27 January 1981

Automatic Coordination and Processing of Electronic Forms in TLA (an Intelligent Office Information System)

by

Oscar M. Nierstrasz

A thesis submitted
in conformity with the requirements
for the degree of Master of Science.

Department of Computer Science
University of Toronto
Toronto, Ontario
Canada
M5S 1A7

Abstract

Many procedures for processing paper forms in offices are well-defined, regular and mundane. This thesis discusses the design and implementation of a facility for specifying automatic procedures in an electronic office forms system, called TLA. A high-level description of a "working set of forms" is used to trigger the automatic procedures. The algorithm which establishes the triggering is presented in detail.

Acknowledgements

I wish to express gratitude to my supervisor, Dr. Denis Tsichritzis, for his foresight, insight and hindsight, and for inspiring an congenial atmosphere within the research group. I also thank Dr. Fred Lochovsky for pointing out many improvements to the manuscript.

Tweedledum, my research partner John Hogg, I thank for his patience, cooperation and intelligence.

Christine Cheung and Simon Gibbs I thank for producing modular software that required minimal changes to accomodate TLA.

Finally, I thank Bob Hudyma, Ivor Ladi, Fausto Rabitti and Marc Graham for their comments and suggestions.

Financial support was gratefully received from the Natural Sciences and Engineering Council of Canada.

Table of Contents

Table of Figures

1. Introduction

Traditionally computer applications in business have been approached with large machines in mind. Since technology made it possible to purchase more than twice the raw computing power for less than twice the money, it was most economical to invest in the largest machine possible and to solve problems given the knowledge that many applications might be sharing one system.

Advances in computer technology, however, have produced small, cheap machines with computing power equivalent to their larger, older and more expensive predecessors. This increased availability of computing power has opened the field to many applications for which the cost was formerly too prohibitive. Furthermore a great deal of interest has been spurred in distributing large applications across many small machines. Although there will always be problems which are best (i.e. most cheaply) solved in batch mode on a single machine, the growing demand for widely distributed, real-time systems has initiated a great deal of research into developing computer systems running on networks of small machines.

The regularity of many of the more mundane office tasks and the regular structure of paper forms makes the office an ideal environment to model on such a network. This thesis assumes that the office functions which would be useful to model concern modification of forms and their routing through the system in some coordinated way. It assumes furthermore that these functions depend on information found on a collection of related

forms, and that this notion of "a working set of forms" is crucial to the design of forms procedures.

The office is modelled as a number of workstations capable of creating and modifying forms and mailing them to one another. Automatic forms procedures running at any given station watch the forms being routed through that station, and when a working set of forms is recognized, the forms are locked, processed and rerouted according to the specification of the procedure.

Chapter 2 attempts to motivate this particular view of the office and describes a prototype office forms system into which this notion of automation was built. Chapter 3 discusses the design specifications, user requirements and the user interface to the automated forms system. The powers and limitations of this approach to automation are discussed in terms of a "good office design" and how various automatic procedures running at different workstations should cooperate.

Chapter 4 surveys some of the implementation concerns, in particular the form-gathering problem, and how a set of working forms can be recognized. Chapter 5 outlines possible useful extensions and some unsolved problems, and attempts to draw some conclusions from the implementation and research.

2. Automation in an Electronic Office

Much work has been done in the last few years in the field of office automation. Some systems provide a facility for some user-specifiable, non-procedural automation, but these are not form-oriented systems. Electronic forms systems with any degree of automation are very application-dependent, not very flexible, and are based on "intelligent forms" rather than intelligent programs that manipulate stupid forms. This chapter will motivate TLA it in the context of previous work.

2.1. Motivation

In recent years computers have demonstrated their usefulness in the fields of information processing and data management. Regularity of information and regularity of processing are characteristics that lend themselves well to computer applications. The greatest experience with data processing, however, is batch processing on large machines. Interactive systems on networks of small machines have only recently begun to gain popularity, now that computer technology has been able to reverse the rule of "the bigger the better".

Furthermore, many applications are real-time dependent, and cannot benefit from the simplicity and efficiency of batch processing. Those which do not depend on data being centrally situated can take advantage of networks with distributed databases. The office can be such an application: the most familiar object in the office for storing data, generating work and communicating information is the form. Forms which are being processed sit on a worker's desk, not in a filing cabinet in the basement. One may model form files using private databases which belong to workstations distributed across a network.

As data objects, forms are very regular and thus ideal for representation in a data base. The filling in of a form can imply many side effects, some of which can be easily handled by computers. Even on the face of a form, data constraint checking and simple calculations are more conveniently performed by a machine than by a human. Furthermore, a newly created form often

has immediate implications for which a computer can provide a real-time response. If a shipping form is filled in, for example, a customer's account should be debitted.

The demand for distributed systems capable of supporting real-time applications will grow as more business functions become computerized and as the desire for compatibility of information and ready availability of computing power increases. Manually entering data from a paper form into a calculator or computer in order to perform some trivial calculation is very expensive if the information must be entered more than once. If the form is entered initially into an electronic forms system, then the computer can perform the trivial calculations without having to be driven by a human.

Reducing the quantity of printed paper is just one benefit of an electronic forms system: computers cannot (or should not) "lose" forms or accidentally bury them under a pile of paperwork; electronic forms can be quickly retrieved; the computational powers of a computer can be exploited to provide "intelligent" forms or forms systems; communication is fast -- forms can be "mailed" quickly between workstations; and information about forms in the system can be more quickly compiled than by visiting office workers' desks to determine the state of an office.

Even a manual electronic forms system -- one with no built-in intelligence, in which the system merely performs user requests to manipulate forms -- has great value, but there is much potential for automation of many aspects of a paper office.

Reminders concerning work to be done and checks on the relative speed of form flow throughout the system can improve office efficiency or aid in the analysis of the distribution of work in the office. The system can easily detect whether a form is "stuck" in the system if the time spent so far at a particular workstation is considerably greater than its usual turnover time, and then take steps to get it "unstuck". More sophisticated analysis can be done given some detailed knowledge of how the system is expected to run. Legality assertions on fields, automatic filling in of fields, automatic creation of accompanying forms and side effects such as notification can also be performed when a form is created. Automatic routing of a form through the system can be achieved by providing mail workstations that scan forms and decide whom to distribute the work to.

The corresponding research problem is not to develop a system which will provide these facilities as "features", but rather to generate a framework in which most offices may be modelled. A useful system must be able to capture a wide variety of office activities in a way that allows one to examine the interaction between them. The problem, then, is one of reduction -- to model simply and elegantly the interesting aspects of the office within a uniform framework -- not to provide an overblown system with enough features to satisfy every request but that of manageability.

2.2. Office Systems Today

Computer applications in business have traditionally been exactly that: applications to particular problems. Even systems that are developed are usually oriented towards a particular problem. Businesses interested in mechanizing or automating some aspect of their operation are not so concerned with developing a system which could also be used by the company across the hall or down the street. The best example of a general business-oriented product is COBOL, not because it is a system, but because it is a language which is actually readable, that is, it may be (relatively) easily understood by non-computer specialists (assuming that programmers do not purposely undermine its readability).

The prospect of providing a system which everyone can use is discouraging: attempts to provide general systems often result in cries from users for added features. Attempts to satisfy those cries can result in large systems with more "bells and whistles" than generally useful software.

Granted that it be difficult to abstractly capture business operations or even office operations in a single system, there have nevertheless been several attempts in the past few years to model them. A short survey of the approaches used will help place this thesis in perspective.

Officetalk-Zero [ELLI79, METC76] is a form-based computer system which incorporates the notions of workstations, in/out trays, dossiers and private databases implemented on a network of small computers.

The most attractive feature of Officetalk is its interface. Every effort was made to make a workstation resemble an office worker's desk: "...the user can shuffle paper, read mail, or read previously filed documents without touching the keyboard." [ELLI79, p.7]

Officetalk is an attempt at mechanizing office functions rather than automating them, however. Some intelligence may be built into individual forms, but there is no way of specifying a procedure which operates automatically upon identifying some collection of forms, nor is there any way of describing a "normal" passage of forms through the network of workstations.

At MIT [ATTA80] a form-based system resembling Officetalk in many ways has been implemented. Here forms can have a great deal of intelligence, but no procedure specification facility exists. Fields which are functionally dependent upon other fields may, for example, be automatically filled in.

OFS [CHEU79, CHEU80, GIBB79, GIBB80], the Office Forms System developed at the University of Toronto bears much resemblence to Officetalk. It is also form-based, incorporates workstations as abstractions of desks, and allows users to manipulate electronic forms much as they would paper forms through a simple interface. Since OFS was used as the underlying forms manipulation system upon which TLA, a form procedure automation system, was built, much more will be said about it in a later section.

QBE (Query by Example, [ZLOO75]), OBE (Office Procedures By Example), and SBA (System for Business Automation, [DEJO80]) are

related products and projects at IBM which inspired some of the superficial interface decisions in TLA.

QBE is an elegant non-procedural interface to a relational database. Relations are represented as tables, and queries to the system are presented definitionally. If one wishes to find all tuples in a database, for example corresponding to employees who make more than $30,000, one simply enters ">30000" in the salary column and "P." (for print) in the name column of the "employee" table.

