Report Number: WUCS-96-31

1996-01-01

Building Distributed Applications with Design Patterns

Gruia-Catalin Roman and James C. Hu

Design patterns are a topic of great current interest within the object-oriented programming community. The motivation is both economical and intellectual. On one hand, there is the hope of establishing a common culture and language that fosters communicatino and growth in the software engineering field. While a community dominated by empiricism is seeking to achieve higher levels of formality by capturing its experiences in the form of catalogs of design patterns, another community, deeply rooted in formal thinking, is seeking to make its mark on the every day workings of the software engineering process. Distributed algorithms and the heuristics...

Read complete abstract on page 2.

Building Distributed Applications with Design Patterns

Gruia-Catalin Roman and James C. Hu

Complete Abstract:

Design patterns are a topic of great current interest within the object-oriented programming community. The motivation is both economical and intellectual. On one hand, there is the hope of establishing a common culture and language that fosters communicatino and growth in the software engineering field. While a community dominated by empiricism is seeking to achieve higher levels of formality by capturing its experiences in the form of catalogs of design patterns, another community, deeply rooted in formal thinking, is seeking to make its mark on the every day workings of the software engineering process. Distributed algorithms and the heuristics used by program derivation methods represent a large repository of fundamental knowledge that has been acquired over the years by the distributed computing community. Attempts to make this body of knowledge available to the broader community have been frustrated to say the least. The main thesis of this paper is that design patterns are a viable mechanism by which distributed computing know-how can impact the practical development of dependability-minded distributed applications. We contend, however, that in order to do so one needs to view design patterns in a new perspective, more formal and more language-independent than the view adopted by the object-oriented programming community. This paper provides a possible characterization of the notion of a distributed design pattern and discusses ways by which design patterns can be transformed into specific applications. An airport baggage delivery system with distributed control is used for illustration purposes.

Building Distributed Applications with Design Patterns

Gruia-Catalin Roman and James C. Hu

WUCS-96-31

December 1996

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

Building Distributed Applications with Design Patterns

Gruia-Catalin Roman
James C. Hu[*]

December 19, 1996

WUCS-96-31

Abstract

Design patterns are a topic of great current interest within the object-oriented programming community. The motivation is both economical and intellectual. On one hand, there is a need to leverage off past experiences on new projects. On the other hand, there is the hope of establishing a common culture and language that fosters communication and growth in the software engineering field. While a community dominated by empiricism is seeking to achieve higher levels of formality by capturing its experiences in the form of catalogs of design patterns, another community, deeply rooted in formal thinking, is seeking to make its mark on the every day workings of the software engineering process. Distributed algorithms and the heuristics used by program derivation methods represent a large repository of fundamental knowledge that has been acquired over the years by the distributed computing community. Attempts to make this body of knowledge available to the broader community have been frustrating to say the least. The main thesis of this paper is that design patterns are a viable mechanism by which distributed computing know-how can impact the practical development of dependability-minded distributed applications. We contend, however, that in order to do so one needs to view design patterns in a new perspective, more formal and more language-independent than the view adopted by the object-oriented programming community. The paper provides a possible characterization of the notion of a distributed design pattern and discusses ways by which design patterns can be transformed into specific applications. An airport baggage delivery system with distributed control is used for illustration purposes.

[*]This paper is based upon work supported in part by Siemens AG. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of Siemens.

1 Introduction

The development of distributed systems continues to present the software engineering profession with exceptional challenges. Large projects frequently fail due to misconceptions regarding their requirements and due to the intrinsic complexity exhibited by the underlying distributed architecture. Even small projects often exceed estimated costs, fail to meet schedules, and achieve only limited levels of dependability. Dependability, defined as the guarantee that software errors are absent, is a very difficult goal to achieve. It requires a high degree of discipline, considerable expertise, and specialized skills. Some of the most important work relating to software dependability involves the development of formal verification techniques, logic verifiers, and model checkers. All these approaches assume the existence of a program and focus on ways to establish formally whether or not it possesses certain important properties. Only relatively small programs can be analyzed in this manner. Nevertheless, it is not in the least unreasonable to assume the existence of verified components and ask the question how one might be able to offer similar guarantees for larger systems.

This is essentially an engineering question. Dependability needs to be built-in during design and implementation, i.e., not merely checked at the end of the process. The idea is not new. Design rules and structured forms have been promoted for their ability to simplify verification. This very notion was central to the push for wide spread use of structured programming and block-structured languages[5]. Program derivation techniques have been the result of a philosophy which contends that one can deliver programs that are correct by construction through a process of gradual refinement. The price one must pay involves finding the right transformation (a highly creative step) and showing that it is correctness preserving (ideally a merely mechanical step)[8, 4]. Another strategy is to build the system out of components and, after each step, to compute the properties of the composite from those of the components. Process algebras follow this particular strategy[7].

We too are interested in an engineering approach to the development of distributed systems. Our investigation is centered on application-specific systems of small to medium size. The approach we explore in this paper is akin to program derivation. Several forms of program derivation have been used successfully in both academic exercises and even in industrial-grade systems[12]. Specification refinement[2] starts with an abstract formulation of the desired properties of the target program, usually expressed in logic, and gradually transforms it into an increasingly more concrete representation up to the point that writing a correct program becomes a trivial exercise. Program refinement[1], on the other hand, uses a simple but correct program as the starting point for a serious of transformations that change it into another program which meets the same correctness criteria but it is more amenable to efficient implementation. Examples of mixed specification and program refinements can also be found in the literature[10, 11].

The novel elements of the program derivation strategy being explored in this paper are the use of design patterns and plastic transformations. The design patterns are abstract distributed algorithms which have been shown to be essential to solutions for important distributed computing problems and whose correctness has long been established. Plastic transformations are program transformations which preserve the essential features of the starting program but add additional details meant to fill the gap between the program as is and the requirements of the target application. Many of these transformations are borrowed from the established program derivation literature.

The similarity to program refinement is not accidental. The goal of our work has not been one of contributing to the formal foundation of program derivation but more one of technology transfer. We sought to revisit program derivation from an engineering perspective. This paper is an attempt to blend the development patterns that can be observed in traditional engineering organizations with the intellectual tools offered by developments in program derivation. A central element of our strategy is the selection of an initial design pattern which represents a dependable skeleton around which we mold the application details. In contrast to traditional program refinement, the starting point is not an abstract and correct solution to the problem but a pattern of behavior the designer expects to see in the ultimate solution. Furthermore, the successive transformations are not focused on replacing one computing mechanism by another but rather on attaching additional pieces without interfering with the essential behavior of the original program. In the long run, we hope to define a process that leads to dependable and cost-effective distributed systems design. Using proven (or easy-to-prove) algorithms and a highly disciplined process, we try to leverage the strengths of the distributed computing field.

The remainder of the paper is organized as follows. Section 2 defines the concept of design pattern for use in the distributed computing realm and contrasts our definition with the one recognized by the object-oriented programming literature. Section 3 is concerned with the process by which patterns are molded into applications. The approach is illustrated on a simplified, but realistic, airport baggage delivery application in which carts carrying bags are routed to their destinations. Section 4 discusses the background of this research and identifies several open questions relating to the design process outlined earlier. Critical among them is the ability to allow industrial designers to apply the technique with minimal or no verification effort. Conclusions are presented in Section 5.

2 Design Patterns

Design pattern, as a technical software engineering term, has its origins in the object-oriented programming literature[6, 3]. In this paper we attempt to apply the basic concept to the realities of distributed computing. As used in the object-oriented programming context, a design pattern (henceforth called *OOP pattern*) is a "*description of communicating objects and classes that are customized to solve a general design problem in a particular context*"[6]. Each pattern is characterized by a name, the problem it solves, the solution it embodies, and the consequences it has on the design, e.g., performance, extensibility, etc. The name and the problem description are needed in order to facilitate the development of a catalog of design patterns and simple communication regarding the patterns at design time. An *iterator* pattern, for instance, provides a sequential access mechanism for the elements of an aggregate object without exposing its internal structure. The solution is generally captured in an abstract template-like form independent of any realization. Possible design consequences (e.g., performance) are important in helping the designer understand the trade-offs involved in the use of the particular pattern. This basic descriptive structure can be refined by considering a richer set of attributes (e.g., known uses, sample code, related patterns, etc.) and by offering a taxonomy of the established design patterns (e.g., creational, structural, and behavioral).

2.1 Issues

The theme underlying the OOP patterns work is the reusability of objects in the context of modern object-oriented programming languages (mostly C++). Introducing the notion of design pattern to distributed systems design (henceforth called *DC patterns*) is more complicated than one might expect. The emphasis has to shift away from components to interactions which are the most complex aspect of a distributed system. This suggests the need to revisit the very concept of a design pattern. Distributed computing patterns must be more abstract, more precise, and more language-independent than their OOP counterparts. Specificity to distributed computing and sensitivity to the system topology are two other key requirements.

2.1.0.1 Abstractness. Properties important in distributed computing applications are global in nature (e.g., sum m available in account p is eventually transferred into account q) and the mechanisms that implement them cut across many processes and interactions. Abstraction is the only way to control the inherent complexity of distributed systems. Abstract solutions are likely to be compact, reusable, understandable and easy to analyze. A higher level of abstraction leads to fewer details and a greater distance between a pattern and the application code constructed from it. The result is an increased need to consider the mechanics of transforming patterns into applications, by contrast with merely "using" existing patterns in new applications.

