: By the definition of conditional composition, $poss(\mathtt{A} : \mathtt{B}) = \{(\Omega\mathtt{A},\mathtt{B}),(\mho\mathtt{A},\mathtt{ABORT})\}$. The theorem immediately follows since both ABORT and B are bounds. □

Theorem 2 *If* P *is a bound, then limitset*$((\mathtt{A}\#\mathtt{B}) : \mathtt{P})$ *is of the form* $\{(\mho\mathtt{A}\wedge\mho\mathtt{B},\mathtt{ABORT}),(\Omega\mathtt{A}\vee\Omega\mathtt{B},\mathtt{P})\}$.

Proof: By theorem 1, *limitset*$((\mathtt{A}\#\mathtt{B}) : \mathtt{P})$ is of the form $\{(\Omega(\mathtt{A}\#\mathtt{B}),\mathtt{P}),(\mho(\mathtt{A}\#\mathtt{B}),\mathtt{ABORT})\}$. The abort and stop conditions of A#B come directly from their definition as $\mho\mathtt{A}\wedge\mho\mathtt{B}$ and $\Omega\mathtt{A}\vee\Omega\mathtt{B}$ respectively. □

Theorem 3 *If* X *and* Y *are bounds, then limitset*$((\mathtt{A}\#\mathtt{B} \mid \mathtt{A} : \mathtt{X} \mid \mathtt{B} : \mathtt{Y})) = \{(\Omega\mathtt{A},\mathtt{X}),\ (\Omega\mathtt{B},\mathtt{Y}),\ (\mho\mathtt{A}\wedge\mho\mathtt{B},\mathtt{ABORT})\}$.

Proof: This can be shown using the above two theorems and an enumeration of the cases for *poss*. The only difficulty lies in the case where both A and B stop simultaneously. One might expect the network $(\mathtt{X} \mid \mathtt{Y})$ to happen in this case. However, the definition of # was written to specifically forbid this option: the options for # are to have all processes abort, or to have exactly one stop and the rest abort. The result in this case is to generate nondeterminism as to which process will abort, and so as to whether X or Y will be produced. □

Theorem 4 *If* B *is a bound,* A::B *is semi-periodic. It is infinite-semi-periodic if* $\mho\mathtt{A}$ *does not exist; finite-semi-periodic otherwise.*

Proof: Expanding A::B we get $\mathtt{T} = \mathtt{A} :(\mathtt{B} \mid \mathtt{T})$. Let us apply *semirecurs*$(\mathtt{T},\mathtt{T})$. *poss*$(\mathtt{T})$ contains only $(\mho\mathtt{A},\mathtt{ABORT})$ and $(\Omega\mathtt{A},(\mathtt{B} \mid \mathtt{T}))$. Let $\mathtt{R1} = \mathtt{ABORT}$ and $\mathtt{R2} =(\mathtt{B} \mid \mathtt{T})$. Part *i*) catches R1 . The subnets of R2 are B and T by definition. B does not obey part *ii.a* or *ii.b* of *semirecurs*. T however obeys *ii.a*. Since the definition only requires that one subnet obey part *ii*) (i.e. that there be one subnet that repeats), this demonstrates that A::B is semi-periodic.

To determine if A::B is infinite, we need to see if it obeys *semirecurs* with the stronger version of part *i*). Applying *semirecurs*$(\mathtt{T},\mathtt{T})$ again we note that if $\mho\mathtt{A}$ does not exist, then only $(\Omega\mathtt{A},\mathtt{B} \mid \mathtt{T})$ is in *poss*$(\mathtt{T})$. In that case, only part *ii*) of the definition is needed to demonstrate that A::B is semi-periodic. Therefore, if $\mho\mathtt{A}$ does not exist, then A::B is infinite semi-periodic. If $\mho\mathtt{A}$ does exist, the $(\mho\mathtt{A},\mathtt{ABORT})$ is also a member of *poss*$(\mathtt{T})$ and this causes the stronger version of part *i*) to fail. In that case, A::B is finite semi-periodic. □

Theorem 5 *If* B *is a bound then limitset*$(\mathtt{A} :: \mathtt{B})$ *has elements of the form* $(c,\mathtt{B})$*, where* $c = \Omega\mathtt{A}^n, n \geq 1$.

Proof: The previous theorem demonstrates that A::B is semi-periodic if B is a bound. The non-periodic branch is the only way elements can be added to the *limitset*. Expanding `A::B` as before, we get $\mathtt{T} = \mathtt{A} : (\mathtt{B} \mid \mathtt{T})$. The non-periodic branch is `B`, and is produced once for each recursion. The condition for each recursion is simply $\Omega\mathtt{A}$. Thus *limitset*(T) will contain couples of the form $(\Omega\mathtt{A}, \mathtt{B}), (\Omega\mathtt{A}.\Omega\mathtt{A}, \mathtt{B}), \ldots$.