To accomplish joins, one places an example (variable) in the corresponding columns of two tables. Smith, for example, would appear in the name columns of both the employee and manager tables to express that one is interested in employees (making more than $30,000) who are also managers. This non-procedural approach to database queries is extended in natural ways to many more operations than those mentioned here. Some unconventional queries, it is admitted, are extremely difficult to express in this way, but for most purposes the approach yields queries which are intuitively easier to grasp than those expressed in terms of more traditional database manipulation languages.

OBE and SBA both extend the principles used in QBE to the area of procedure specification. One is able to have procedures triggered automatically upon some condition holding true over the database. Triggers may also include time conditions. Furthermore, a hierarchy of triggers may be defined so as to capture a flow of actions witnin a procedure given that certain conditions

hold under specific circumstances.

Although OBE and SBA deal with tables more than with individual tuples, it was felt that many of the principles apparent in these systems could be applied to forms, which are encoded as tuples.

SCOOP (System for Computerization of Office Processing, [ELLI79, ZISM77]) was an actual attempt to specify automatic office procedures using augmented Petri nets [PETE77]. The approach models form flow in the context of the entire system and so must in general capture an enormous amount of detail. The kind of information captured by Zisman's model is at the level of interest more appropriate to an office analyst than to an office worker. The semantics of an office procedure are described within a single model, however, and the interaction of tasks and events is captured rigorously, rather than inferred from a collection of related but physically independent application routines.

OSL (Office Specification Language, [HAMM79]) also is an approach to modelling office procedures, but is not intended as an implementable computer system. It is instead to be thought of as a management and design tool used for description, specification and analysis of office procedures.

Finally, BDL (Business Definition Language, [ELLI79, HAMM77]) is a form-based office automation language. Although it is well-suited to office applications, it is a specialist's tool that is used for defining office procedures at a very high level.

It is again not appropriate for the day-to-day operations of an office worker.

2.3. Design Considerations

This section discusses a number of issues one must consider in developing a system which will support automation. These issues include (i) the interface, (ii) the problem of dynamically altering the nature of the automation, and (iii) the degree of automation (or the allowable complexity of triggering procedures).

Any computer system which is to gain acceptance by non-technical workers must provide an interface which neither intimidates new users nor frustrates experienced ones. If the system is extremely complicated, then it must be intelligent enough to aid and instruct new users when necessary without interfering with those users who are already familiar with its intricacies and thus require no spoon-feeding. Otherwise it should have an interface which is simple enough for a new user to learn quickly, yet expressive enough for a useful range of application. In the latter case, an extremely high-level, non-procedural language with simple, intuitive semantics would be required, and if the range of functions to be captured is small enough, such a language is even feasible.

Automation in an electronic office can be static or dynamic: in a static system, software packages are written for each application, and new applications must be written to be compatible with existing modules. It is "fixed" in the sense that features cannot be arbitrarily enabled or disabled. The behaviour of a static system is predictable if the office it models is well-

understood and the model is accurate. Modifications to the design and flow of the office model may well be painful, but there is no built-in restriction to the "fanciness" of features that may be built into a new application.

A dynamic system would provide a single software package which interprets a high-level language tailored to the operations and features available in the electronic office. The office is modelled by the high-level description. If the menu of operations and features is small and the nature of the automated functions is highly modular, then the specification of automation in the high-level language becomes very simple. In addition, changes to the design and flow of the model require only small changes to the high-level specification, not to the software package. Of course, additions and modifications to the set of available features and operations on the data objects is again painful, but now a wide range of variation is possible at little cost, where before there was none.

The meaning of an "automatic procedure" depends on the generality of what one wishes to model, or wishes to be able to model. At some point it becomes too difficult or expensive to have computers do the work, and that is where the human interface appears. How much can be automated depends on how much intelligence should be built-in, and how much intelligence can be bought through some kind of interface to library routines or application programs.

Whatever the degree of automation be, all activity is ultimately initiated from outside the system. Automation must be user-driven at some stage, and hence some manual activity must take place. If those activities which may be automated duplicate some or all of the manual activities, then questions of conflict and control must be answered. If a manual and an automatic procedure conflict, a decision must be made about which is to be given control. Certain activities must be locked, and facilities for graceful recovery must be provided in case an executing procedure must be aborted because of such a conflict. What constitutes an "error" determines what sort of recovery must take place. In case of errors or partial (user-assisted) automation, users must be able to interact with automatic procedures to enable them to run to completion. One must identify the data objects, the security restrictions on them, the range of operations which may be performed on them, and the degree to which manual and automatic procedures may conflict with respect to them.

If the specification of automatic procedures is in any way complicated or subtle, there should be some way of debugging the procedures, or else the system should be able to detect anomalies arising from any badly-written ones. Determining what are to be considered "anomalies" is no simple task. One may choose to accept anything as being valid, but if not, then the detection of anomalies is divided between potential ones -- those which might arise given a particular flow of data -- and thus require some analysis for their detection, and the actual ones, which may be

detected by observing the performance of the system. In the second case, the symptoms rather than the causes are reported. If anomalies are detected at run-time, some facility for halting the system or parts of it should be available.

The domain of automation determines the modularity of procedures. One must decide what limit to place upon the generality of those conditions which may trigger an automatic procedure. If the complexity of those conditions is too great, triggering may be either too expensive or too difficult to implement. The state upon which an automatic procedure is triggered may range from the field values of a single form, to the set of forms belonging to a workstation, to all sets of forms at the workstations in the entire network. Intelligent forms are easy to implement if the intelligence is restricted to the information found on the form and possibly some readily accessible repository of related information. If intelligence is restricted to a workstation, automation is still feasible, but management of the data objects, be they forms, tables, relations or whatever, becomes more complicated, since transfer of data between workstations affects the state of the workstation in an unpredictable way. If the intelligence applies to the state of the system as a whole, automated procedures can be very expensive, especially if the system is implemented as a network of small machines, and any transaction on any machine, or any communication between workstations on the same or different machines alters the state of the system.

If office workers restricted their attention only to their desks and nobody paid attention to what was happening at the

global (managerial) level, then work might get shuffled around from desk to desk forever without anyone noticing. If the domain of automatic procedures includes only a small part of the system, the resulting modularity of automation could cause the system to behave unpredictably in the event of an unfortunate configuration of automatic procedures throughout the system. In such a case where automatic procedures do not "know about one another", but can produce output which may be consumed by other automatic procedures, analysis guaranteeing any form of "correctness" may be very expensive to do on-the-fly by the system whenever a new procedure is written. It would otherwise be the responsibility of the users or some administrator to ensure that the configuration "correctly" models the real office. An ideal system for automation should provide a facility for observing or controlling activity on a global scale.

Furthermore, one must decide who is to write the automatic procedures. If a single person can be trained to manage the automation, then the interface can be fairly rich and complicated. If individual users are permitted to write procedures running on their own workstations, then the interface must be simple enough that users can feel as confident writing the procedures as they do using the system. One could argue that control over the office design and form flow is lost once users are able to specify the automation -- a badly-written procedure or an unfortunate combination of cooperating procedures may destroy balance in the system -- but it may be possible for the system to be able to detect such anomalies, and, more importantly, it may

be preferable to let the control of automation lie where the local implications are best understood.

Many of the issues pointed out in this section have no clear solution. Some are obvious cases of situations that would be approached differently depending on the kinds of applications one wished to run on a system supporting automation. Others are research topics ("correctness", for example) which will be treated more in the years to come. The approach that was chosen for TLA will be discussed in the next chapter.

2.4. TLA

The TLA project was conceived as an effort to introduce automation into a prototype office forms system (OFS) rather than as an attempt to build a fully automated system from scratch. OFS allows its users to perform a small set of operations on one type of object: the electronic form. TLA concentrates on one aspect upon which to base automation -- that of communication. Actions are associated with the flow of forms through the system, and so can be triggered automatically when forms or combinations of forms arrive at particular nodes in the network.

Office workers are expected to make decisions and do work on the basis of the condition of their desk rather than upon that of the whole office. Thus automatic procedures are triggered upon conditions local to the workstation, such as the arrival of mail, rather than upon the state of the system. This restriction considerably lessens the cost and complexity of the most general triggering condition, whose domain includes not only the user's machine but the entire network, yet leaves those conditions general enough to solve interesting problems. The restriction is also in keeping with the OFS principle of insulation of private workstations: users cannot produce side effects outside their own workstation. The only information available to them concerning the state of the system is the trace of a form's passage through the system.

Only by releasing control of a form and mailing it to another station may one indirectly affect the rest of the system,

given that the form's arrival and content have a meaning understood by the receiving station. If the form is not "understood", it is never processed by an automatic procedure and waits forever. The specification of an automatic forms procedure should capture this meaning in terms of preconditions and actions.

2.4.1. OFS: An Office Forms System

OFS [CHEU79, CHEU80, GIBB79, GIBB80] is an electronic forms management system written in the C programming Language [KERN78]. OFS provides an interface to MRS [HUDY78, KORN79, LADD79], a small relational database system also written in C. OFS translates its data objects, which are images of paper forms, into tuples of an MRS relation.