2.1.0.2 Precision. The description of OOP patterns is usually informal in style and includes no formal guarantees. In distributed computing, dependability relies on formal verification possible only when the description is exact. A design pattern must be operational in nature, i.e., an abstract program, and must include guarantees derived from formal proofs. The proofs, however, need not be of concern to the designer making use of the pattern. The idea is to achieve dependability by exploiting proven components.

2.1.0.3 Language-independence. Language details do not seem to play an important role in the design of distributed systems. While OOP patterns rely heavily on the availability of certain specific language capabilities common among object-oriented programming languages, this should not be the case for DC patterns. This is in concert with the requirement for a more abstract treatment of the solution.

2.1.0.4 Specificity to distribution. The underlying model of computation, however, does affect the correctness, feasibility, and performance characteristics of the resulting system. Shared memory, message passing, remote procedure call, synchrony, atomicity, fairness, failure characteristics, etc., shape fundamental features of the solution space. A designer must be able to tell from the pattern its implications for the target architecture intended to support the desired application. The computational model is an important attribute of a design pattern.

2.1.0.5 Sensitivity to topology. Many solutions to distributed computing problems rely on assumptions about the connectivity among the components. Linear graphs are important in pipeline problems, ring structures lead to specialized solutions, etc.

```
for $k, j$ such that $j = 2, 4, \ldots, 2^p \wedge 1 < k \leq 2^p \wedge k \bmod j = 0$

program Sum($k, j$)
declare
    $x[k,j], x[k, \frac{j}{2}], x[k - \frac{j}{2}, \frac{j}{2}]$ : integer
initially
    $x[k,j]$                        $= \perp$
[]  $x[k, \frac{j}{2}]$             $= A[k]$                 if $j = 2 \sim \perp$ otherwise
[]  $x[k - \frac{j}{2}, \frac{j}{2}]$   $= A[k - \frac{j}{2}]$   if $j = 2 \sim \perp$ otherwise
assign
    $x[k,j] := x[k, \frac{j}{2}] + x[k - \frac{j}{2}, \frac{j}{2}]$ if $x[k, \frac{j}{2}] + x[k - \frac{j}{2}, \frac{j}{2}] \neq \perp$
end
```

Figure 1: A parallel summation algorithm in UNITY.

2.2 Definition

Even though we have been discussing what are some of the important attributes of a design pattern in distributed computing, we have been skirting the central question. What is a design pattern? As in object-oriented programming, the answer is somewhat subjective. The goal of this paper is to explore possible answers to this very question. A good starting point is to treat a design pattern as "*a verified distributed algorithm that solves a sufficiently general and formally stated problem frequently encountered within some broad design context.*" The algorithm, the problem specification, the assumptions regarding the underlying computational model, and the connectivity among processes form the core of the design pattern as it might be entered in a catalog of useful patterns. This definition rules out software architectures, schemas, packages, and computational models. Our perspective is clearly biased in favor of exploiting the vast knowledge that the research community has already acquired in this area and reflects the view that certain algorithms have been studied extensively precisely because they are central to a very broad range of problems. Moreover, correctness proofs, complexity analysis, and impossibility results contribute to an increased level of trust on the side of a practicing designer. This perspective does not rule out the development of pattern catalogs specific to a particular application domain or even product line. Actually, we believe that successful application of design patterns is most likely to take place in specialized software development domains.

While every design pattern has at its core a distributed algorithm, the converse is not true. Algorithms that solve very specific tasks are generally not good candidates for consideration. Take, for instance, an algorithm that does parallel array summation (Figure 1), expressed here using the UNITY notation.

In this algorithm, a constant array A of size 2^p is summed up by a tree of processes with the final sum being left in the array element $x[2^p, 2^p]$ (illustrated in Figure 2). The specific

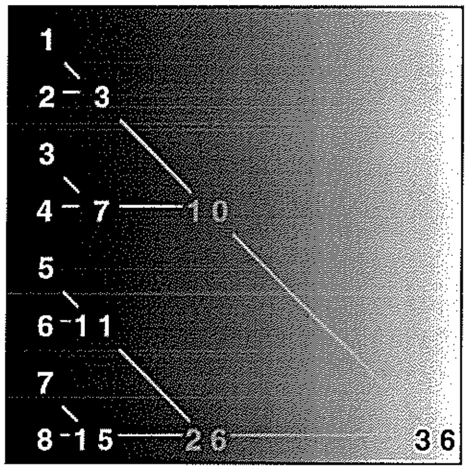

Figure 2: Visualization of the summation program, where A has size 2^3.

use of addition and the reliance on testing for undefined values to control the ordering of operations argue against considering this algorithm as a candidate design pattern. At the same time, tree based processing is general and widely used. This suggests that we may want to consider this program as an instantiation of a more general design pattern such as the one captured by the program shown in Figure 3, where the tree structure is preserved but each process computes its own specific function. The final result is the fixpoint of the process community. The proof that the fixpoint is reached relies on the fact that, once the arguments of the function $F_{k,j}$ become stable, the value in $x[k, j]$ eventually becomes stable as well.

As already indicated, a full description of the pattern is much more involved and a catalog of such patterns is akin to a textbook on distributed algorithms. Actually, many classical distributed algorithms such as leader election, consensus, reliable broadcast, snapshots, etc. are immediate candidates for inclusion in a catalog of general purpose design patterns. It should be noted, however, that a design pattern perspective changes both the manner

```
for $k,j$ such that $j = 2, 4, \ldots, 2^p \wedge 1 < k \leq 2^p \wedge k \bmod j = 0$

program S($k,j$)
declare
    $x[k,j], x[k,\frac{j}{2}], x[k-\frac{j}{2},\frac{j}{2}]$ : same-type-as-$A$
initially
    $x[k,\frac{j}{2}]$              $= A[k]$              if $j = 2$
[]  $x[k-\frac{j}{2},\frac{j}{2}]$  $= A[k-\frac{j}{2}]$  if $j = 2$
assign
    $x[k,j] := F_{k,j}(x[k,\frac{j}{2}], x[k-\frac{j}{2},\frac{j}{2}])$
end
```

Figure 3: An abstract program that is a suitable design pattern candidate.

in which algorithms are formulated and the way they are documented. The focus is no longer on presenting and analyzing the solution to a specific problem but on structuring the algorithm and its properties for reusability. We experienced this first hand recently as part of an investigation into the application of established distributed algorithms to problems in mobile computing[9]. Our ostensible goal was to show how algorithms designed to compute distributed global snapshots provide reasonable solutions for the problem of delivering messages among mobile nodes; we also showed that the results are immediately applicable to the related problems of route discovery and maintenance among mobile agents both in the presence of mobile support stations and in ad-hoc networks. A by product of this study was an abstract description of a class of snapshots algorithms that can be easily transformed into any one of the variants published in the literature. Exercises of this kind will help us develop a better understanding of how to extract and formalize design patterns.

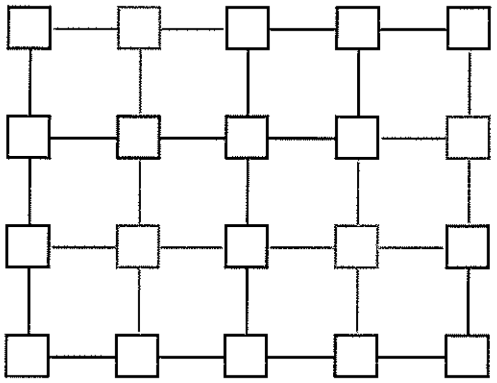

Figure 4: Processors form a grid; dormant processors are shown in a lighter color.

More research is needed to establish standards for the development and documentation of distributed computing design patterns, a process fairly well understood when it comes to OOP patterns. Carrying out this task for many general purpose algorithms may entail an effort comparable to that of developing a dictionary or cataloging the plants growing in the rain forests. A more manageable task, however, might be to build such catalogs for specific application domains such as transportation systems, embedded controllers, power control, etc. The desirability of such an undertaking is reinforced by the frequently made observation that many systems being developed by the same organization are simply variants of older ones. We believe that the task is both manageable and profitable.

2.3 An illustration

Assuming the availability of a catalog of design patterns, the key issues we must consider next are: how to select the right pattern for the task at hand, how to transform the pattern into an application, and how to handle the situation when multiple patterns are needed to construct the application. These are important research questions for which we have only partial answers at this time. They are discussed and illustrated in the next section. We provide here a brief preview by way of a simpler illustration.