Note: if `T` is finite semi-periodic, then the *limitset* will additionally contain couples of the form $(\Omega\mathtt{A}^n.\mho\mathtt{A}, \mathtt{ABORT})$ for $n \geq 0$. □

Theorem 6 `A:;B` *is periodic. It is finite-periodic if* $\mho\mathtt{A}$ *exists, and the limitset is of the form* $\{(c, \mathtt{ABORT}) \mid c = \Omega\mathtt{A}^n.\mho\mathtt{A}, n \geq 0\}$.

Proof: Expanding `T=A:;B` we get `T = A:(B;T).` It is straightforward to verify *periodic*(T).

Let us assume $\mho\mathtt{A}$ exists. Then *poss*(T) has two elements $(\mho\mathtt{A}, \mathtt{STOP})$ and $(\Omega\mathtt{A}, \mathtt{B;T})$. The `B;T` element satisfies part *ii*) of both the infinite-periodic and periodic definitions. The `STOP` only satisfies part *i*) of periodic; it will not satisfy the stronger part *i*) constraint for infinite-periodic.

The only way that elements can be added to the *limitset* is via the $(\mho\mathtt{A}, \mathtt{STOP})$ element of *poss*(T). If this happens the first time through the definition of *periodic*, then $(\mho\mathtt{A}, \mathtt{STOP})$ is added to the *limitset*. If it happens the second time (i.e., the on the first recursion through part *ii*) then $(\Omega\mathtt{A}.\mho\mathtt{A}, \mathtt{STOP})$ is added; the third time, $(\Omega\mathtt{A}.\Omega\mathtt{A}.\mho\mathtt{A}, \mathtt{STOP})$, etc. □

Theorem 7 *If* `P` *is infinite-periodic (infinite semi-periodic) and* $\Omega\mathtt{I}$ *exists, then* `T = I#P` *is finite-periodic (finite semi-periodic).* `I#P` *has a limitset of* $limitset(\mathtt{P}) \cup \{(\Omega\mathtt{I}, \mathtt{ABORT})\}$. *If* $\mho\mathtt{I}$ *exists, then limitset(* `I#P` *) additionally has the couple* $(\mho\mathtt{I}, \mathtt{ABORT})$.

Proof: The *poss* of `T=I#P` contains three options: either `I` aborts, `I` stops, or `P` starts its periodic (semi-periodic) recursion. This is directly the definition of finite periodic (finite semi-periodic). The only extension `I` makes to *limitset*(P) is to add its termination conditions. □

E Detailed Evolution of the Placing Network

In the text we make the following assertion without proof:

$$(\mathtt{Place}_x \mid \mathtt{Placelaw}) \overset{time(tp)}{\longrightarrow} (\mathtt{Placed}_x \mid \mathtt{Placelaw}) \tag{26}$$

Here is the detailed proof which uses a straightforward application of the algebraic definitions of Appendix A and the evolution definitions of Appendix B:

$$\begin{array}{rcl} (\mathtt{Place}_x \mid \mathtt{Placelaw}) & = & (\mathtt{Place}_x \mid \mathtt{Exists}_{\mathtt{Place}_r}\langle p\rangle :: (\mathtt{Delay}_{tp}; \mathtt{Placed}_p)) \\ & \longrightarrow & (\mathtt{Place}_x \mid (\mathtt{Delay}_{tp}; \mathtt{Placed}_x) \mid \mathtt{Placelaw}) \\ & = & (\mathtt{Delay}_{tp} \mid (\mathtt{Delay}_{tp}; \mathtt{Placed}_x) \mid \mathtt{Placelaw}) \\ & \overset{time(tp)}{\longrightarrow} & (\mathtt{Placed}_x \mid \mathtt{Placelaw}) \end{array} \tag{27}$$

If the plan and world processes are defined as in the main text, then the complete detail of the placement of the first part (the tray) can be deduced as follows. Let us consider the first part of the plan and the salient part of the world as

$$\begin{array}{lcl} \texttt{Plan} & = & \texttt{Locate}_t\langle p\rangle : \texttt{Place}_p ; \texttt{QP} \\ \texttt{World} & = & (\texttt{Winit} \mid \texttt{Placelaw}) \\ \texttt{Winit} & = & (\texttt{TERM} :: \texttt{Part}_t \mid \texttt{QW}) \end{array} \tag{28}$$