An OFS system consists of a set of stations distributed over a number of machines in a network. Each user has a private database in which the form tuples are created, stored and modified. A user may only manipulate those forms which he "owns" in the sense that they reside in his database. Communication and interaction between stations is achieved by allowing users to mail forms to one another.

The only automatic form processing that OFS will do occurs if a form is mailed to a special automatic station -- a station which periodically reads its mail and submits the forms as input to an application program written in C. Such application programs must be compiled since there is no facility in OFS for interpreting files of commands. The programs must also be written so as to preserve compatibility with OFS or else unpredictable and undesirable side effects may be discovered. Consequently, an OFS programmer must be equipped with a great deal of knowledge of the inner workings of OFS (as was required for the transformation of OFS into TLA).

2.4.2. OFS Operations

A distinction is made between form types, form blanks and form instances. A form blank is simply the form template used to display a form instance. A form instance corresponds to an actual tuple in the database. Its fields may have values assigned to it, and it always has a unique key assigned at creation by the system. A form type is the specification of field lengths and security types corresponding to the form blank and its associated relation. A form file is the relation used to store all forms of the same type belonging to a station. The collection of form files for a station is a form database. Figure 1 shows a form blank and form instance for the form type called "ORDER FORM". Note that some fields of the form instance need not have values associated with them, although the key field must.

Form fields may be of 6 different security types. Manual fields of type (1) may be inserted or modified at any time, (2) may be inserted at any time but not modified, or (3) must be inserted at form creation and never modified. Automatic fields are (1) key fields, always the first field of a form, (2) date fields, and (3) signature fields bearing the station's name if the preceding field is filled in. There is no facility for restricting the range of values a field may accept, other than the field length.

Form files may be accessed by MRS, with the feature that field securities and the unique key condition may be ignored. As

ORDER FORM

KEY: _____ _____

Customer number: ____ Customer name: _____
Item: ____ Description: _____
Price: _____
Quantity: _____
Total:_____

An order form blank

ORDER FORM

KEY: 00001.00000

Customer number: 5184 Customer name: Denis the Menace_
Item: 0001 Description: Office Forms System_____
Price: 200_____
Quantity: 1_____
Total:_____

An order form instance

Figure 1 Form blanks and instances

a result, the MRS interface is not meant to be available except to privileged users.

Form operations are creation, selection, and modification. Forms may also be attached to dossiers: lists of forms which are not necessarily of the same form type, but which have something in common that the user wishes to capture.

Forms may not be destroyed, although they may be mailed to a "garbage station" which conceptually shreds the electronic form, and may in fact either archive or erase it depending on the needs of a particular application. Instances are unique, and must

always exist at exactly one location in the system either in some form file or waiting in a mail tray. Forms may be mailed from one station to another, but they must wait in a mail tray and be explicitly retrieved in order to be placed in the receiving station's form file. Copies may be made of forms, but they are assigned a unique key consisting of the key of the original form together with a system-generated copy number distinguishing it from the original. Copies may be modified so that they no longer resemble the original.

2.4.3. OFS Configuration and Implementation

The OFS network consists of one host machine connected to a number of satellite machines. Communications managers running on each machine pass messages between the host control node and its satellites when a form instance is either created or relocated. The host manages all information related to form keys, copy numbers, account names and passwords, and mail trays. Forms, for example, may not be created by a satellite station unless the host sends it a unique form key, so the host must be running for any useful work to be done at a satellite station. The communications manager of the host machine passes messages to a "HOST" program, which processes the messages and sends information back when required.

For a form to be mailed from any station to any other station, the form tuple must be deleted from the sending station's form file, and sent to the HOST program which inserts the tuple in a mail tray relation. The receiving station must also send a message to the HOST in order to read its mail. Each form relocation is logged on the host machine so that forms may be traced or located.

Users may mail forms anywhere, but they are not inserted in the receiving station's form file until the owner requests it. The HOST holds the mail in the mean-time, thus guaranteeing the users' control over their form databases. Communication with the HOST is therefore necessary even when both the sending and receiving stations are situated on the same machine. Mail must

be retrieved before any further action is performed with it, even if it is to be immediately shipped away again.

OFS is implemented as a collection of overlay modules, each of which handles a different set of actions. Mail-retrieval, tracing and form-creating, for example, are each implemented as independent modules. Although it is necessary to switch modules in order to perform different operations, this is fairly fast and painless. The individual modules are guaranteed to fit on a small machine, whereas the entire collection together might not.

The form databases are implemented as UNIX directories [RITC78] owned by a privileged form administrator. Users may only manipulate their databases by executing the OFS program, thereby gaining access rights to the directories which OFS recognizes as belonging to them. Privacy of form files and consistency within the system may thereby be ensured.

The only interface OFS provides with application programs is through the mechanism of an automatic station which regularly reads its mailbox and interprets the form tuples as input to some locally-written program. Examples are the print station, which produces a hard copy of the form, and the garbage station, which conceptually shreds or archives the form.

3. TLA Design

Since TLA was built on top of an existing forms management system, the range of possible features for specifying automation was fortunately limited, but still so large that only a tiny subset of conceivable features could be considered. Restricting this range to a useful but implementable subset was motivated largely by what was currently possible in OFS.

Compatibility with OFS was also an important concern, partly to avoid major changes to the existing code, but also to simplify conversion of any existing OFS system to one that supported TLA.

The user interface had to be very simple, and the facilities available had to be easily understood since, as described in an earlier section, our object was to provide a dynamic, high-level automatic forms procedure specification tool, rather than a static system for a particular set of applications. TLA had to be as easy to use as OFS.

The sections in this chapter discuss the motivation behind some of the design decisions and some of the consequences. The last section covers some of the implications concerning "good office design" in terms of TLA, and addresses some problems not solved by the system.

3.1. Design Specifications

Compatibility with OFS was maintained in TLA. Changes to code and the internal representation of an OFS system were mostly additions of modules and UNIX file directories. Where existing files and code were modified, compatibility was maintained, so that OFS would simply ignore the added TLA features. Conversion costs from an OFS system to one that supports TLA are negligible, and any TLA system could be run with the OFS subset.

A set of features was chosen to study the design and implementation issues of a reasonably useful but unembellished automatic forms system. A number of assumptions were made about the meaning of a "forms procedure", especially within the context of OFS, and some features were discarded as being beyond the scope of such a small scale project.

Simplicity of design for the sake of programming and application was a major consideration. The user interface is presented in terms of objects the OFS user is already familiar with: specifying operations within a procedure corresponds closely to their manual counterparts in OFS. A user who is editing an automatic forms procedure manipulates "sketches" of forms -- form-like objects that represent the forms that the procedure will eventually manipulate. The same form template which OFS uses to display form instances is used quite differently in TLA to describe preconditions and actions in office procedures. The specifications are non-procedural and are virtually syntax-free, except for pattern matching and field-referencing conventions.

Since the language is presented in terms of form-like objects that the user is presumably already familiar with, there is not much the OFS user needs to learn in order to use TLA.

TLA does not assume any knowledge of the system other than what is available to the user in his form file or his mail tray. This corresponds to the notion in OFS that users can only manipulate the forms that they "own". Anything happening outside their own workstation does not concern them. This motivated restricting the domain of automation to that of individual workstations. If procedures are allowed to know only about the state of a workstation and the forms in its form file or mail tray, then the state of the system is a variable that does not concern it. The complexity of determining when to trigger a procedure is thereby considerably reduced.

An automatic procedure is meant to capture the notion of an office worker collecting forms at his (or her) desk until a "complete set" is compiled, processing those forms, and then filing them or sending them on their way. Processing of the collection of forms may cause forms to be modified or new forms to be added to the set. Reference tables and calculating tools are made available through an interface to some local library of application programs.

The other aspect of automation supplied by TLA is that of "smart forms" which automatically fill in certain fields of a form with previously filled-in fields as arguments. The domain here is that of the form alone, so triggering takes place when-

ever a form is created or modified. "Smarter forms" with fields that change value depending upon time conditions, the state of the system, or any other variable, were not tackled, although some "smarter form problems" can be solved with TLA's automatic procedures.

Automatic procedures have preconditions and actions, but no postconditions in the usual sense. Satisfying all preconditions guarantees the successful completion of all actions. There is only a very limited sense in which a procedure may "fail" -- if it is never triggered, for example, because missing forms do not arrive. Postconditions may be interpreted either in terms of which actions are performed dependent on earlier actions, or in terms of the preconditions of another automatic procedure to which control of the forms is passed.

Lastly, since automatic procedures presumably run concurrently with the manual functions of the users, conflicts could arise over the form manipulations. Forms being collected by an automatic procedure could be modified or shipped away manually, or even "stolen" by another automatic procedure. When a complete set of forms is gathered for some procedure, then, it has to be grabbed and temporarily "removed" from the system until it is processed, but there must be no possibility of the forms disappearing forever.