Let us consider a rectangular grid of processors, some of which may be active while others may be dormant, and let's assume that at the start of the computation one needs to take a count of the number of active processors (Figure 4). The traditional way of attacking such a problem is to devise an algorithm from scratch and to code it. When working with patterns the starting point is the catalog. We are looking for a pattern useful in computing a function over a set of values distributed across some graph structure which in our problem appears to take the form of a sparse rectangular grid. Our search might be rewarded by finding an entry that might look as follows:

Problem	Assign to each node of a binary tree a value that depends only upon itself and its descendents.
Algorithm	**program** Btree(k) **assign** $x[k] :=$ $F_k(x[\text{RightChild}(k)], x[\text{LeftChild}(k)])$ **if** $\neg$Leaf(k) $\sim$ $F_k()$ **if** Leaf(k) **end**
Specification	**let** $\mathcal{F}(k) \equiv \begin{cases} F_k() & \textbf{if } \text{Leaf}(k) \\ F_k(x[\text{RightChild}(k)], x[\text{LeftChild}(k)]) & \textbf{otherwise} \end{cases}$ **true** $\longmapsto x[k] = \mathcal{F}(k)$ **stable** $x[k] = \mathcal{F}(k)$
Complexity	$O(\log(n))$ time and $O(n)$ communications, where n is the number of nodes in the tree.
Communication	Shared variables.
Topology	Complete binary tree.

Table 1: A binary tree design pattern.

Even though our problem involves a sparse grid, we could build a binary spanning tree (Figure 5), let the leaf nodes compute the constant *one* and for the non-leaf nodes instantiate F_k such that $F_k(a, b) \equiv a + b + 1$. Having made these observations, the next step is to start modifying the algorithm above until it fits the problem at hand. Along the way we will need to consider variable renaming, the data exchange protocol among the processors, the fact that we do not actually have a complete binary tree, and the possibility of using another pattern to construct the spanning tree in the first place. The simple one-line solution above will grow into a much larger program. The key is to make sure that its defining properties are preserved at every step. In the next section we examine this process in a more realistic setting.

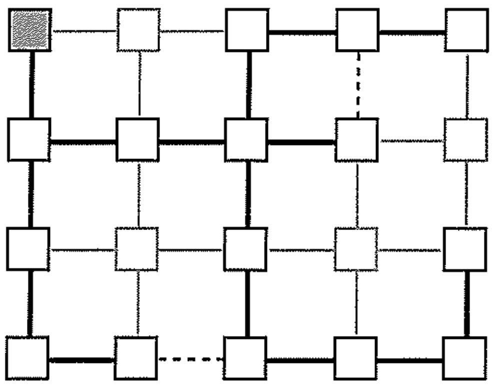

Figure 5: A spanning tree embedding designed to facilitate the use of the tree processing design pattern; the root node is shaded.

3 Sample design process

This section presents a design methodology that relies on the gradual refinement of preexisting design patterns. The approach is illustrated on an application involving automated baggage delivery in an airport setting. The type of design solution being sought out is highly distributed. Each system component is expected to be "smart" and amenable to use in many system configurations without redesign. The design methodology is presented informally in a manner that mimics the way a designer might apply it in an industrial setting. The presentation is divided into three parts: an overview of the application domain, an analysis of the target application leading to the identification of one or more useful design patterns, and the actual derivation of the application code from the initial design patterns.

3.1 Application Overview

A baggage cart system is made up of carts, tracks, and stations. Each cart has an electric motor which can be energized by the underlying track and a platform which carries a single bag. Information about each bag, e.g., destination and identification, appears on a bar-coded label placed on the bag at the time it is loaded. Stations are places where baggage is removed from and placed into baggage carts. For the sake of brevity, we will ignore the mechanics of loading and unloading and focus strictly on the control of the cart movement. Consequently, there is no need to differentiate among tracks that are between stations and those that are located directly at a station. All tracks are unidirectional. Some of the tracks are straight pieces while others represent merge or switch pieces. We generalize these configurations and treat each track as consisting of three parts: an n-way join, a straight piece, and an m-way switch, in order. The controls associated with each "smart" track can set the switch position and can energize the track thus allowing a cart to move forward. Entry and exit sensors are used to determine the presence of carts on individual tracks but the join and switch pieces extend outside the two sensors, i.e., a departure signal is not a guarantee that the cart actually left the track. Only one cart is allowed to be present at a time on any given track. An optional scanner is used to read the baggage label in order to assess its destination. We use track identifiers for bag destinations. An abstract illustration of a cart on a track is shown in Figure 6.

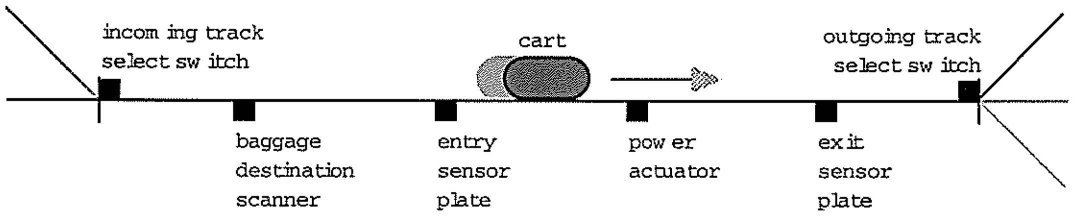

Figure 6: Abstract view of a baggage cart system.

The precise layout of the track is unknown but we assume that every track is reachable from any other track. The number of carts is assumed to be much smaller than the number of tracks in the layout. Our design goal is to build a dependable control system that guarantees that each piece of baggage gets to its destination safely. This requirement translates into ensuring that all carts make progress toward their respective destinations, avoiding track collisions and maintaining accurate information about the identity of the bags in transit.

3.2 Problem Analysis

In this section we concentrate on identifying the design patterns that will form the starting point for the design process. We begin by considering the key formal properties the system must satisfy. Next we outline a general design solution for the system as a whole. It is this solution that provides the insights into which patterns are most appropriate for the job. The heuristics are reminiscent of object-oriented design methods in which tentative solutions are first developed in order to identify candidate objects and later redefined once the objects are specified formally.

3.2.1 System properties

Even though this presentation is informal, some of the terminology we will employ to describe it is technical and belongs to the program verification vocabulary. For instance, system properties are divided into safety and progress conditions. The only progress property of interest to us is the requirement that each bag eventually reaches its destination. Safety properties dealing with the cars focus on their movement, e.g., carts cannot share the same track, can only move forward, must follow the track layout, must obey switch positions, cannot move without power, etc. Safety properties regarding the bags ensure that a bag remains on its cart until the destination is reached and that it does not change its destination along the way. A more precise and formal specification of the problem can be developed but it is not really needed in this section.

3.2.2 Solution outline

We plan to structure the system in accordance with the topology of the layout. Each track will have its own controller which will be able to communicate with the controllers associated with each of the adjoining tracks. The advantage of such an approach is that new systems can be put together out of smart components without redesign. This kind of highly distributed solution is likely to become important in applications that involve large sets of possible configurations. Once designed and verified, application engineers should be able to put together new systems by following a simple set of prescribed assembly rules which guarantee system correctness by construction.

The typical behavior of a track controller can be described by considering what happens when a single cart moves over the track. The controller receives a reservation request from one of its neighbors. The track is reserved and energized and the request is acknowledged if the controller has not made any other commitments. Once the cart arrives and its presence is detected by the entry sensor, the controller must inform the origin of the cart and must attempt to reserve the next section of track. If the reservation is acknowledged before the cart is ready to leave, the cart is allowed to depart and an acknowledgment is expected when the cart enters the next track. Otherwise, the power is temporarily cut off waiting for the next track to become available.

The selection of an outgoing track and the corresponding switch position is determined by the bag destination. This may be supplied by the controller making the reservation request or may be read by the scanner just prior to the arrival on the track. Since scanners are optional, controllers do not rely on them to detect cart arrival. Also, for a track involved in loading and unloading we assume that these actions take place before the scanner is encountered. This way we ensure that the destination is current. Whenever a destination is read, the entire path from the current track to the destination is recomputed. This is accomplished dynamically through a forward search. The shortest path is saved and passed along with the cart to the next controller.

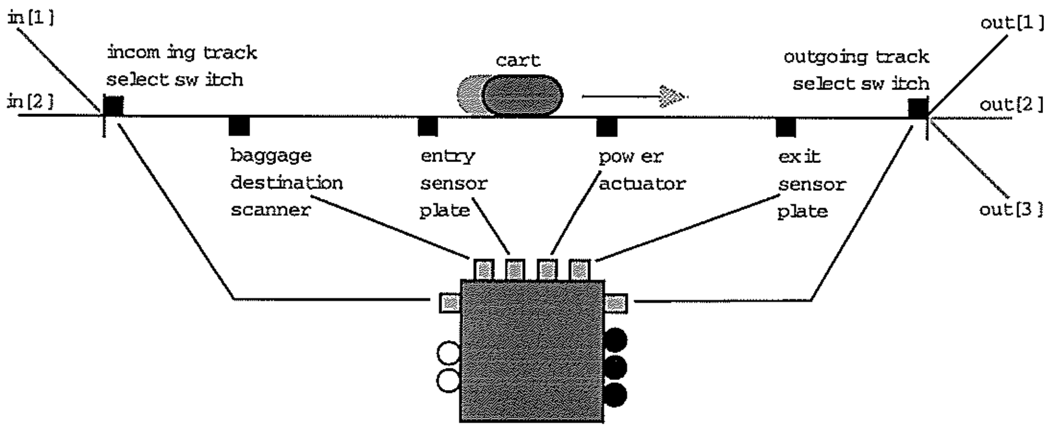
in[1]
in[2]
incoming track select switch
cart
outgoing track select switch
out[1]
out[2]
out[3]
baggage destination scanner
entry sensor plate
power actuator
exit sensor plate

Figure 7: The baggage cart system components with controller and ports.