where `QP` and `QW` abbreviate the rest of the plan and world, for the purposes of this exposition. While this is correct for `QP` it is not generally correct for `QW`, since the world can change asynchronously as the execution of the plan proceeds. However, in this example, because the only way the world can affect the first placement in the plan is via the occurrence of the tray, the simplificiation does hold. Using the definition of evolves and (15),(27) from the world model definition, we can trace the following chain of evolutions:

$$\begin{array}{lcll} \texttt{System} & = & (\ \texttt{Plan} \mid \texttt{World}\ ) & \\ & = & (\ \texttt{Plan} \mid \texttt{Winit} \mid \texttt{Placelaw}\ ) & \\ & = & (\ \texttt{Locate}_t\langle p\rangle : \texttt{Place}_p ; \texttt{QP} \mid & \\ & & \ \texttt{TERM} :: \texttt{Part}_t \mid \texttt{QW} \mid & \\ & & \ \texttt{Placelaw}\ ) & \\ & \xrightarrow{time(t1)} & (\ \texttt{Locate}_t\langle p\rangle : \texttt{Place}_p ; \texttt{QP} \mid & By\ (15) \\ & & \ \texttt{Part}_t \mid \texttt{QW} \mid & \\ & & \ \texttt{Placelaw}\ ) & \\ & \longrightarrow & (\ \texttt{Place}_t ; \texttt{QP} \mid & By\ (17) \\ & & \ \texttt{Part}_t \mid \texttt{QW} \mid & \\ & & \ \texttt{Placelaw}\ ) & \\ & \xrightarrow{time(tp)} & (\ \texttt{QP} \mid & By\ (20) \\ & & \texttt{Part}_t \mid \texttt{QW} \mid & \\ & & \texttt{Placed}_t \mid \texttt{Placelaw}\ ) & \end{array} \tag{29}$$

F Snet Segmentation Algorithm

The following algorithm will cut a network, phrased as a string of processes and composition operators, into a set of sensory motor sub-network segments. The algorithm handles communication links. `Exists` links can be handled in the same way, but are omitted here for clarity. The algorithm expects the motor process to be the rightmost element of any network or disabling composition[9].

We make use of the following variables and functions: CurrentProcess is the process in the network currently under consideration by the algorithm. NextProcess and NextCompOp refer to the next process and composition operator to the right of the current process in the network. Variables(), Results() and Ports() are functions which produce the set of parameter values, the set of port names or the set of result names of a process. UnresPar will be the set of parameters whose dependance has not yet been resolved, and UnresCom will be the set of communication ports whose dependance has not yet been resolved. For each motor process, the algorithm steps through the network starting at the motor process and terminating when all processes which are the sources of variable and port references necessary to parameterize that motor process have been included in the Snet sub-network for that motor process.

For each motor process i **Do**

[9]Operating on a parse tree rather than a string would remove this restriction.

```
        CurrentProcess ← motor process i
        Snet_i ← CurrentProcess
        UnresPar ← Variables(CurrentProcess)
        If CurrentProcess ∈ { IN, OUT }
                Then UnresCom←UnresCom+Ports(CurrentProcess)
        While UnresPar and UnresCom not empty Do
                Case NextCompOp Of
                # or ;
                    CurrentProcess ← NextProcess
                    include NextProcess in Snet_i
                    UnresPar ← UnresPar+Variables(CurrentProcess)
                :; or :: or :
                    CurrentProcess ← NextProcess
                    include NextProcess in Snet_i
                    UnresPar ← UnresPar+Variables(CurrentProcess)-Results(CurrentProcess)
                 | (with port to port communication map c)
                    If UnresCom is mapped by c onto Ports(NextProcess)
                            Then include NextProcess in Snet_i
                            UnresCom ← UnresCom - c(Ports(NextProcess))
                     Else skip over NextProcess
                End Case
                If CurrentProcess ∈ { IN, OUT }
                    Then UnresCom←UnresCom+Ports(CurrentProcess)
        End While
End For
```

Figure Captions:

Table Captions:

Footnotes:

1. Indeed, Heyman [11] has a DEC approach which is initially very similar to the approach we will demonstrate here.

2. This is necessary to include nondeterministic processes, e.g., network (6) in Section 5.

3. See Section 8 for a note on our implementation of this; Section 7 will present the formalization of the effects of this process.

4. Since `TERM` is nondeterminstic, we need to use superscripts to distinguish the multiple identical copies in this example.

5. A process can also be forced to abort, but we will see how to include those conditions later.

6. We could assign a unique name to each instance of each of the part models which arrive; however, this is not necessary and complicates the analysis.

7. A monoid is a set operation which is closed, associative and has an identity element.

8. A semi-ring is a pair in which the first element is a commutative monoid and the second is a (usually non-commutative) monoid that distributes over the first. Semi-rings occur often in computational mathematics [1].

9. Operating on a parse tree rather than a string would remove this restriction.