3.2. User Interface

The specification of an automatic procedure in TLA bears some resemblence to SBA/OBE [HOGG81]. The precondition segment of a procedure is like a QBE query with forms instead of tables as the data objects. The action segment is similarly intuitively natural. The appearance of a value in a field indicates, for a precondition that that value is to be matched, or for an action that that value to be inserted in the field.

A TLA procedure is a collection of "sketches", where a sketch resembles a form, but is to be distinguished from form blanks, form types or form instances. A form sketch, or a "sketch of a form", indicates either a request to the system to find "a form that looks like this", or indicates a request to modify a form that has already been retrieved. A form sketch, then describes a form instance before or after processing by the procedure, and does so in the medium of the same form blank which is the template for the form instance being described. Actions and preconditions which do not refer to information found on the face of a form are specified by sketches of "pseudo-forms": for example, the condition that a procedure process only forms coming from user "joan" must be indicated on a special "source sketch", a pseudo-form that describes another form.

There is no facility for specifying the order in which forms in the working set should arrive, or the order in which actions be performed. TLA merely ensures that the procedure be logically consistent. The specification is non-procedural, in the sense

that the user indicates what forms are to be collected, and what is to be done with them, but not how they are to be collected or how the actions are to be performed.

3.2.1. Preconditions

Preconditions in TLA are what, when and where. The working set of forms is perhaps the most obvious thing one would wish to specify. Furthermore, one may wish to refer to forms that come only from certain workstations, forms created or modified only by oneself (i.e. they already reside in the user's workstation rather than in the mail-tray), or forms that have just been processed by another automatic procedure running at the same workstation. Lastly, one may wish to run a procedure only at certain times or ranges of times.

Obviously the last two conditions refer to external information not found on the surface of any of the forms in the working set of forms defined for the procedure. As such they require pseudo-forms (forms that have no meaning outside of automatic procedures) to capture the restriction if uniformity of the interface is to be maintained. The source restriction of a form is then specified by filling in the source sketch pseudo-form logically linked to that form's precondition sketch.

Form sketches are used to capture the restrictions referring to values that appear on the face of the forms in the working set. Local restrictions are constant field values, sets or ranges of values, and relations between values of the fields on a given form. The local restrictions refer only to the values appearing on the face of a single form in the working set. If TLA determines that a given form satisfies the local restrictions (including the source condition) for some sketch in some

automatic procedure, then it notes that information and attempts to match that form with other forms to obtain a complete working set for that procedure.

Figure 2 is an example of a precondition sketch instructing TLA to watch for order forms requesting "Tin tear-drops". Since this information can be found right on the order form, it is a local precondition. A sample procedure including such a sketch might perform the single action of returning a form that says "We stopped making those things years ago!".

ORDER FORM

KEY: _____

Customer number: ____ Customer name: _____
Item: ____ Description: Tin tear-drops
Price: _____
Quantity: _____
Total: _____

Figure 2 A precondition sketch

Global restrictions on the working set of an automatic procedure are the join conditions between values of fields appearing on different forms. In general one expects all the forms in a procedure's working set to be linked by certain common field values, such as account numbers. Equality joins, therefore, are probably adequate to model most applications of automatic procedures.

Figure 3 shows how a link is made to find an inventory form for the item requested on an order form. Each sketch in a procedure has a name assigned by the user, and this name ("order" in the example) is appended to the field name if the field is to be referenced within a sketch for a different form in the working set [1]. Note that one could equivalently have placed the restriction "=item.inv" in the item number field of the precondition sketch for the order form.

INVENTORY RECORD

KEY: _____

Item: =Ite Description: _____

Price: _____

Quantity in stock: _____

=Item.order

Figure 3 A global (join) precondition

One may wish to restrict the source of mail being processed by one's automatic procedures. If the accounting department receives an order form from the ordering department, that may be interpreted as a request to forward a customer's credit-rating and balance to the warehouse so that the order may be approved. If, however, the order form arrives from the warehouse, that may indicate that the order has gone through, and that the customer's

[1] Since the precondition is longer than the space available for the corresponding field, it appears on the screen in an overflow region. In future examples, overflows will appear on the form to improve readability.

account should be debitted and an invoice mailed out. (The author makes no claim that the cited example be realistic -- merely that it illustrate a point.) Figure 4 shows an origin pseudo-form sketch for such an application. Forms may thus be processed differently depending upon their point of origin.

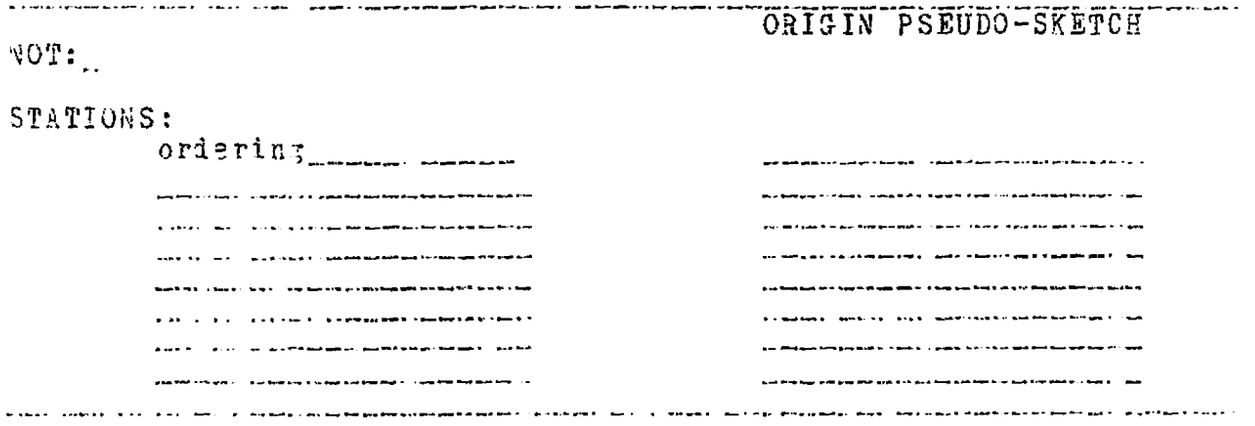


Figure 4 An origin pseudo-sketch

3.2.2. Actions

Actions which do not concern themselves with field values must similarly be expressed via pseudo-forms, but all form-modification actions are indicated on form-sketches. In general, every form manipulated by a forms procedure has a precondition form sketch, an action form sketch, and one each of precondition and action pseudo-form sketches.

The action form sketch indicates all insertions and updates to the form. The values to be inserted may be constant values (eg. an authorization), copied field values (presumably from another form in the working set), or possibly function calls (calls to application programs). Since security type 3 fields in OFS may be modified at any time, one needs to be able to distinguish between the original and the updated value of any field. A field which must be copied to another form may itself be modified, and the wrong value must not be used. Furthermore, the function calls may access both the original and updated values of fields, and, in fact, the original value of a field will often be one of the arguments to a function call uplate to that field.

The action sketch of figure 5 illustrates several features: the price of an item is filled in by copying it from an inventory form; a program called "mult" is called to calculate the total; and the original value of Quantity is accessed whereas the updated value of Price is used. Note that the symbols "#", "?" and "!" are used to access functions, original and updated field values. If none of these is used, a constant string value is

inserted.

ORDER FORM

Customer number: ____ Customer name: ____ KEY: ____
Item: ____ Description: ____
Price: ?Price.inv
Quantity: ____
Total:#mult !Price ?Quantity

Figure 5 An action sketch

Some analysis needs to be done to ensure that every updated field ultimately depends only upon values originally available on the working set of forms -- it is clearly incorrect to update each of two fields by copying over the updated value of the other. If the Price field of the order form were updated to "!Price.inv" and the Price field of the inventory form were updated to "!Price.order", then clearly no order of execution could make sense of the request.

Field securities must be obeyed: procedures that create forms must fill in certain fields, and procedures that modify forms must only modify fields with an appropriate security type. If a procedure modifies a field of type 2, then by implication the modification is an insertion since no value inserted into a field of that type may be modified, and so there is an implied precondition that that field be empty in the retrieved form. Of course implied actions must also be evaluated if a procedure modifies or inserts a field which is an argument to an automatic

field.

Follow-up actions performed after all forms are modified include copying of forms, attaching forms to dossiers and shipping forms to other workstations. Each of these is expressed on a pseudo-form sketch. A weak sort of postcondition is available by employing a function call to decide the shipping destination, the number of copies to be made, and so on, but branching within a procedure, and other general postconditions can only be achieved by cooperating forms procedures which accept different cases of the working set of forms.

If the processing of an order causes the quantity of an item in stock to dip below a certain acceptable level, for example, then one may wish to send a memo to a manager advising him (or her) that production on the item should be increased. The procedure which processes orders, however, is incapable of conditionally producing this memo as a postcondition to inventory update. It could unconditionally produce such a memo and then functionally decide to mail it either to the manager or to a garbage collection station. A cleaner approach, though, is to have a separate procedure which searches for low inventory items, and then fires off the memo.

The advantage of this approach is that individual tasks are clearly identified -- automatic procedures are uncomplicated and completely devoid of any control flow requiring careful analysis or debugging. Furthermore, the implementation suffers no added complexity because of the interpretation of postconditions as

separate procedures. The low inventory checker, for example, is only invoked when an inventory form is updated.