Our tentative design sketch relies on the notion that polling sensors is orders of magnitude faster than the movement of carts. As a result, we make the following operational assumption about the sensors. Once activated, the sensor stays in the set position until the system resets it and the cart does not move any meaningful distance in the interim. In essence, this assumption captures certain real-time behavioral aspects of the system without the complexity of actually having to model time itself.

3.2.3 Search for patterns

Without having made much of an investment in the system design, certain features of a possible design solution are emerging already. First, tracks are being used as resources and allocated to carts. In our solution, only two tracks are allocated at a time – it is conceivable that other designs might allocate the entire path before the cart proceeds. A number of patterns deal with resource allocation. Dining and drinking philosophers are two that might come to mind immediately. It is important to notice, however, that the allocations shift along the graph following its connectivity in a manner that resembles a token passing pattern. It is this latter option that we will focus our attention on. Second, the path followed by a cart is computed dynamically. The rational behind the approach has to do with the added flexibility of being able to modify the track layout dynamically without changing the control logic. Path discovery involves search and the immediate choices are depth-first and breadth-first. The design, however, suggests the notion of a parallel search in which the first response is selected for immediate use and the rest are discarded. This becomes our second pattern. Assuming that we do have access to a catalog of design patterns (or a book on distributed algorithms for the time being) we will try to use available code samples as the starting point for the application development. Because pattern composition is an issue we have not addressed yet, we will follow a conservative strategy in which we develop one pattern first and later bring the other into the picture. The token passing pattern will be considered first.

3.3 Solution Development

Throughout this paper we will be using the UNITY programming notation to describe the evolving design. Programs consist of sets of conditional multiple-assignment statements (see the **assign** section). Statement selection satisfies a weak fairness constraint. Each statement, once selected, executes atomically. All program variables are declared using a Pascal-like notation (see the **declare** section). Macro-like definitions for expressions over the program variables appear in the **always** section while constraints over the set of initial values are defined in the **initially** section. In order to facilitate modularity and conciseness, we take some liberties with the basic UNITY notation. A separate variable declaration section has been added for those variables which are shared with other programs. We associate a separate program with each smart component of the system and allow them to communicate via shared variables–in UNITY, by convention, variables having the same name are shared among all the programs. It is more convenient for us to assume that all the variables are local unless declared as shared. Additional notation is provided to define explicitly which variables are the same in distinct programs, i.e., even the names of the shared variables are treated as local. Finally, we introduce a **define** section that provides for arbitrary macro definitions. The notation will become clear as we introduce the first version of the token passing design pattern below.

3.3.1 Token passing

Token passing algorithms have been used in a number of settings and for various purposes including fair access to communication buses. The version we use in this section is highly stylized and very easy to understand. This is precisely the kind of expectations we have from a collection of ready-to-use design patterns. This enhances the designer's level of confidence and reduces the chance of introducing errors during the design process.

3.3.1.1 Token passing on a ring. In our target design, each track has associated with it a smart controller. We assume that each controller runs one or more programs. For starters, we place a piece of the token passing protocol on each controller α. The UNITY code for the program appears in Figure 8. The variables *in* and *out* represent the input and output communication channels, respectively, while the variable *token* encodes the holding of the token. We assume that initially there are no tokens in transit, i.e., *in* and *out* hold undefined values. Some of the programs are initialized as holding a token as defined by the function *TokenInit*. The only assumption we make here is that the number of programs which hold a token initially is much smaller than the total number of components. The ring structure is made evident in the way the output channel of one program is matched with the input channel of the next–the CONNECTIVITY specification defines the ties among programs in a system. The definition of the function *next* ensures that the configuration is a ring, a situation that will have to be rectified later on in the design.

We assume that the pattern allows for multiple tokens because such a generalization does not increase the complexity of the program or the difficulty of the proof. The proof itself is not difficult and could be readily communicated even to practitioners not initiated in formal verification. One subtle point is the fact that the proof requires auxiliary variables to distinguish among tokens when verifying properties such as "each token eventually passes through each processor."

ALGORITHM

```
program TokenPass0(α)
    share
        in,out : boolean
    declare
        token : boolean
    initially
        in,out = false,false
    []  token = TokenInit(α)
    assign
        token,in := in,false       if ¬(token∨out) ∧ in
    []  token,out := false,token if token
end
```

CONNECTIVITY

$\text{TokenPass0}(\alpha).\text{out} = \text{TokenPass0}(\text{next}(\alpha)).\text{in}, \text{ where } \text{next}^n(\alpha) = \alpha$

Figure 8: Generic token passing design pattern.

3.3.1.2 Tokens with identifiers. Because the token is being used as a place holder for a cart, it makes sense to associate with each token an identifier that relates directly to the cart. Unfortunately, in our design sketch there is no unique identifier for carts, only for the bags they carry–we assume that the cart always carries a bag or pretends to do so by having a bar-coded label painted on its platform. Therefore, the token must acquire the label of the associated bag. Since all bags have distinct labels, the token identities will change over time but no two tokens will have the same identifier. This latter property will allow us, in most cases, to avoid the introduction of auxiliary variables that keep track of the token identity.

In all other respects the proof is not affected by this first refinement. The code changes are also minor. We generalize the type of the token away from a boolean. As can be seen in Figure 9, there is no explicit mention of a token identity. The token type becomes *token-type*. The identity of the token can be obtained by employing a function *id.t* which returns the identifier for token *t*. A property stating that each token identifier is unique within this system is added to the list of proof obligations. It affects mostly the definition of the initialization function *TokenInit*.

```
program TokenPass1(α)
    share
        in,out : token-type
    declare
        token : token-type
    always
        IsToken = (token≠⊥) [] IsIn = (in≠⊥) [] IsOut = (out≠⊥)
    initially
        in,out = ⊥,⊥
    []  token = TokenInit(α)
    assign
        token,in := in,⊥          if ¬(IsToken∨IsOut) ∧ IsIn
    []  token,out := ⊥,token      if IsToken
end
```

Figure 9: Token passing with token identity.

3.3.1.3 Tokens with a destination. In continuing to bring tokens closer to carts, one important attribute of the cart is that it has a particular destination station. Once reached, this destination is reassigned. Since we have a generic type for tokens now, adding this to the token itself is trivial. However, the statements of the system need a slight rewrite now, in order to compute the new destinations once the original destination has been completed. This refinement is depicted in Figure 10.

The program utilizes a function *NewDest*, to create a new destination for a token when needed. This is turn is used by *NewToken* which forms a new token if one is needed. It otherwise returns the original token back.

We have found the use of such functions to be very helpful in maintaining the clarity of the refined program. A more direct approach would be to add a new statement to the program which checked when the token had entered its destination, and at that point assign a new destination. In this case, and perhaps in others, it is clearer to abstract the notion of constructing a *new* token when it is being sent to the neighboring process, and let the constructor handle the different situations that may arise.

```
program TokenPass2(α)
    share
        in,out : token-type
    declare
        token : token-type
    always
        IsToken = (token≠⊥) [] IsIn = (in≠⊥) [] IsOut = (out≠⊥)
    []  NewToken(k)
            = k[dest/NewDest(α,k)] if dest.k=α
            ~ k otherwise
    initially
        in,out = ⊥,⊥
    []  token = TokenInit(α)
    assign
        token,in := in,⊥          if ¬(IsToken∨IsOut) ∧ IsIn
    []  token,out := ⊥,NewToken(token)
                                  if IsToken
end
```

Figure 10: Token passing with token destination.

3.3.1.4 Token passing on a graph. The alterations to the token passing algorithm have thus far been small and simple. This is intentional, as it simplifies the proof obligations. However, the baggage cart system has a graph topology, while the token passing algorithm has thus far remained on a ring. Here, we extend the control process of the token passing algorithm to handle multiple inputs and outputs. The key is to note that the changes do not alter the fundamental safety properties, such as permitting at most one token to enter, nor the fundamental progress properties, that a waiting token will eventually be admitted and a token will eventually be released to the next control process. The program in Figure 11 takes on a decidedly different look from the previous programs. The *in* and *out* variables have now become arrays. The function *LineOut* determines which index into the *out* to send the token onto given the current destination. *LineIn* is used to fairly receive tokens waiting to be accepted. To accomplish this, *LineIn* has side effects–every time it is invoked the value of s is incremented by one modulo n.

The intuition behind this refinement has to do with the fact that, despite the graph structure, the path followed by the token to its destination is still a linear one. The proof of the token eventually reaching the destination remains unchanged in its essential features. The same decreasing metric, distance to destination, can be used to show progress.

```
program TokenPass3(α)
    share
        in[0..n-1],out[0..m-1] : token-type
    declare
        s : integer
        token : token-type
    define
        in=in[LineIn(s)] [] out=out[LineOut(dest.NewToken(token))]
    always
    []  NewToken(k)
            = k[dest/NewDest(α,k)] if dest.k=α
            ~ k otherwise
    initially
        ⟨||i : 0 ≤ i < n :: in[i] = ⊥ ⟩ || ⟨||j : 0 ≤ j < m :: out[j] = ⊥ ⟩
    []  token = TokenInit(α)
    assign
        in,token := ⊥,in
            if token=⊥ ∧ ⟨∀ t :: out[t]=⊥⟩
    []  out,token := NewToken(token),⊥
            if token≠⊥
end
```

Figure 11: Token passing algorithm with multiple input and output channels.