3.3. Summary

The description of an office modelled by TLA includes the workstations, the set of all forms, the work which is to be done on the forms at each workstation, the coordination of forms as they flow through the office in some organized fashion, decision points where different actions take place, and some notion of the possible set of "correct" life histories of any form. We have assumed that this description may be modularized to such an extent that some collection of small, localized procedures running at the various workstations captures enough detail that procedures need not "know" about one another nor about the state of the system at any time. Deciding whether this collection does in fact model what we wish it to model, however, requires an analysis in context, and some understanding of the possible ways in which procedures interact. At present we assume that the owner of each workstation understands the local implications of any of his automatic procedures given its preconditions and actions, and that some manager understands the global properties of the system implied by the local properties captured at each workstation.

The local properties necessary to guarantee global correctness [which we at this point leave undefined] must be simple enough to be captured at any workstation by automatic procedures which are small, comprehensible, simple to write and trivial to debug. The system, as a consequence, should be configured so that desk functions can be easily localized. Multiple procedures each with slightly different sets of working forms should be

written to handle decision points. A single procedure can easily handle functional dependence such as modifying or inserting fields based on values of fields on forms retreived, or shipping a form to a station whose name is functionally determined from the form instances. Multiple procedures, though, are necessary if a decision is made, for example, whether to create a new form of type A or type B, since a different action sketch would have to be included in either case.

TLA provides facilities for specifying automation at two levels of granularity: that of the workstation and that of the individual form. The highest level of granularity which one would be interested in is that of form flow in the system. That form flow is in fact defined by the collection of TLA procedures in the system is a side effect. One has no real control in TLA over the flow, nor is there a simple, intuitive way of analysing the procedures to determine what possible histories a form may have in flowing through the network. Ideally one would like to be able to specify for a given form or collection of forms some notion of correctness that encapsulates legal histories, side effects, and so on. Such a specification would be used not as a procedure which describes what must be done with a form, but rather as a guideline which either (i) is applied heuristically to form instances to warn an administrator when a form strays from its path, or (ii) is applied analytically to form procedures to warn an adminstrator when a combination of automatic procedures implies a potential anomaly or deviation from correctness.

There is no attempt in the implementation of TLA to deal with this topic. It is seen as a subject of considerable depth and complexity which may not necessarily have an ideal solution, but should be approached as a rich area for further research.

4. TLA Implementation

Because the working set of forms for an automatic procedure must be gathered over a period of time, the information contained in that procedure's specification is not all needed at once. The results of any analysis done during the form-gathering process must be recorded for later use. One is not interested in the actions of a procedure, say, until a working set of forms is identified. Translation of the non-procedural sketch specification must therefore be analysed and translated so that the information needed at various stages in the interpretation of the procedure can be retrieved as painlessly as possible.

Bookkeeping during the form-gathering phase is outlined, and the algorithm for identifying a complete working set of forms is described in some detail.

4.1. Translation

An automatic forms procedure in TLA is specified by a collection of sketches, and as such describes what is to be done rather than how to do it. Although the sketch representation is very convenient for the user as an aid to understanding a procedure and capturing the amount of detail which is of interest to a non-programmer, the format is wholly unsuitable for implementation. The specification must be analysed and translated for greater run-time efficiency. As much analysis as possible is done in the translation phase in order to reduce the execution time.

Since one cannot predict when the forms required to trigger an automatic forms procedure may arrive, the processing must of necessity be broken into distinct parts. The specification in terms of sketches contains information of four basic kinds: local (form) constraints, global (working set) constraints, duplicate form types, and actions. The execution of a forms procedure makes use of these four at different stages. For that reason it is convenient to distil this information from the sketch specification once at procedure definition time, and translate it into formats that require no further run-time analysis.

When TLA is notified of the availability of a form for automatic processing, it first checks whether the form matches the local conditions of any precondition sketch for that form type in any automatic procedure running on the workstation. The local conditions are comprised of the source restriction (where

the form is expected to come from) and the field constraints that depend only on information found on the face of the form. If a form does not match the local constraints of any precondition sketch, then TLA can confidently assume that no automatic procedure is prepared to handle it. Conversely, if a form does match the local constraints of one or more precondition sketches, then, whether or not a working set including that form is complete, there is always the possibility that at some time that form may become part of a working set for some procedure, given the arrival of the missing forms.

The form instance in figure 6 matches the local condition of the precondition sketch, namely that Quantity>Ø, but there may not necessarily be a global match if there is no order form with the same item number. Of course, even if there is an order form with the same item number, it may not satisfy the other constraints of its precondition sketch, whatever they might be. Nevertheless TLA notes that a local match has been made and patiently waits for the rest of the working set to arrive.

Usually, TLA will check the local constraints of a form, record its findings, determine that the form does not complete a working set, and then interrupt the precondition portion of the procedure until more forms arrive. Further processing may not occur for some time. For that reason, all local constraints (and source conditions) for forms of the same type are extracted from all automatic procedures running at a given workstation and stored in a common file. Only a single file need then be opened to check the local constraints of a given form for all

INVENTORY RECORD

KEY: _____

Item: =Item.order Description: _____
Price: _____
Quantity in stock: >0____

Precondition sketch

INVENTORY RECORD

KEY: 00001.00000

Item: 0002 Description: Three Letter Acronym____
Price: 8500____
Quantity in stock: 12_____

Form instance matching local preconditions

Figure 6 Local matching

procedures. Information which is not yet of interest rests elsewhere.

Even if a complete working set of forms conceptually arrives together, the processing of the forms is sequential, and TLA learns about each form individually. A locking algorithm guarantees that two forms cannot be processed at once at a given workstation. Generally forms will not arrive simultaneously. Thus one can expect a considerable delay between the establishment of local constraints and the evaluation of links between forms. When the local constraints have been matched for a form, TLA checks whether any link conditions between the corresponding sketch and any other sketch of the procedure are satisfied by that form and forms for which TLA has already found local

matches. Even if no new links are found, links may yet be found with forms that have not arrived. The link conditions are stored in files by procedure, since TLA will reference the single procedure for which local conditions have been matched at any one time.

If, in the previous example, TLA found an order for item #002, it would note that the link between the inventory and order form precondition sketches were satisfied by these two form instances. If the working set consisted of only these two forms, then the procedure actions would be performed. Otherwise TLA will wait until forms are found to match the remaining links of the procedure for these two form instances.

Actions are performed only once a working set of forms has been compiled -- something which need never occur, in fact -- and so actions are also stored in a separate file. TLA preprocesses procedures to check the legality of actions and to determine a legal order of execution if one exist. No further run-time analysis is required -- actions are guaranteed to run to completion.

The example in figure 7 implicitly requires that Price must first be copied from the inventory form before its value may be multiplied by the Quantity. This establishes a legal order of actions for that sketch.

Finally, a list of which sketches refer to forms of the same type is stored in another file for the purpose of checking at run time that forms matching the sketch in a procedure's working set

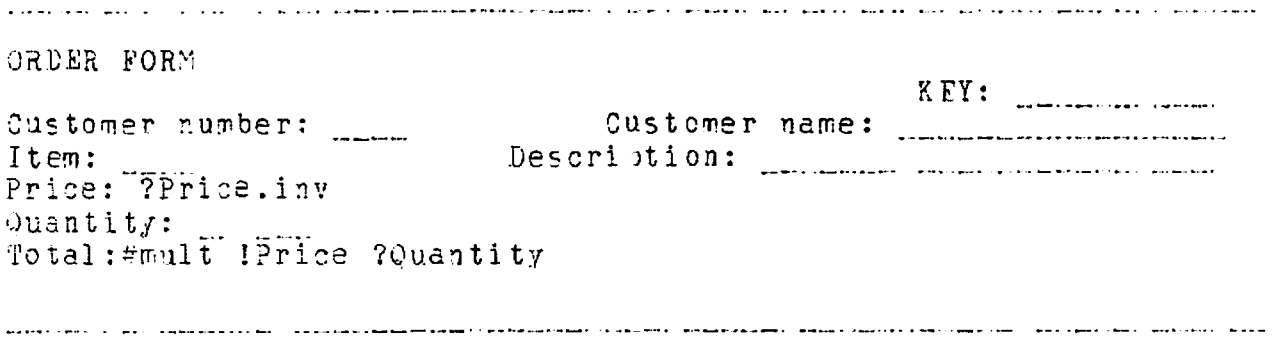


Figure 7 Ordering of actions

are, in fact, distinct. A form may match two precondition sketches of a procedure, but only if those two sketches are of the same form type. Therefore, comparison of form keys is done only between forms of the same type rather than between all forms in the working set. The comparison of all form types in the working set with each other is done before the procedure is allowed to run.

An admittedly unlikely example is captured in figure 8 which is triggered if TLA detects two inventory forms for a single item. Since there are two precondition sketches in the procedure, TLA assumes that they refer to two different forms in the working set. Otherwise any inventory form would trivially satisfy both precondition sketches and thus trigger the procedure with presumably undesireable side effects. When the procedure is written, TLA notes immediately that two precondition sketches describe forms of the same type, and thereafter performs a key comparison of those forms in any working set identified, to guarantee that they are not one and the same.

INVENTORY RECORD

KEY: _____

Item: _____ Description: _____

Price: _____

Quantity in stock: _____

Precondition sketch inv1

INVENTORY RECORD

KEY: _____

Item: =Item.inv1 Description: _____

Price: _____

Quantity in stock: _____

Precondition sketch inv2

Figure 8 Duplicate form types in a procedure

These various files drive an interpreting routine which is triggered whenever a possibility exist that a form be required for automatic processing, that is, at form creation, form modification or mail delivery. Under certain circumstances, one may also require the output of a procedure to be available as input to other procedures, or even as input to the same procedure. However, one must take care not to inadvertently cause an infinite loop, with one procedure continuously reprocessing its output, or worse, constantly spewing out new forms and clogging up the system.

4.2. Triggering and Graph-chasing

The most difficult part of running automatic procedures is the form-gathering. When a form is mailed or created, TLA must decide whether the form is needed in some working set. Matching local constraints of sketches is easily done, but the relationships between actual form instances may be much more complicated than those of the sketches in the working set. The graphs which describe the working set are discussed, and the algorithm which identifies a collection of forms satisfying the working set is given.

4.2.1. Form Images

The TLA automatic procedure interpreter is triggered upon receipt of mail, form creation and form modification. Since the last two are the responsibility of the user, triggering in these cases involves only the spawning of a new interpreting process. In the first case, however, the interpreting process is initiated by the user who sent the mail. Typically, mail will be sent from a station on a satellite machine to another, also on a satellite (assuming one host node and many satellites on the network). Mail is routed to the host machine where the form is saved and an entry in a mail tray is made. The receiving station may at any point thereafter retrieve any or all forms from the host which are listed in its mail tray.

Automatic procedures are meant to run regardless of whether the user to whom the corresponding station presumably belongs ever signs on after the procedure is written. Whether or not any automatic procedures exist at a station receiving mail, the following steps are taken: if the sending station is on a satellite machine, it sends a message to the host consisting of the contents of the form tuple and the name of the station which is to receive the mail. The host then stores the tuple, updates the receiving station's mail tray (and the form relocation log) and, if the recipient is not on the host machine, the host sends a message to the recipient's machine, consisting of the name of the receiving station.

At the recipient's machine, the interpreting process is started. It then communicates with the host, asking for images of each new form in the recipient's mail-tray. The interpreter maintains files of form images for each form available for automatic processing, and deletes the images when the forms have been processed either automatically or by the user. The images are copies of the contents of each form for use by the interpreter alone, and are stored just as forms are stored. The user, however, has no access to the images as forms -- they may not be modified, shipped away, or otherwise manipulated, and so they are not properly forms or copies of forms, but merely images of forms. [The author apologizes for the proliferation of such terms as form blanks, types, instances, files, sketches and images, but hopes that the reader appreciates the need for a semantic distinction.]

Since mail may arrive while the interpreter is running, it continues to process all mail until it discovers an empty tray. Only one interpreter may run at any time for a given station, in order to eliminate the obvious problems which would arise if two interpreters began to process forms which "belonged together" in one automatic procedure. A lock is therefore placed on the running of the interpreter for a given station.

If an order form and an account form belonging together arrived simultaneously at a station, and two interpreters were allowed to process them concurrently, then each would discover that the local preconditions were matched. However, the link between the two would be missed since neither interpreter could

yet be aware of the form being processed by the other. Alternatively, each interpreter might discover the link, but an attempt by each to lock the working set could result in deadlock.

4.2.2. Sketch and Instance Graphs

It is useful to abstract the working set of a form procedure in terms of a graph with the sketches as coloured vertices, and the join conditions, whatever they might be, as edges in the graph. The graph corresponding to the procedure specification is the sketch graph for which the form-gathering algorithm must find corresponding forms and satisfy join conditions. A corresponding graph for form instances attempts to match forms to sketches in the sketch graph with the join conditions of the working set holding between the actual forms. The instance graph generated by the forms retrieved may, in the worst case, not look very much like the sketch graph, and the correspondence must be established carefully.

Consider the precondition sketches in figure 9. A link between the account and order forms is established across the customer number, and a link between the orier and inventory forms is captured by two global conditions, one by item number and the other by quantity.

The corresponding sketch graph is shown in figure 10. Each sketch is represented by a labelled/coloured node, and each collection of global conditions between a pair of sketches is represented by a single edge.

When a form is passed to the interpreter, it first reads the file of local constraints for the forms of that type. Whenever a match is found, the interpreter notes which sketch of which procedure is matched by the form, and it enters a tuple consisting

CUSTOMER ACCOUNT KEY: _____

Customer number: =number.order
Credit rating: __
Balance: _____

ORDER FORM KEY: _____

Customer number: ____ Customer name: _____
Item: _____ Description: _____
Price: _____
Quantity: <=Quantity.inv
Total: _____

INVENTORY RECORD KEY: _____

Item: =Item.order Description: _____
Price: _____
Quantity in stock: _____

Figure 9 Precondition sketches of a procedure

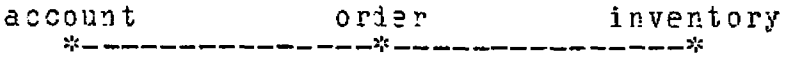


Figure 10 A sketch graph for a single procedure

of the form type, the form key, the procedure and the sketch matched into an MRS relation (called "NODE").

The file of global constraints for the procedure matched is then read. For every link concerning the matched sketch, TLA establishes whether the current form satisfies the join conditions with any of the forms previously recorded in the NODE relation.

For every new link found, TLA inserts a tuple into another MRS relation called EDGE. EDGE records the form keys, types, sketch names and procedure name of every link established.

The NODE and EDGE relations describe an instance graph with forms as vertices or nodes and links between them as edges. The vertices are coloured according to which sketch the form matches. If a form matches two or more distinct sketches in one or more procedures, it is multiply represented, once for each sketch. Procedure names partition the instance graph, since there can be no links between sketches of different procedures. For each partition we wish to match the sketch graph that describes the working set of forms for that procedure, with sketches as nodes, and join conditions as edges. Nodes are assigned a unique colour for each sketch, and the corresponding colours are used in the instance graph. An instance of the sketch graph, then, must be found within the instance graph.

Figure 11 shows the instance graph for the procedures of figure 9. Forms have been found to match each of the precondition sketches of the procedure, but there is no complete working set. The moment a working set is found, though, it is processed and thus disappears from the instance graph. Note that most of

the disconnected subgraphs of the instance graph are in fact subgraphs of the sketch graph. In the last case, however, there are two orders for a single item, and the relationship is not that simple. The first account form to complete either working set will complete the "copy" of the sketch graph to be found in the instance graph.

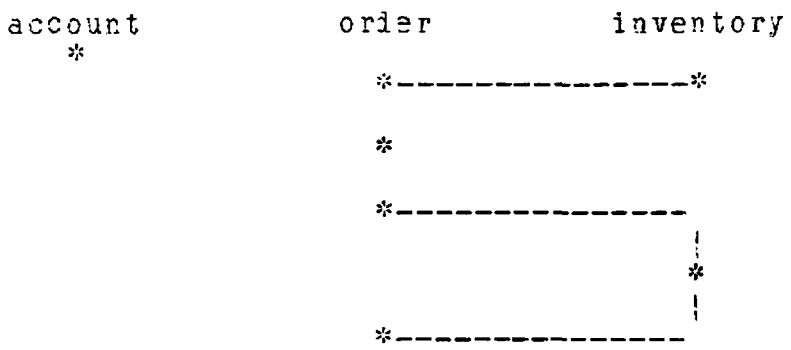


Figure 11 The instance graph for a procedure

The relationships between the forms in the working set of a form procedure are expected to be best expressed in terms of the join conditions, so the sketch graph will generally be connected. The instance graph, however, will more often consist of several partially complete working sets of forms, and so will be disconnected. The only likely situation in which the sketch graph will be disconnected occurs if one or more of the forms in the working set are uniquely identifiable within the system, independent of the other forms in the working set. A "total" form for a station, for example, is updated every time the automatic procedure which accesses it is run.

If the join conditions imposed on the working set of forms are "nice" then each connected subgraph of the instance graph will also be a subgraph of the sketch graph. It is conceivable, however, that two forms satisfying a particular precondition sketch for a procedure may each satisfy a join condition with a third form which satisfies the local conditions for a second sketch in that procedure. This anomaly will occur either if the imposed join conditions are "not nice enough", or if duplicate forms are inadvertently created and passed through the system. In this case, the connected subgraphs of the instance graph are not as simply related to the sketch graph as before. Thus, establishing when a complete working set of forms has been compiled requires careful analysis. (See the last example above.)