3.3.1.5 Token passing with token path. Up to this point *LineOut* assumed that some fixed path existed for every destination and used the latter to select the proper output line. This increases the storage requirements for the system and undermines flexibility, e.g., the ability to reconfigure the track layout without making changes to each smart device involved in the system configuration. For these reasons, we extend the token definition to include a precomputed path. It is the function *NewPath* which is responsible for this computation while *LineOut* is subjected only to minor modifications. Once computed, the path is required to remain fixed until the token reaches the final destination, which is the last identifier in the path. The computation of the new path will be the tie point between the two patterns that we employ in this design.

```
prpgram TokenPass4(α)
    share
        in[0..n-1],out[0..m-1] : token-type
    declare
        s : integer
        token : token-type
    define
        in=in[LineIn(s)] [] out=out[LineOut(path.NewToken(token))]
    always
    []  NewToken(k)
            = k[path/NewPath(NewDest(α,k))] if dest.k=α
            ~ k otherwise
    initially
        ⟨||i : 0 ≤ i < n :: in[i] = ⊥ ⟩ || ⟨||j : 0 ≤ j < m :: out[j] = ⊥ ⟩
    []  token = TokenInit(α)
    assign
        in,token := ⊥,in
            if token=⊥ ∧ ⟨∀ t :: out[t]=⊥⟩
    []  out,token := NewToken(token),⊥
            if token≠⊥
end
```

Figure 12: Token passing algorithm with a token path.

3.3.1.6 Token passing with explicit modes. It is the state of the controller that will determine ultimately the behavior of the carts. In this refinement we take a first step towards shifting the view away from the token perspective imposed by the design pattern and towards the control states one expects from the application. The changes do not affect the correctness of the program. The values assumed by the variable *mode* are simply a reflection of the current state of the program. The presence of *mode* in the assignment conditions is redundant for now but in later refinements we plan to replace predicates involving *token* by predicates involving *mode*. The three values assumed by *mode* are: *idle* which indicates that no token is present; *busy* which captures the fact that the token is present; and *leaving* which addresses the situation in which the token has been passed on but it may not have arrived to its destination.

```
program TokenPass5($\alpha$)
    share
        in[0..n-1],out[0..m-1] : token-type
    declare
        s : integer
        token : token-type
        mode : (idle,busy,leaving)
    define
        in=in[LineIn(s)] [] out=out[LineOut(path.NewToken(token))]
    always
    []  NewToken(k)
            = k[path/NewPath(NewDest($\alpha$,k))] if dest.k=$\alpha$
            ~ k otherwise
    initially
        $\langle \| i : 0 \leq i < n :: \text{in}[i] = \perp \rangle \, \| \, \langle \| j : 0 \leq j < m :: \text{out}[j] = \perp \rangle$
    []  token = TokenInit($\alpha$)
    []  mode = busy if token$\neq\perp$
            ~ idle otherwise
    assign
        in,token,mode := $\perp$,in,busy
            if token=$\perp$ $\wedge$ $\langle\forall$ t :: out[t]=$\perp\rangle$ $\wedge$ mode=idle
    []  out,token,mode := NewToken(token),$\perp$,leaving
            if token$\neq\perp$ $\wedge$ mode=busy
    []  mode := idle
            if $\langle\forall$ t :: out[t]=$\perp\rangle$ $\wedge$ mode=leaving
end
```

Figure 13: Token passing algorithm with modes.

3.3.1.7 Token passing with expanded modes. There are fundamental differences between the way carts and tokens can be controlled. A token arriving on an input can be delayed until the controller is ready to pick it up. An arriving cart cannot be allowed to enter a track that is not empty. Advance knowledge is needed to determine the fact that the track will be ready to receive the cart. Some sort of advanced reservation is needed. Adding a reservation mechanism involves receiving a reservation request and acknowledging it only when the track becomes available. This adds an intermediary state between *idle* and *busy*, we call it *reserved*. Next, once the cart, represented by a token, arrives it becomes necessary to send the reservation request and to wait for the acknowledgment–the state *reserving* is added. Finally, we need to distinguish between the case when the token is ready to depart but needs to wait for the reservation acknowledgment–the new state *waiting*. The revised program below captures these changes.

```
program TokenPass6(α)
    share
        in[0..n-1],out[0..m-1] : token-type
    declare
        s : integer
        token : token-type
        mode : (idle,reserved,busy,reserving,waiting,leaving)
    define
        in=in[LineIn(s)] [] out=out[LineOut(path.NewToken(token))]
    always
        NewToken(k)
            = k[path/NewPath(NewDest(α,k))] if dest.k=α
            ~ k otherwise
    initially
        ⟨||i : 0 ≤ i < n :: in[i] = ⊥ ⟩ || ⟨||j : 0 ≤ j < m :: out[j] = ⊥ ⟩
    []  token = TokenInit(α)
    []  mode = busy if token≠⊥
            ~ idle otherwise
    assign
        in,mode := ⊥,reserved
            if in=reserve ∧ mode=idle
    []  in,token,mode := ⊥,in,busy
            if in≠⊥ ∧ mode=reserved
    []  out,mode := reserve,reserving
            if mode=busy
    []  mode := waiting
            if out≠⊥ ∧ mode=reserving
    []  out,token,mode := NewToken(token),⊥,leaving
            if out=⊥ ∧ (mode=reserving ∨ mode=waiting)
    []  mode := idle
            if (∀ t :: out[t]=⊥) ∧ mode=leaving
end
```

Figure 14: Token passing algorithm with expanded modes.

3.3.1.8 Token passing with power actuator. Thus far, we ignored the manner in which the controller interacts with a baggage cart. The algorithm is still concerned only with the movement of tokens. Intuitively, a cart would simply follow a token around on the track, and since we can demonstrate that safe passage for tokens is provided by the system, the safe passage of carts would be assured. In reality, a cart moves about the tracks only as long as power is supplied to it. The controller of a cart would need to stop a cart if it attempts to leave the track before the controller has knowledge that it is safe to do so. The only state that relates to this situation is *waiting*. We need to make sure that the cart cannot advance while the controller is in this state. To capture the interaction between the controller and the energizing of the track we add a variable *power* which denotes the actuator for the power relay. The modified program makes sure that the power is turned off whenever the controller is waiting for a reply to a reservation request. A further optimization could be made by observing the fact that in the *idle* state there is no need for power–we choose not to make this change.

```
program TokenPass7($\alpha$)
    share
        in[0..n-1],out[0..m-1] : token-type
        power : boolean
    declare
        s : integer
        token : token-type
        mode : (idle,reserved,busy,reserving,waiting,leaving)
    define
        in=in[LineIn(s)] [] out=out[LineOut(path.NewToken(token))]
    always
        NewToken(k)
            = k[path/NewPath(NewDest($\alpha$,k))] if dest.k=$\alpha$
            ~ k otherwise
    initially
        $\langle \| i : 0 \leq i < n :: \text{in}[i] = \perp \rangle \parallel \langle \| j : 0 \leq j < m :: \text{out}[j] = \perp \rangle$
    []  token = TokenInit($\alpha$)
    []  mode = busy if token$\neq\perp$
            ~ idle otherwise
    []  power = true
    assign
        in,mode := $\perp$,reserved
            if in=reserve $\wedge$ mode=idle
    []  in,token,mode := $\perp$,in,busy
            if in$\neq\perp$ $\wedge$ mode=reserved
    []  out,mode := reserve,reserving
            if mode=busy
    []  power,mode := false,waiting
            if out$\neq\perp$ $\wedge$ mode=reserving
    []  out,token,mode := NewToken(token),$\perp$,leaving
            if out=$\perp$ $\wedge$ mode=reserving
    []  out,token,power,mode := NewToken(token),$\perp$,true,leaving
            if out=$\perp$ $\wedge$ mode=waiting
    []  mode := idle
            if $\langle \forall$ t :: out[t]=$\perp\rangle$ $\wedge$ mode=leaving
end
```

Figure 15: Power actuator control.

3.3.1.9 Token passing with sensor detection. Factoring in the use of sensors and their interactions with the control system is complicated by the absence of the notion of time in the model and by the difficulties related to formal modeling of the environmental behavior. One can defeat the control system, for instance, by lifting carts off the tracks and by placing them in random positions on other tracks. One can also run the control system at such low clock rates that the information received from sensors is always out of date. We assume that none of these are happening and formalize this fact in terms of two behavioral assumptions. First, we assume that a cart does not move away from a sensor before the system resets it, i.e., the software execution speed is orders of magnitude faster than the cart movement speed. Second, we assume that sensors turn on in a manner that is consistent to normal cart movement, e.g., the entry sensor is activated before the exit sensor.