One may assume that, whenever a working set is found, it is processed and leaves the domain of our station's collection of automatic procedures. At any given time, then, when TLA has finished processing a form and has not yet begun to process the next form, we know that the instance graph contains no copies of the sketch graph. If a copy of the sketch graph is identified, then a working set has been found, the procedure is executed, and the corresponding nodes and edges are purged from the instance graph so that no more working sets remain. When a new form arrives, a working set of forms may be completed only if that new form is included. The analysis of the instance graph, then, need only concern the connected subgraphs which include nodes representing the new form.

One would expect join conditions giving rise to sketch trees to be most common, since the "cheapest" description of the relationships between sketches would contain no cycles. If A is related to B and B is related to C, then one would hope not to find any other relationship holding between A and C other than the transitive one. In practice, however, things may not be that simple. Join conditions might give rise to cycles, or even disconnected sketch graphs, as mentioned earlier.

If the warehouse has a single "Value" form at its workstation keeping track of the total dollar value of its stock, then procedures which update it would include a blank precondition sketch for a "Value" form. Since there is no confusion about which Value form is needed, there are no local or global conditions to be specified for it. The corresponding sketch graph in figure 12 is therefore disconnected.

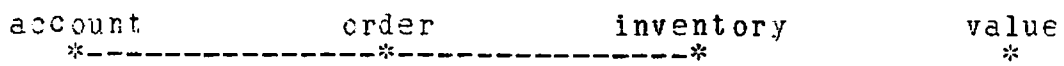


Figure 12 A disconnected sketch graph

Furthermore, if customers had separate accounts for each item they order (granted, a preposterous example under most circumstances), then a link between account and inventory accross item number would create a cycle in this sketch graph.

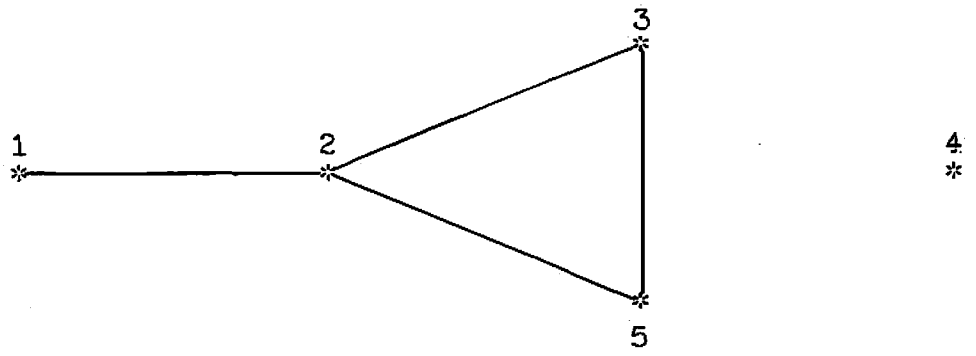
4.2.3. Graph-chasing

The algorithm which searches the instance graph for a copy of the sketch graph employs a list of potential working dossiers. Initially there exists a single such dossier containing only the key of the newly added form. Edges are traversed in the instance graph and keys are added to each dossier until all the edges and nodes in the sketch graph have been checked.

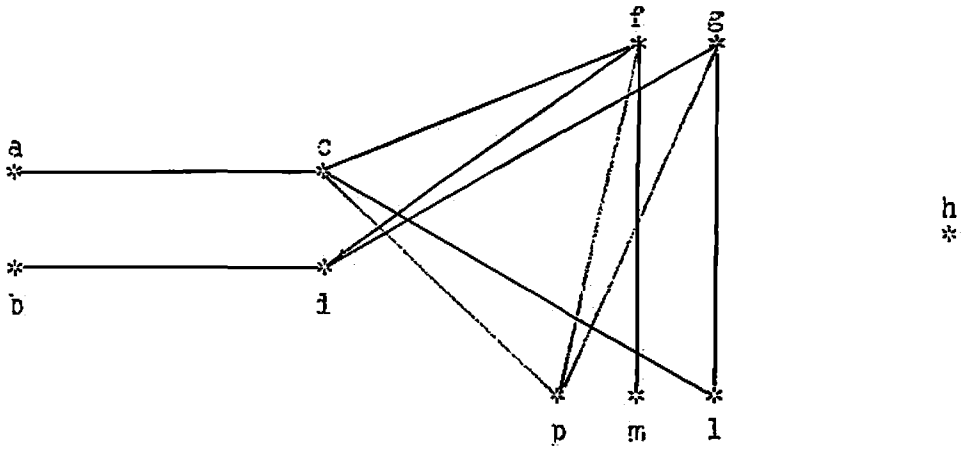
Conceptually, we start at the node of the sketch graph corresponding to the new form. We traverse edges leading out from that node, and check off any new nodes that we reach. We may follow any previously untraversed edges leading from any node we have thus far reached. Edges will lead back to old nodes wherever cycles occur. If the sketch graph is disconnected, then the subgraph containing the first node will be traversed first. Edges not in that subgraph, of course, cannot lead from old nodes until an edge is traversed which checks off two new nodes.

The sketch and instance graphs in figure 13 will be used to illustrate the graph-chasing algorithm. The example contains both cycles and disjoint subgraphs, but is not intended to necessarily correspond to a particular real-life example.

Sketches 3 and 5 are sketches for the same form type but represent distinct forms in the procedure. {a, b, c, ...o} are keys belonging to forms that match the local conditions of the sketch graph. Form a, for example, matches sketch 1. Edges in the instance graph represent joins. Forms c and f, for example, satisfy the global conditions between sketches 2 and 3.



Sketch graph (type(3) = type(5))



Instance graph (p is the most recently added node)

Figure 13 Sample sketch and instance graphs

The addition of form p results in the completion of the working dossier (a,c,f,h,p) where previously no complete working dossier existed. The algorithm presented here will identify this set of forms.

As we trace a path through the sketch graph, we try to mimic our actions non-deterministically in the instance graph. If we

follow an edge in the sketch graph, we attempt to follow that edge in the instance graph for each dossier in our list. For each success we add a new key to some dossier, and for each failure, we delete a dossier. Whenever several edges may be traversed in the instance graph for a given edge of the sketch graph, we split the current dossier and add a new node for each copy of that dossier. The closing of a cycle in the sketch graph corresponds conceptually to a select on the dossier list, ensuring that links actually exist in the instance graph for the two relevent forms represented in each dossier.

Figure 14 describes the steps followed in locating the working dossier in our example. If at any point we lost all our working dossiers, the algorithm would halt with no working set of forms identified.

The sketch and instance graphs are described as follows: The sketch graph is $G'(N',E')$ where $N' = \{1, \ldots n\}$ is the set of colours and E' is a subset of $N' \times N'$ containing no (i, j) where $i = j$. F is the set of form keys. The instance graph is $G(N,E)$ where N is a subset of $N' \times F$ and E is a subset of $N \times N$. Furthermore, we adopt the convention that if $x = (i, k)$ belongs to N, then $x' = i$ and $x'' = k$, and if $e = (x, y)$ belongs to E, then $e' = (x', y')$.

In the example,

$N' = \{1,2,3,4,5\}$,

$E' = \{(1,2), (2,3), (3,5), (2,5)\}$,

$F = \{a,b,c,d,f,g,h,l,m,p\}$,

potential working dossiers 1 2 3 4 5	
p	p is a new form matching sketch 5.
f p g p	From node 5 in the sketch graph we can reach node 3 along edge (3,5). We follow ((3,f),(5,p)) and ((3,g),(5,p)) in the instance graph and "split" our potential working dossier.
c f p d f p d g p	We now follow edge (2,3), splitting the first dossier of the previous step.
a c f p b d f p b d g p	Follow edge (1,2).
a c f p	Edge (2,5) completes a cycle. We perform a select on the dossiers resulting from the last step. Since ((2,d),(5,p)) is not in the instance graph, we lose two potential working dossiers.
a c f h p	We have traversed all the edges in the sketch graph and need to add a form that matches node 4.
a c f h p	Check that form f differs from form p.

Figure 14 Finding a working set of forms

$N = \{(1,a), (1,b), \ldots(5,p)\}$, and

$E = \{((1,a),(2,c)), ((1,b),(2,d)), \ldots((2,c),(5,p))\}$.

We note, then, that for each x in N, x′ must belong to N′, and for each e in E, e′ must belong to E′ -- i.e. nodes and edges in the instance graph correspond to nodes and edges of the sketch graph.

If finding a complete set of forms is equivalent to locating an instance of the sketch graph within the instance graph, we can express this as follows: We seek all subsets N" of N such that (1) $\{x' | x \text{ in } N"\} = N'$ and (2) for each (i, j) in E', there exists x and y in N" such that x' = i, y' = j and (x, y) belongs to E -- i.e. for each node and edge of the sketch graph there exist unique corresponding nodes and edges in the spanning graph G'[N"].

In the example

$$N" = \{(1,a), (2,c), (3,f), (4,h), (5,b)\}.$$

The algorithm for finding all such subsets N" makes use of the knowledge that any working set of forms must include the most recently added node, say x. Furthermore, there are two check-lists, node and edge, with slots for each element of N' and E' respectively, all initially set to false, and a dossier list, D, initially empty. Each dossier has n slots to hold all the keys of any working set of forms found by the algorithm:

Suppose x in N represents the newly added form.
Add a dossier to D, with slot x' set to x": x must belong to
    the working set.
Set node[x'] to true: check off node x' of the sketch graph.
For each e = (i, j) in E' such that edge[e'] is false do
{
    if both node[i] and node[j] are false
    then
        make one copy of each dossier in D for each

```
            (y, z) in E where y' = i and z' = j,
            setting slots i and j to y" and z".
        else if exactly one of node[i] and node[j] is false
            (without loss of generality, node[i])
        then
            for each dossier in D make one copy for
            each (y, z) in E where y' = i, z' = j, and
            y" is already in slot i of the dossier,
            setting slot j to z".
        else if node[i] and node[j] are true
        then
            for each dossier in D delete the dossier
            if (y, z) is not in E where y" and z" are
            in slots i and j of the dossier.
        endif.
        set edge[e'] to true
        set node[i] to true
        set node[j] to true
}
```

Check that forms of the same type are different.

In the above algorithm, wherever the words "make one copy for each..." occur, the concerned dossiers are deleted if no copies can be made. If D is empty when the algorithm is finished, then no working dossiers were found. If D is not empty, then the "first" dossier containing no duplicate keys is chosen as the working dossier.

Since the station's owner may have moved some of the forms in the working set while the interpreter was running, each of the forms must be grabbed before the actions may be performed. Each form in the working set is deleted from the system so that the only copy is the interpreter's image of the form. If any of the forms cannot be found, then the interpreter restores all the forms grabbed thus far, and aborts the forms procedure.

4.3. Actions

Only if all the forms are successfully grabbed does the interpreter perform the set of actions. In the translation phase, the legality of actions, implied actions and a legal order of actions have already been determined.

Actions may "fail" if a string is too long to be inserted in a given field, or if a form is mailed to a non-existent station. In the former case, TLA chooses to insert the null string by default, with the understanding that both humans and procedures are intelligent enough to interpret this not as a value, but as a non-value. In the latter case, OFS (and consequently TLA) returns the mail to the sending workstation. Since TLA procedures are capable of recognizing the source of mail, it is presumed that this anomaly could be appropriately dealt with if a user felt it necessary.

5. Conclusions

TLA captures a very limited sense of what is meant by an "automatic forms procedure". The context of OFS limits the range of possible actions upon forms, but there are still many things that humans can do in OFS which have not been modelled in TLA. Automatic procedures, for example, are not smart enough to expect the return of a form which has been shipped away, and subsequently take some action if a response is not received within some desired turnaround time.

Form flow is determined by the particular configuration of procedures across the system, but neither analytic nor heuristic tools are available for determining any notion of "correctness". It is the responsibility of the users and a form administrator to guarantee that there are no undesirable side effects resulting from some particular combination of automatic procedures. Whether such analysis could be performed within any reasonable complexity bound, or even if it could be performed mechanically at all is not known, since the meaning and domain of "correctness" is not defined in the general case, and perhaps not even for any given application.

The complexity of interpreting automatic procedures and form-gathering clearly depends on (1) the size of the working set for a procedure, (2) the number of automatic procedures running at workstations, and (3) the number of form images "waiting" in the instance graphs of a workstation. As pointed out in an earlier section, the complexity of identifying an instance of the

sketch graph within the instance graph grows if the join conditions are so peculiar that the instance graph is not merely a subgraph of the sketch graph. Obviously, whatever factors contribute to this complexity must be considered in any "good office design". Performance in an electronic office will be degraded by poor distribution of automation, but exactly what constitutes "good design", and to what extent it is feasible for a given application is not yet known.

Partly completed working sets of forms may or may not have a particular meaning in terms of exceptions and errors. If forms are "missing" from a working set, the forms that are there may also be part of another working set. The missing forms would determine which procedure is to be activated. As such, there is no way of telling which procedure forms are missing until they arrive. Also, missing forms may or may not eventually arrive, and there is no way of interpreting their absence as an error, except by placing some arbitrary time limit upon form-gathering.

Since forms may satisfy partly completed working sets for a number of procedures, there would be a need for some convenient way of displaying these sets so that users could interpret what is "missing" and possibly act on this information. Instance graphs could be quite complicated in general, and several partly completed sets may overlap in a single instance graph. It seems that a graphic display would be better suited to presenting this information than lists of form keys, since the splitting cannot be presented linearly.

A simple feature that would increase user interaction with automatic procedures would be a function whose value is determined by the user. When the interpreter sees this function assigned to a field in an action sketch, it holds all the forms in the working set, notifies the user when he next signs on, and waits until the user makes a request to inspect the working set. At that point the user is allowed to assign a value to the field (or possibly abort the procedure), and then execution will resume.

Form flow in TLA is determined by the configuration of automatic procedures, and triggering of procedures takes place when combinations of forms arrive at a workstation. Flow of execution could be made more explicit by passing control between procedures. Within a single workstation, then, one could pass working sets of forms and subsets thereof between procedures, thus explicitly determining the order of operations without having to over-distribute desk activity within the office. Procedures could then be called from other procedures without the need for form-gathering, since the calling procedure would pass the forms already gathered. Decision points could be modelled by branching rather than by a variety of similar working sets of forms, thus reducing some of the work involved in form-gathering. Which procedure is to be called could be decided by evaluating a function whose arguments are field values from the working set.

We have not answered what degree of generality is required for procedure specification in the electronic office, but a small prototype has been presented which solves some of the problems,

and suggests approaches for providing other useful features. A framework is needed for describing form flow and automatable procedures in an office so that notions of correctness may be analysed for a given model.

6. Bibliography

ATTA80
Attardi, G., Barber, G. and Simi, M., "Towards an Integrated Office Work Station", MIT, 1980.

CHEU79
Cheung, C., "OFS -- A Distributed Office Form System with a Micro Relational System", M.Sc. thesis, Department of Computer Science, University of Toronto, 1979.

CHEU80
Cheung, C. and Kornatowski, J., The OFS User's Manual, Computer Systems Research Group, University of Toronto, 1980.

DEJO80
de Jong, P., "The System for Business Automation (SBA): A Unified Application Development System", Information Processing 80, Lavington, S.H. (ed.), North-Holland, The Hague, 1980.

ELLI79
Ellis, C.A. and Nutt, G.J., "Computer Science and Office Information Systems", Computing Surveys, March 1980.

GIBB79
Gibbs, S., "OFS: An Office Form System for a Network Architecture", M.Sc. thesis, Department of Computer Science, University of Toronto, 1979.

GIBB80
Gibbs, S., The OFS Programmer's Manual, Computer Systems Research Group, University of Toronto, 1980.

HAMM77
Hammer, M., Howe, W.G., Kruskal, V.J. and Wladawsky, I., "A Very High Level Programming Language for Data Processing Applications", Comm ACM 20, 11 (1977), pp. 832-840.

HAMM79
Hammer, M. and Kunin, K.S., "Design Principles of an Office Specification Language", MIT paper, 1979.

HOGG81
Hogg, J., "TLA: A System for Automating Form Procedures", M.Sc. thesis, Department of Computer Science, University of Toronto, 1981.

HUDY78
Hudyma, R., "Architecture of Microcomputer Distributed Database Systems", M.Sc. thesis, Department of Computer Science, University of Toronto, 1978.

HUDY80
Hudyma, R., "The Hardware Design of Distributed Office Workstations" in A Panache of DBMS Ideas III, Technical Report 111, Computer Systems Research Group, University of Toronto, 1980.

KERN78
Kernighan, B.W. and Ritchie, D.M., The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1978.

KORN79
Kornatowski, J.Z., The MRS User's Manual, Computer Systems Research Group, University of Toronto, 1979.

LADD79
Ladd, I., "A Distributed Database Management System Based on Microcomputers", M.Sc. thesis, Department of Computer Science, University of Toronto, 1979.

LADD80
Ladd, I. and Tsichritzis, D., "An Office Form Flow Model" in 1980 NCC proceedings.

METC76
Metcalfe, R.M. and Boggs, D.K., "Ethernet: Distributed Packet Switching for Local Computer Networks", Comm. ACM 19, 7 (1976), pp. 364-404.

MORG80
Morgan, H.L., "Research and Practice in Office Automation", Department of Decision Sciences, The Wharton School, University of Pennsylvania, Philadelphia, PA, USA, 1980.

PETE77
Peterson, J.L., "Petri Nets", ACM Computing Surveys 9, 3 (1977), pp. 223-252.

RITC78
Ritchie, D.M. and Thompson, K., "The UNIX Time-Sharing System", The Bell System Technical Journal, Vol. 57, #6 (July-August 1978), pp. 1905-1929.

ZISM77
Zisman, M.D., "Representation, Specification and Automation of Office Procedures", PhD dissertation, Wharton School, University of Pennsylvania, 1977.

ZLOO75
Zloof, M.M., "Query by Example", AFIPS Conference Proceedings, Vol. 44, 1975 NCC.

ZLOO80
Zloof, M.M., "A Language for Office and Business Automation", IBM Research Report, IBM Thomas J. Watson Research Centre, Yorktown Heights, New York, USA, 1980.