In this last refinement we add two sensors, *entry* and *exit.* They communicate the cart's arrival onto the track and its imminent departure for the next track. Once tested, the sensor in question is reset. Upon arrival of the cart, the token waiting already on the input is accepted. Once the cart is ready to depart, the token is put on the output, unless a confirmation for the reservation request is still pending. In this latter case, the power is turned off. A cursory look at the resulting program may suggest also the possibility that the exit sensor is triggered while the controller is still in the *busy* state. If this happens, the sensor is not reset unless the state changes to *reserving* and the cart does not move away for the sensor before this happens. This is simply a reflection of the fact that the cart movement allows sufficient time to send the reservation request out before traveling the entire length of a track.

This completes our derivation of the system except for the manner in which the cart path is selected whenever a change of destination takes place. This is the subject of the next section.

```
program TokenPass8(α)
    share
        in[0..n-1],out[0..m-1] : token-type
        power : boolean
        entry,exit : boolean
    declare
        s : integer
        token : token-type
        mode : (idle,reserved,busy,reserving,waiting,leaving)
    define
        in=in[LineIn(s)] [] out=out[LineOut(path.NewToken(token))]
    always
        NewToken(k)
            = k[path/NewPath(NewDest(α,k))] if dest.k=α
            ~ k otherwise
    initially
        $\langle \| i : 0 \le i < n :: \text{in}[i] = \perp \rangle \parallel \langle \| j : 0 \le j < m :: \text{out}[j] = \perp \rangle$
    []  token = TokenInit(α)
    []  mode = busy if token≠⊥
            ~ idle otherwise
    []  power = true
    []  entry,exit = false,false
    assign
        - accept reservation and send confirmation
        in,mode := ⊥,reserved
            if in=reserve ∧ mode=idle
        - accept token upon cart arrival
    []  in,token,entry,mode := ⊥,in,false,busy
            if in≠⊥ ∧ entry ∧ mode=reserved
        - make reservation request
    []  out,mode := reserve,reserving
            if mode=busy
        - cut power off to departing cart while waiting for confirmation
    []  power,exit,mode := false,false,waiting
            if out≠⊥ ∧ exit ∧ mode=reserving
        - allow cart to move on to the next track
    []  out,token,exit,mode := NewToken(token),⊥,false,leaving
            if out=⊥ ∧ exit ∧ mode=reserving
        - restore power upon receiving confirmation
    []  out,token,power,mode := NewToken(token),⊥,true,leaving
            if out=⊥ ∧ mode=waiting
        - accept acknowledgment of the cart's arrival on the next track
    []  mode := idle
            if ⟨∀ t :: out[t]=⊥⟩ ∧ mode=leaving
end
```

Figure 16: Program with cart detection sensors.

3.3.2 Search

We turn now to the question of how to implement the dynamic selection of the cart path encapsulated in the function *NewPath*. We remind the reader that upon receiving a new bag, a scanner reads its destination. This, in turn, results in a search for a path to be followed by the cart. By selecting the path dynamically we allow the configuration to change by adding new sections of track as needed. The removal or closing of tracks is handled in an indirect manner. Because the closing of a track requires the recomputation of the path, we simply need to force this recomputation by turning on some scanner in a section of track that is reached before entering the closed track and by turning off access to the closed track. However, we need to make sure that no carts are on the way along the path between the scanner and the closed track at the time–we do not actually carry out the example to this level of detail in this paper. An unpleasant consequence of adding the track removal mechanics is the fact that the progress proof for reaching the destination breaks. The well-founded metric used to show that the distance to the destination decreases may be increased by the track closing. If we do have an upper bound on the number of closings that a cart may encounter, a new metric can be devised with minimal changes to the proof–a pair of integers where the first one decreases with each encountered closing and the second one is the current distance to the destination.

We assume the existence of one search module per track. Each search module communicates with the corresponding controller to receive a destination used to seed the search and with the search modules on the neighboring tracks. All communications are bi-directional. Many searches can be carried out simultaneously but, due to the fact that they are treated as independent activities, we need to concern ourselves only with a single search. The design pattern that forms the basis for this part of the derivation is a parallel search pattern on a tree structure. The presentation follows the same style we employed in the previous section but, for the sake of brevity, occasionally we combine several steps into one.

3.3.2.1 Parallel search on a tree. The starting point for the derivation is a search design pattern shown in Figure 17. The pattern we use here was actually derived from a simpler broadcast and acknowledgment design pattern but we omit that part of the derivation since it is reasonable to expect the availability of search patterns in a pattern catalog. In the search pattern, the network configuration is assumed to be a tree. The request comes in the form of a node name to the root node and propagates from the root towards the leaves until either the node in question is reached or a leaf is encountered, i.e., the node has not been found along that path. An *ACK* or a *NACK* is returned to the parent in the tree. Each parent in the tree combines the answers received from the children and passes a reply up the tree. When the root receives replies from all its children, it sends a clear request down the tree. The request propagates in the same manner with a final acknowledgment arriving at the root.

The root of the tree represents the search module associated with a track that has a working scanner. When the destination is read, it is placed on the input to the search module. The leaf nodes represent tracks where the search can stop. They could be defined statically, but here again we prefer a dynamic solution. The derivation process will seek to eventually define automatically which nodes should be treated as leaves. At this point, however, the design pattern assumes that the tree is given.

The mechanics of the message propagation is accomplished in the following manner. Each module has one back channel *b* and possibly several forward channels *f[i]*. These are the local names used inside a search module and each back channel for one module is the forward channel of a preceding module. Each channel is bi-directional and consists of a one-place message buffer that can be written only by the sender and read by the receiver. A request is placed by the parent in *in.b*. The child passes on the request on its forward channels by copying it in *out.f[i]*. The replies fill the buffers in the opposite direction. The clear message is represented by the removal of requests in the forward direction while the acknowledgment of the clear is captured by the removal of the replies in the backward direction. There is one exception to this. The root node does not clear the original request. The reason for this is that, at this point, there is no connection between this program and the track controller.

ALGORITHM

```
program Search1($\alpha$)
    share
        f[1..n],b : message-channel
    always
        $\langle \| i :: \text{done}(i) = (\text{in.f}[i] \neq \perp) \rangle$
    [] $\langle \| i :: \text{clear}(i) = (\text{in.f}[i] = \perp) \rangle$
    [] $\langle \| i :: \text{found}(i) = (\text{in.f}[i] = (\text{ACK})) \rangle$
    [] DONE = $\langle \forall i :: \text{done}(i) \rangle$
    [] FOUND = $\langle \exists i :: \text{found}(i) \rangle$
    [] ARRIVED = $(\text{in.b} \neq \perp \wedge \text{dest.in.b} = \alpha)$
    [] FORWARD = $(\text{in.b} \neq \perp \wedge \text{dest.in.b} \neq \alpha)$
    [] RETURN = $((\text{DONE} \vee \text{ARRIVED}) \wedge \text{in.b} \neq \perp \wedge \text{out.b} = \perp)$
    [] RETURNED = $(\text{DONE} \wedge \text{out.b} \neq \perp \wedge (\text{in.b} = \perp \vee \langle \forall \gamma :: \neg \kappa(\gamma, \alpha) \rangle))$
    [] CLEARED = $\langle \forall i :: \text{clear}(i) \rangle \wedge \text{in.b} = \perp$
    [] receipt
            = (ACK) if FOUND $\vee$ ARRIVED
            ~ (NACK) otherwise
    [] $\langle \| k : k \text{ a message} :: \text{new}.k = k \rangle$
    initially
        $\langle \| i :: \text{in.f}[i], \text{out.f}[i] = \perp, \perp \rangle$
    || in.b,out.b
            = $\perp, \perp$ if $\langle \exists \gamma :: \kappa(\gamma, \alpha) \rangle$
            ~ Destination($\alpha$),$\perp$otherwise
    assign
        $\langle \| i : \text{FORWARD} \wedge \langle \forall j :: \text{out.f}[j] = \perp \rangle :: \text{out.f}[i] := \text{new.in.b} \rangle$
    [] out.b := receipt if RETURN
    [] $\langle \| i : \text{RETURNED} :: \text{out.f}[i] := \perp \rangle$
    [] out.b := $\perp$ if CLEARED
end
```

CONNECTIVITY

$$\begin{aligned} \kappa(\alpha,\beta) &\equiv \langle \exists i :: \text{Search1}(\alpha).\text{f}[i] = \text{Search1}(\beta).\text{b} \rangle \\ \kappa(\alpha,\beta) &\Rightarrow \langle \forall \gamma : \gamma \neq \alpha :: \neg \kappa(\gamma,\beta) \rangle \end{aligned}$$

Figure 17: Parallel search pattern.

3.3.2.2 Parallel tree search with distance computation. In this refinement, the message carries a distance field which is updated along the way towards the destination. (See Figure 18.) An infinite value is used to indicate that the search failed along the particular branch. The only modifications to the program involve the representation of messages which results in the redefinition of *ACK*, *NACK*, and the action of the function *new*, which generates the successful reply or the next message to be sent along the forward direction. In both cases it adjusts the distance accordingly.

ALGORITHM

```
program Search2($\alpha$)
    share
        f[1..n],b : message-channel
    always
        $\langle \| i :: \text{done}(i) = (\text{in.f}[i] \neq \perp)\rangle$
    []  $\langle \| i :: \text{clear}(i) = (\text{in.f}[i] = \perp)\rangle$
    []  $\langle \| i :: \text{found}(i) = (\text{done}(i) \wedge \text{dist.in.f}[i] \neq \infty)\rangle$
    []  $\text{DONE} = \langle \forall i :: \text{done}(i)\rangle$
    []  $\text{FOUND} = \langle \exists i :: \text{found}(i)\rangle$
    []  $\text{ARRIVED} = (\text{in.b} \neq \perp \wedge \text{dest.in.b} = \alpha)$
    []  $\text{FORWARD} = (\text{in.b} \neq \perp \wedge \text{dest.in.b} \neq \alpha)$
    []  $\text{RETURN} = ((\text{DONE} \vee \text{ARRIVED}) \wedge \text{in.b} \neq \perp \wedge \text{out.b} = \perp)$
    []  $\text{RETURNED} = (\text{DONE} \wedge \text{out.b} \neq \perp \wedge (\text{in.b} = \perp \vee \langle \forall \gamma :: \neg\kappa(\gamma, \alpha)\rangle))$
    []  $\text{CLEARED} = \langle \forall i :: \text{clear}(i)\rangle \wedge \text{in.b} = \perp$
    []  receipt
            = new.in.b if ARRIVED
            $\sim$ in.b[dist/$\langle \min i :: \text{dist.in.f}[i]\rangle$] if FOUND
            $\sim$ in.b[dist/$\infty$] otherwise
    []  $\langle \| k : k \text{ a message} :: \text{new}.k = k[\text{dist}/(\text{dist}+1)]\rangle$
    initially
        $\langle \| i :: \text{in.f}[i], \text{out.f}[i] = \perp, \perp\rangle$
    ||  in.b,out.b
            $= \perp, \perp$ if $\langle \exists \gamma :: \kappa(\gamma, \alpha)\rangle$
            $\sim$ (Destination($\alpha$),0),$\perp$ otherwise
    assign
        $\langle \| i : \text{FORWARD} \wedge \langle \forall j :: \text{out.f}[j] = \perp\rangle :: \text{out.f}[i] := \text{new.in.b}\rangle$
    []  out.b := receipt if RETURN
    []  $\langle \| i : \text{RETURNED} :: \text{out.f}[i] := \perp\rangle$
    []  out.b := $\perp$ if CLEARED
end
```

CONNECTIVITY

$$\kappa(\alpha,\beta) \equiv \langle \exists i :: \text{Search2}(\alpha).\text{f}[i] = \text{Search2}(\beta).\text{b}\rangle$$
$$\kappa(\alpha,\beta) \Rightarrow \langle \forall \gamma : \gamma \neq \alpha :: \neg\kappa(\gamma,\beta)\rangle$$

Figure 18: Distance computation during search.

3.3.2.3 Parallel tree search for a path. The program developed so far does not provide the path itself. In this next refinement, the search messages are augmented with information about which nodes were visited along the way. When the destination is found, this information is returned in the form of a path to the destination. It is convenient for us to continue to use the infinity symbol to denote a path that failed to find the destination. We denote path construction using the plus symbol and we introduce the function *src* to return the first node along the path, i.e., the root of the tree. We also extend *min* so as to return one of the paths having the shortest length–it allows for the possibility to have two nodes with the same name even though the tree structure disallows it at this point. As evident in Figure 19, the assign section continues to be unaffected. All program changes involve only definitions and the initialization.

```
program Search3($\alpha$)
    share
        f[1..n],b : message-channel
    always
        $\langle \| i$ :: done($i$) = (in.f[$i$]$\neq\perp$)$\rangle$
    []  $\langle \| i$ :: clear($i$) = (in.f[$i$]$=\perp$)$\rangle$
    []  $\langle \| i$ :: found($i$) = (done($i$) $\wedge$ path.in.f[$i$]$\neq\infty$)$\rangle$
    []  DONE = $\langle \forall i$ :: done($i$)$\rangle$
    ||  FOUND = $\langle \exists i$ :: found($i$)$\rangle$
    ||  ARRIVED = (in.b$\neq\perp$ $\wedge$ dest.in.b=$\alpha$)
    ||  FORWARD = (in.b$\neq\perp$ $\wedge$ dest.in.b$\neq\alpha$)
    []  RETURN = ((DONE $\vee$ ARRIVED) $\wedge$ in.b$\neq\perp$ $\wedge$ out.b=$\perp$)
    []  RETURNED = (DONE $\wedge$ out.b$\neq\perp$ $\wedge$ (in.b=$\perp$ $\vee$ src.out.b=$\alpha$))
    []  CLEARED = $\langle \forall i$ :: clear($i$)$\rangle$ $\wedge$ in.b=$\perp$
    []  receipt
            = new.in.b if ARRIVED
            $\sim$ in.b[path/(min $i$ :: path.in.f[$i$])] if FOUND
            $\sim$ in.b[path/$\infty$] otherwise
    []  $\langle \| k$ : $k$ a message :: new.$k$ = $k$[path/(path+$\alpha$)]$\rangle$
    initially
        $\langle \| i$ :: in.f[$i$],out.f[$i$] = $\perp,\perp\rangle$
    ||  in.b,out.b
            = $\perp,\perp$ if $\langle \exists\gamma$ :: $\kappa(\gamma,\alpha)\rangle$
            $\sim$ (Destination($\alpha$),$\alpha$),$\perp$ otherwise
    assign
    []  $\langle \| i$ : FORWARD $\wedge$ $\langle \forall j$ :: out.f[$j$]=$\perp\rangle$ :: out.f[$i$] := new.in.b $\rangle$
    []  out.b := receipt if RETURN
    []  $\langle \| i$ : RETURNED :: out.f[$i$]:=$\perp$ $\rangle$
    []  out.b := $\perp$ if CLEARED
end
```

Figure 19: Path searching algorithm.

3.3.2.4 Searching the graph. While our previous algorithms have been searching along a tree, the actual search will be performed over a more general directed graph. This refinement addresses the question of constructing the tree while following the same basic computation. The idea is to construct a spanning tree by detecting the points where two or more branches of the search meet and by cutting the search for all but one of them. The code is modified slightly to account for the processing associated with each of the possible inputs. Let us consider the case of a node which is neither the root of the search nor the destination. The first request to arrive on the back channel is propagated along all the forward channels. A subsequent request can fall into one of two categories. If it arrives along a path to the root that is longer than the shortest known path so far, it is rejected by returning a path of infinite length. If the new request has a path which is shorter than the shortest one, it is kept and rejections are sent on the back channels to the other requests. The function *min* is assumed to make a deterministic choice among multiple paths of equal minimal length. Once all replies return on the forward channels, the back channel selected as the minimum receives the best path found down the line. At this point any additional late requests are simply rejected. Finally, the arrival of the first clear request on one of the back channels is forwarded and, when acknowledgements are received along all the forward channels, each back channel is cleared thus passing the acknowledgement up the tree to the root. Clear requests on channels that received reject messages are acknowledged without regard to the state of the forward channels in order to avoid possible deadlocks when a node is present along a path the loops back on itself. In analyzing the solution, it is helpful to notice that a path that loops through a node cannot be minimal and is always rejected and that a node which turns out to be the destination does not send out any requests on the forward channels.

This concludes the main part of the derivation process. What remains to be done next is the composition of the two separate solutions we have derived so far.

ALGORITHM

```
program Search4($\alpha$)
    share
        f[1..n],b[1..m] : message-channel
    always
        $\langle \| i$ :: done($i$) = (in.f[$i$]$\neq\perp$)$\rangle$
    [] $\langle \| i$ :: clear($i$) = (in.f[$i$]$=\perp$)$\rangle$
    [] $\langle \| i$ :: found($i$) = (done($i$) $\wedge$ dest.in.f[$i$]$\neq\infty$)$\rangle$
    [] DONE($x$) = ($\langle\forall i$ :: done($i$)$\rangle$ $\vee$ $\neg$MIN($x$) $\vee$ LATE)
    [] FOUND = $\langle\exists i$ :: found($i$)$\rangle$
    [] ARRIVED = $\langle\exists i$ :: in.b[$i$]$\neq\perp$ $\wedge$ dest.in.b[$i$]$=\alpha\rangle$
    [] FORWARD($x$) = (in.b[$x$]$\neq\perp$ $\wedge$ dest.in.b[$x$]$\neq\alpha$)
    [] RETURN($x$) = ((DONE($x$) $\vee$ ARRIVED) $\wedge$ in.b[$x$]$\neq\perp$ $\wedge$ out.b[$x$]$=\perp$)
    [] RETURNED($x$) = (DONE($x$) $\wedge$ $\langle\exists i$ :: out.b[$i$]$\neq\perp$ $\wedge$ (in.b[$i$]$=\perp$ $\vee$ src.out.b[$i$]$=\alpha$)$\rangle$)
    [] CLEARED($x$) = (($\langle\forall i$ :: clear($i$)$\rangle$ $\wedge$ $\langle\exists i$ :: in.b[$i$]$=\perp$ $\wedge$ out.b[$i$]$\neq\perp\rangle$)
                    $\vee$ (in.b[$x$]$=\perp$ $\wedge$ $\alpha\in$path.out.b[$x$])
    [] MIN($x$) = (path.in.b[$x$]=$\langle$min $i$ :: path.in.b[$i$]$\rangle$)
    [] LATE = $\langle\exists i$ :: out.b[$i$]$\neq\perp\rangle$
    [] receipt($x$)
            = new.in.b[$x$] if ARRIVED $\wedge$ MIN($x$) $\wedge$ $\neg$LATE
            $\sim$ in.b[$x$][path/(path+suffix($\alpha$,$\langle$min $i$ :: path.in.f[$i$]$\rangle$))]
                if FOUND $\wedge$ MIN($x$) $\wedge$ $\neg$LATE
            $\sim$ in.b[$x$][path/$\infty$+$\alpha$] otherwise
    [] $\langle \| k$ : $k$ a message :: new.$k$ = $k$[path/(path+$\alpha$)]$\rangle$
    initially
        $\langle \| i$ :: in.f[$i$],out.f[$i$] = $\perp,\perp\rangle$
    || in.b,out.b
            = $\perp,\perp$ if $\langle\exists\gamma$ :: $\kappa(\gamma,\alpha)\rangle$
            $\sim$ (Destination($\alpha$),$\alpha$),$\perp$ otherwise
    assign
        $\langle$[]s ::
                $\langle \| i$ : FORWARD(s) $\wedge$ $\langle\forall j$ :: out.f[$j$]$=\perp\rangle$ :: out.f[$i$] := new.in.b[s] $\rangle$
            []  out.b[s] := receipt(s) if RETURN(s)
            []  $\langle \| i$ : RETURNED(s) :: out.f[$i$]:=$\perp\rangle$
            []  out.b[s] := $\perp$ if CLEARED(s)
        $\rangle$
end
```

CONNECTIVITY

$$\kappa(\alpha,\beta) \equiv \langle\exists i,j :: \text{Search4}(\alpha).\text{f}[i] = \text{Search4}(\beta).\text{b}[j]\rangle$$

Figure 20: Path searching on a graph.

3.4 Schema composition

At the moment, the token passing and search programs are completely disconnected. The token passing program allows a cart to safely traverse a path through the graph to its destination. The search program will find a path to a desired destination. Now comes the task of integrating these two programs. When the cart receives a new piece of baggage which is scanned for a destination, the current track automatically receives a request for a path to it. This initiates the search algorithm. Meanwhile, the cart is moving through the track. The track does not allow the cart to continue before a path has been computed. After the path is computed, it is stored in the token, and the cart can continue towards its destination.

This is achieved is through the definition of the function *NewPath*. *NewPath* is the mechanism by which the token passing algorithm obtains the results from the layer below. The computation is initialized when the destination scanner retrieves a new destination from the cart.

This composition requires a slight modification to the token passing program to power down the cart if the path search is not completed before the cart attempts to move off the track. Notice that this action occurs also if the reservation request has not yet been acknowledged. In this situation, a convenient mechanism to utilize is to allow *NewPath* to behave as an *asynchronous remote procedure call*. Its initial invocation is when *out* is being computed for the reservation request. The request finishes upon return from the *NewPath*, and when the request is acknowledged, the cart is allowed to pass.

This is only one way that we can compose distributed systems. It has the unusual property that there is a single point of contact and a single synchronization condition. This scheme would not work, for instance, if the cart picks up the response later down the track. A more dynamic point of interaction would have to be established between the two systems being composed. An important issue for further research is to define a repertoire of simple structured composition rules and to study their formal properties.

4 Discussion

The ultimate goal of this research is to free the engineer building distributed applications from the burden of having to verify each new system or any changes to an existing one. The starting point was the idea of building large distributed systems from small trusted components by following application-specific composition rules that guarantee the correctness of the resulting system by construction; advances in model checking and automatic verification could be used at the component level while the rules could be verified once and used across a large class of systems.

Having demonstrated the basic idea in the context of one application, we set out to develop the formal underpinning of the approach and to formalize a design methodology that could be employed in other settings. The first strategy we investigated associated with each application component a program which encapsulated its control logic, generalized its behavior and structure by comparing the programs associated with diverse types of components, and assembled them into a system by using shared variables or message passing. The investigation suggested that our initial success was primarily the result of careful crafting of the components and the development of regular composition rules. In other words, the strategy we followed was application-specific, required a very high degree of creativity, and entailed standard but complex proofs over generalized system structures. Formal characterization of the process seemed to be very difficult to achieve. Furthermore, issues relating to the composition rules, while they took advantage of the system topology, did not appear to be specific to the method being used. Nevertheless, the concept of having an isomorphic relation between the system's physical and control topologies continued to be an attractive design strategy.

The regular nature of the composition led us to investigate possible ways that one might structure the interactions among programs in order to provide certain strong guarantees which, in turn, could reduce the complexity of the proofs. We focused our attention on the potential impact of using various schemas (e.g., communication via variables that have a single writer). However, we concluded that, even when a particular schema reduces the complexity of the verification, the process by which the system is developed remains unaffected. We still design individual components and prove properties of a system resulting from their composition into a single larger program. The approach did raise, however, questions about the fine-grained nature of the components that are being composed. This naturally brought about the notion that we ought to compose not processes but distributed systems. The new idea that emerged was to develop systems which are simple and highly specialized for a particular subtask (structurally isomorphic) and combine them in some structured way. While raising some interesting questions about new forms of composition among distributed systems (rather than concurrent programs), the question of how to develop these specialized systems remained open and, in some way, brought us back to where we started in the first place.

The concept of a distributed design pattern emerged out of this context. It is rooted in the notion that established distributed algorithms provide a rich repository of experience and analytical studies. Its abstract, language-independent, and formal nature make it ideal for reuse. They only issue was how to bridge the gap between the generality of the algorithms and the specificity of the application. Program derivation provided the formal foundation while the strategy of matching the system structure to the physical structure of the application allowed us to achieve a certain degree of specialization.

The next steps in this research must be a combination of formal and empirical studies.

First, a more formal treatment of the derivation process must be developed. Frequently encountered program transformations must be cataloged and formalized. We made extensive use of data refinement techniques, coupling invariants, and atomic action refinement. We also provided a simple method for handling the interfaces to the environments, an approach that masks any references to real time. Other techniques for dealing with the environment need to be investigated as well. As anyone might have expected, composition techniques are a central issue in this kind of research. Our work, however, brings about a new perspective on this topic. The notion of composing distributed systems having similar topologies does not seem to have been addressed in the literature at this time. The example used in this paper employs a very simple composition mechanism with a single point of contact and a call-return interface. This made verification easy. One system perceived the other as a simple function call, the other simply reacted to a stimulus and returned a value. Other more complex interface patterns need to be categorized and their implications on the complexity of the verification task must be evaluated.

Future formal studies, must be accompanied by the development and evaluation of a catalog of design patterns for a specific application domain. This is the only way that formal results will be able to have a credible impact on industrial practices. In our presentation, we attempted to give the reader a sense of the level of rigor that an engineer using distributed design patterns will be expected to apply to the design process. Because we started with a trusted pattern and we modified it in small increments, the proofs were omitted by accepting a small element of risk. In situations where this is not acceptable the added cost of verification can still be kept manageable.

5 Conclusions

Distributed design patterns, as proposed in this paper, own their philosophical underpinning to object-oriented programming and borrow their methods from program derivation. The result is a design strategy that is distinct from both. The approach allows us to leverage off a large body of previous research on distributed algorithms. Distributed applications are built through the composition of multiple distributed computations that have the topology of the underlying physical system, can be developed in a systematic manner, and entail minimal verification efforts. In the paper we used an airport baggage delivery application as a vehicle for our discussion and relied on standard distributed algorithms to furnish us with needed design patterns. The extended example we constructed demonstrates the feasibility of the approach. In addition, it sketched out the essential features associated with this kind of design process and identified the key research issues requiring further investigation.

References

[1] R.J.R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13(2-3):133–80, May 1990.

[2] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison Wesley, May 1989.

[3] James O. Coplien and Douglas C. Schmidt. *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[4] J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems*, LNCS 430. Springer-Verlag, 1989.

[5] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[7] Matthew Hennessy. *Algebraic Theory of Processes.* The MIT Press, 1988.

[8] Simon S. Lam and A. Udaya Shankar. Refinement and projection of relational specifications. In de Bakker et al. [4].

[9] Amy L. Murphy, Gruia-Catalin Roman, and George Varghese. An algorithm for message delivery to mobile nodes. Draft, November 1996.

[10] Gruia-Catalin Roman, Rose F. Gamble, and William E. Ball. Formal derivation of rule-based programs. *IEEE Transactions on Software Engineering*, 19(3):277–296, March 1993.

[11] Gruia-Catalin Roman and C. Donald Wilcox. Architecture-directed refinement. *IEEE Transactions on Software Engineering*, 20(4):239–258, April 1994.

[12] Mark G. Staskauskas. Formal derivation of concurrent programs: An example from industry. *IEEE Transaction on Software Engineering*, 19(5):503–528, May 1993.